Contents lists available at ScienceDirect

The Journal of Systems and Software

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

VISOR: A fast image processing pipeline with scaling and translation invariance for test oracle automation of visual output systems

M. Furkan Kıraç, Barış Aktemur, Hasan Sözer*

Ozyegin University, Nişantepe Mah. Orman Sk. No: 34-36, Alemdağ – Çekmeköy 34794, İstanbul, Turkey

ARTICLE INFO

Article history:
Received 7 August 2016
Revised 11 June 2017
Accepted 12 June 2017
Available online 13 June 2017

Keywords:
Black-box testing
Test oracle
Computer vision
Image processing
Test automation

ABSTRACT

Test oracles differentiate between the correct and incorrect system behavior. Hence, test oracle automation is essential to achieve overall test automation. Otherwise, testers have to manually check the system behavior for all test cases. A common test oracle automation approach for testing systems with visual output is based on exact matching between a snapshot of the observed output and a previously taken reference image. However, images can be subject to scaling and translation variations. These variations lead to a high number of false positives, where an error is reported due to a mismatch between the compared images although an error does not exist. To address this problem, we introduce an automated test oracle, named VISOR, that employs a fast image processing pipeline. This pipeline includes a series of image filters that align the compared images and remove noise to eliminate differences caused by scaling and translation. We evaluated our approach in the context of an industrial case study for regression testing of Digital TVs. Results show that VISOR can avoid 90% of false positive cases after training the system for 4 h. Following this one-time training, VISOR can compare thousands of image pairs within seconds on a laptop computer.

1. Introduction

Testing activities are essential to ensure the reliability of systems. Increasing system size and complexity make these activities highly expensive. It was previously reported (Beizer, 1990; Myers et al., 2012) that testing can consume at least half of the development costs. A typical approach for reducing this cost is to adopt test automation (Berner et al., 2005; Rafi et al., 2012). This can involve the automation of a set of various activities such as the generation of test inputs/cases, execution of these on the system under test, and the verification of the results. Hereby, the last activity is performed by a so-called *test oracle* (Howden, 1978) that differentiates the correct and incorrect behavior of the system.

A recent survey (Barr et al., 2015) suggests that the problem of automating test oracles has received significantly less attention in the literature compared to the other aspects of test automation. However, test oracle automation is essential to achieve overall test automation. Otherwise, the tester has to manually check the system behavior for all test cases.

One might assume the availability of formal specifications of intended system behavior and/or annotations of pre/post-conditions in source code (Meyer, 1988). In such cases, test oracle automation becomes a straightforward comparison task. However, this assumption is mostly not applicable in state-of-the-practice (Barr et al., 2015). One might also use metamorphic testing techniques (Segura et al., 2016) that rely on metamorphic relations as derived invariants of correct system output. These techniques have proven to be practical and effective in various application domains (Segura et al., 2016; Zhou et al., 2016). Hereby, the biggest challenge is the discovery of metamorphic relations relevant for the domain (Barr et al., 2015).

Test oracle automation is especially hard when the evaluated system output is not unique/exact and when it takes complex forms such as an image (Delamaro et al., 2013). Under these circumstances, test oracles cannot perform trivial comparisons with respect to a reference output. Otherwise, they tend to be fragile and they lead to many false positives (Garousi and Mntyl, 2016).

In this paper, we focus on *black-box* testing of software-intensive consumer electronics. These embedded systems are subject to regular regression testing activities without any access to the source code or internal execution platform. Furthermore, these systems work with a variety of screen sizes. Test cases are evaluated by taking a *snapshot* of the graphical user interface, and comparing this snapshot with a previously taken reference image that serves as the expected output. There exist tools to perform image

* Corresponding author.
E-mail address: hasan.sozer@ozyegin.edu.tr (H. Sözer).

http://dx.doi.org/10.1016/j.jss.2017.06.023
0164-1212/© 2017 Elsevier Inc. All rights reserved.

comparisons between such image pairs, e.g. Perceptual Image Diff,[1] DSSIM,[2] ImageMagick.[3] However, these comparisons lead to a high number of false positives (Selay et al., 2014; Lin et al., 2014). That is, an error is reported due to a mismatch between the compared images although an error does not exist. As a result, images corresponding to each reported error should be manually inspected to identify which of them should be classified as an error or not. This process requires considerable time and effort.

We observed that false positives are mainly caused by scaling and shifting between the reference image and the snapshot. Systems that are robust under both shifting and scaling of an image are said to be invariant to translation and scale transformations. In this paper, we analyze these problems and build a pipeline of image processing techniques to reduce false positives. We particularly design and implement the pipeline to run fast. Hereby, we do not aim at providing contributions in the area of image processing. However, existing techniques in this domain have not been systematically reviewed for addressing relevant problems in test oracle automation. To the best of our knowledge, there does not exist any published work that focuses on this problem. Moreover, the solution is less likely to be introduced effectively in industries because test automation is mainly performed by software and/or test engineers, whereas an effective solution for this problem should benefit from a strong background on image processing and computer vision.

We illustrate and evaluate the effectiveness of our approach in the context of an industrial case study. We collected thousands of captured and reference image pairs that are used for automated regression testing of a commercial Digital TV system. We counted the number of false positive cases caused by the comparison of these images. Results show that more than 90% of these cases can be avoided when we use our approach. The comparison of all the image pairs can be completed within seconds on a laptop computer. The system requires a one-time training step for parameter calibration if/when the system or environment changes. This step takes 4 h.

In the following section, we provide the problem statement and introduce our industrial case study. In Section 3, we explain our overall approach and the image processing techniques we employed. In Section 4, we present and discuss the results obtained based on the case study. In Section 5, we summarize the related work, and discuss why they fall short in our problem context. Finally, in Section 6, we provide our conclusions.

2. Industrial case study: digital TV systems

In this section, we introduce an industrial case study that serves as a running example for both illustrating the problem context and evaluating the adopted techniques. We focus on automated regression testing of Digital TV (DTV) systems. In particular, we investigated the testing process of DTV systems at Vestel Electronics,[4] which is one of the largest TV manufacturers in Europe. DTV systems have become complicated software systems, including over ten million lines of code. In addition to conventional TV functionalities, they provide features such as web browsing, on-demand streaming, and home networking (Sivaraman et al., 2001). The number of such additional features is increasing day by day. This trend makes it essential to employ efficient testing techniques to ensure the product quality despite limited resources. In the following, we first describe the current practice of testing in the company, which defines the context of our case study. We then discuss the observed issues, in particular those that are related to test oracle automation. Finally, we describe the data set we collected for our case study.

2.1. The current practice

The current practice of testing for DTVs at Vestel Electronics involves the following activities:

1. *Specification:* A test engineer prepares test scenarios in the form of executable test scripts. These scripts are later executed by an in-house-developed test automation tool that drives test execution by sending a sequence of remote controller key press events to the TV. Test scripts for some of the TV features are automatically generated by adopting model-based testing. For these features, first, test models are created by specifying the possible usage behavior. Then, a set of purchased as well as in-house-developed tools (Gebizli and Sozer, 2014; Gebizli et al., 2015, 2016) are employed to refine these models and automatically generate test scripts.
2. *Reference image collection:* The test scenarios are executed on a "master" TV that is known to function correctly. At certain points during the execution of the test scenarios, snapshots of the TV are taken to serve as reference images. These points of execution are hard-coded in test scripts by the test engineers. For some cases, the test engineer also prepares a *mask* to accompany the reference image. In that case, some portions of the image are not used for comparison during testing.
3. *Testing:* When a TV is to be tested, the same test scenarios are executed. At the same points when reference images were captured from the master TV, screenshots of the TV under test are taken. A snapshot is then compared with the corresponding reference image.

Test scenarios involve validating various GUI features of the TV, such as checking whether the correct text and shapes are displayed, color and transparency rendering is as expected, hyper text is visualized correctly, text wrapping is proper, menu items are positioned appropriately, and response to certain user-interaction events are as defined.

During reference image collection and testing, a snapshot of the screen is taken using one of the following three methods:

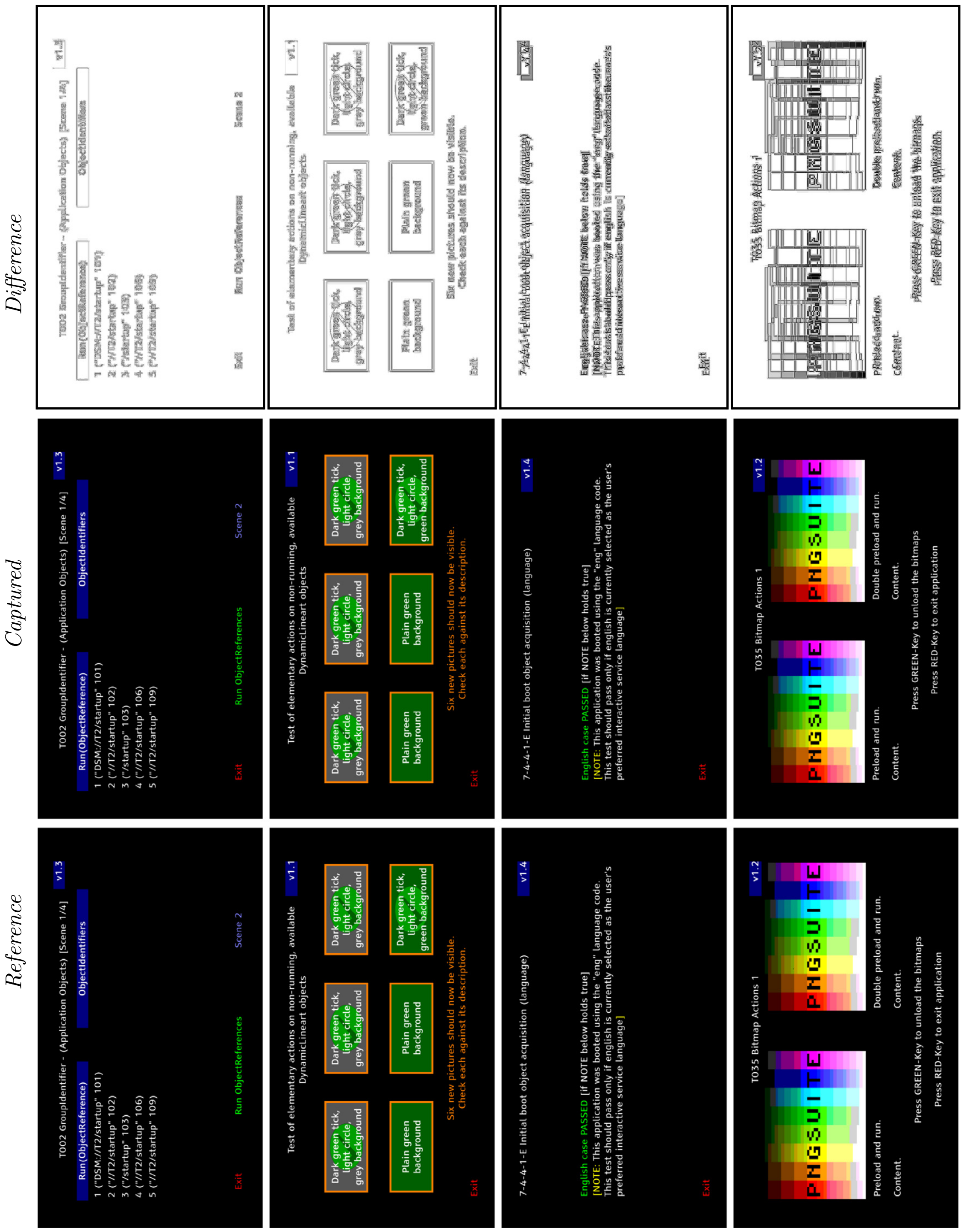
- Via a connection made to the TV through a peripheral port – usually the ethernet.
- Via an LVDS (Low-Voltage Differential Signaling) reader – a device that intercepts and reads the signals going to the LCD (Liquid Crystal Display) panel. The device has to be installed on the cable that goes into the panel.
- Via an external camera.

Test engineers prefer the first method listed above, because it requires the least installation effort, and exact snapshots can be taken. However, such a port is not always available, because it just does not exist or it is occupied for another purpose during testing. LVDS reader is the second favorite option, because it still gives the opportunity to take high quality snapshots even though images occasionally show salt-and-pepper noise or similar problems. The drawback of this method is that it is costly or even infeasible to install the LVDS device on the TV. The third option, using an external camera, is the least favorite because the snapshots taken with this method show a great variation in illuminance, color tones, and the view angle.

2.2. Observed issues regarding test oracle automation

During testing, a test oracle compares the captured snapshot with respect to the previously taken reference image. An exact

[1] http://pdiff.sourceforge.net.
[2] https://github.com/pornel/dssim.
[3] https://www.imagemagick.org.
[4] http://www.vestel.com.tr.



T002 GroupIdentifier - (Application Objects) [Scene 1/4] v1.3
Run(ObjectReference) ObjectIdentifiers
1 ("DSM://T2/startup" 101)
2 ("//T2/startup" 102)
3 ("/startup" 103)
4 ("//T2/startup" 106)
5 ("/T2/startup" 109)
Exit Run ObjectReferences Scene 2

Test of elementary actions on non-running, available DynamicLineart objects v1.1
Dark green tick, light circle, grey background
Dark green tick, light circle, grey background
Dark green tick, light circle, grey background
Plain green background
Plain green background
Dark green tick, light circle, green background
Six new pictures should now be visible. Check each against its description.
Exit

7-4-4-1-E Initial boot object acquisition (language) v1.4
English case PASSED [if NOTE below holds true]
[NOTE: This application was booted using the "eng" language code.
This test should pass only if english is currently selected as the user's preferred interactive service language]
Exit

T035 Bitmap Actions 1 v1.2
PNGSUITE PNGSUITE
Preload and run. Double preload and run.
Content. Content.
Press GREEN-Key to unload the bitmaps
Press RED-Key to exit application

Fig. 1. Sample test cases that incorrectly failed due to scaling, shifting and/or anti-aliasing differences between the reference and the captured images.

match of the compared images is expected as the pass criterion for the test. As a result, minor differences between the images (e.g., slight variations in scale, anti-aliasing, etc.) cause a test to be deemed failing, even though there exists no behavioral error of the system. Consequently, the test oracle yields a high number of false positives. This drawback is also acknowledged in the literature (Garousi and Mntyl, 2016).

Fig. 1 illustrates 4 cases that incorrectly failed based on exact comparison of the reference and captured images. The differences are caused by rendering issues (i.e. anti-aliasing of characters and shape edges, different font kerning settings) as in rows 1 and 2, or shifting and scaling issues as in rows 3 and 4. Some test cases contain text only (rows 1 and 3), while others may also contain shapes (rows 2 and 4). Fig. 2(a) shows an example case where a mask is used. The black regions in a mask are excluded during image comparison.

There are several types of changes that lead to fragile tests. One of them is related to different screen sizes. The captured image might not match with the reference image if it was captured from a TV with a different screen size, or by a camera with different intrinsic parameters (e.g. different lens, different sensor, etc.). We refer to this type of change as *scaling*, in which the aspect ratio of

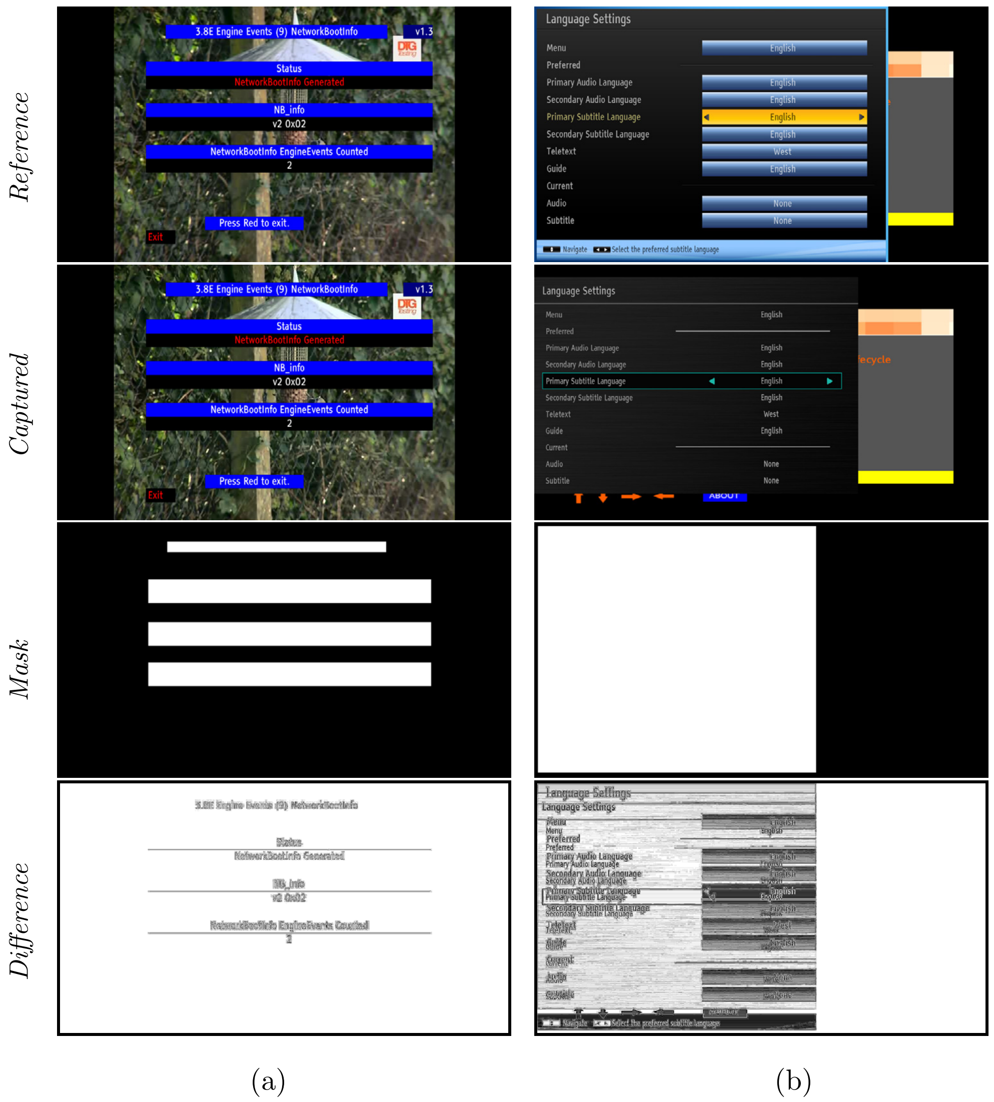


Fig. 2. Two test cases where (a) a mask is used, (b) there is a GUI theme change.

the original screen is not preserved. Even two cameras with exactly the same brand and model may differ in scaling due to their varying intrinsic parameters. As another change, the captured image can be shifted with respect to the reference image. We refer to this type of change as *translation*. A common source of change is the anti-aliasing and rendering differences caused by the use of a different font and/or rendering parameters (such as kerning) on the TV under test. These issues also result in scaling and translation-related differences.

Yet another source of change is the difference in the user interface of the product. Vestel produces DTV systems for 157 different brands in 145 countries worldwide. Although the functionality is similar, products manufactured for different customers can have different GUIs. For example, a button can be moved from the top-left corner of the screen to the top-right. These variations also lead to fragile tests. Fig. 2b depicts a case where the captured image differs significantly from the reference image due to a change in the GUI theme. We consider this problem as out-of-scope for this work. We focus on handling *translation and scaling* changes only.

Existing visual test automation tools (Memon et al., 2000; Eaton and Memon, 2007; Alégroth et al., 2015; Leotta et al., 2016) cannot be applied for testing DTV systems. These tools are supposed to run on the same machine as the system under test where they have access to the GUI of the system. Unlike desktop applications and Web applications, it is not possible to access GUI widgets of DTV systems, such as buttons and text boxes, to control and monitor them. The main resource problem with testing DTVs is that a peripheral port, such as the ethernet port, is not always available. This prevents the testers from accessing the software, regardless of the operating system. It is also not an option to run the tests on an emulator (e.g., as done by Sikuli (Chang et al., 2010) for testing Android apps); black box tests for DTV systems are always performed on real devices to be able to capture errors due to external factors and hardware issues, which cannot be all represented in an emulator.

2.3. The collected data set

We collected a data set to evaluate the accuracy of the currently applied approach and our approach. This data set involves a set of reference and captured image pairs. These image pairs were actually used as part of test scripts that are executed for a real TV system in the company. A test engineer within the company manually examined these image pairs and marked them as either a *correct execution* or a *failure case*. We obtained the verdicts of the currently

Table 1
Categorization of the collected data set when using exact image matching.

	TN	TP	FN	FP
Test scenarios	∼ 1500	6	0	216
Image comparisons	N/A	96	0	2563

employed test oracle for these image pairs as well. Recall that this oracle gives a verdict according to *exact* image comparison.

According to the verdict of the test oracle, test cases are categorized as *true negative* (TN), *true positive* (TP), *false negative* (FN) and *false positive* (FP) based on the following definitions that we use throughout the paper:

- **TN:** An error does *not* exist, and the oracle did not report an error.
- **TP:** An error exists, and the oracle reported an error.
- **FN:** An error exists, but the oracle did *not* report it.
- **FP:** An error does *not* exist, but the oracle reported an error.

Table 1 summarizes the results. Hereby, the first row lists the number of test scenarios (scripts) for each of the TN, TP, FN and FP categories. Each test scenario usually includes more than one image comparison at different points of execution. Hence, the second row separately lists the total number of image comparisons made. Note that the FN category trivially contains no cases. This is because the current oracle gives verdict according to exact image comparison; if there is actually an error, the reference and the captured images must differ, and the oracle catches an error.

We have been informed that there are approximately 1500 TN scenarios; the exact number in this category and the number of image comparisons per each of these test scenarios were not disclosed to us. We collected 6 sample TP test scenarios, each representing a different cause of error. These errors are related to accent character rendering, geometrical shape display, digital text rendering, incorrect screen output due to functional error, missing text, and UI changes. There are a total of 96 image comparisons performed in the 6 TP test scenarios. We collected 216 FP test scenarios in which a total of 2563 image comparisons are performed. All these cases have been manually examined and labeled as TP/FP by a test engineer at Vestel.

In this work, our goal is to reduce the number of FP cases, because manually examining these to determine whether they are true or false positives is a time-consuming task for the testers. To this end, we focused on the FP cases we received from the company. We observed that, although there are cases that fail because of a change in the GUI theme (e.g. Fig. 2(b)), or color and transparency differences, the majority of the FP cases (incorrectly) failed due to changes caused by anti-aliasing, shifting, and scaling issues. Therefore, we focus on reducing FP's caused by *translation and scaling* reasons; we do *not* attempt fixing problems related to color or GUI layout.

3. Test oracle automation with VISOR

We implemented our approach as a tool, named VISOR. The general structure of the tool is shown in Fig. 3. VISOR takes 3 inputs for each test case: (*i*) The image that is captured during test execution. (*ii*) The previously captured *reference image* that serves as the ground truth. (*iii*) An optional *mask image* that visually specifies the regions within the reference image that should be included or excluded for comparison. The only output of VISOR is a *verdict* regarding the success or failure of a particular test case.

Fig. 3 shows a series of image filters and transformations employed by VISOR. Recall that our problem context involves usage of captured images of Digital TV screens, which are prone to illumination, translation, scaling variations, and noise. VISOR's pipeline has been specifically designed to address the image differencing problems induced by translation and scaling reasons, which comprise the majority of causes that lead to false positives according to our observations. On the other hand, VISOR has an extendable and adaptable architecture. One can add/remove/replace filters or adjust parameters to apply the approach in another context. That is, VISOR is a generic pipeline and we implemented an instance of it for the industrial case.

The main idea behind our pipeline is based on finding the salient sub-region (SSR) that contains the foreground items in both the reference and captured images, aligning the SSRs of the corresponding images, and then comparing the aligned regions. Precision and accuracy of SSR extraction and alignment phase is vital for the overall effectiveness since perceptual image differencing algorithms depend heavily on exact alignment of images (Yee et al., 2001; Yee, 2004). For our problem context, we found that using a single rectangular region as SSR gives good results.

In the following, we discuss each step of the pipeline in detail by evaluating alternative techniques that can be adopted.

3.1. Background color removal

This is the first filter we apply in our pipeline. Although the background color is black for most of the data set images, there are also numerous cases where the background is different. Background color needs to be removed before detecting the salient regions of both frames. Background segmentation is a deeply studied topic in computer vision; there are numerous methods for it (Benezeth et al., 2010). Some of them learn a background model from a supervised training data set, while others use temporal information in video frames. Principal Component Analysis (PCA) (Wold et al., 1987) has been successfully applied to training data set for reducing the dimensionality of vector space represented by concatenated image pixels. Since background pixels are assumed to be stationary most of the time, PCA application learns a model of background pixels that repeat in the data set. Any behavior that cannot be represented by the PCA model is considered an outlier, and detected as a foreground pixel. An alternative approach is to model each pixel in an image by a Gaussian Mixture Model (GMM), meaning that a pixel can take values sampled from a GMM (Lee, 2005). If the pixel is outside of the range of GMM, it is considered to be an outlier pixel, hence, a foreground pixel. Both of these methodologies apply either to video frames or to data sets that contain samples significantly similar to each other. In our case, each data set image can be substantially different from others. Therefore, we need an algorithm that can work on a single image frame without using temporal information, namely a single-shot segmentation algorithm. Fortunately, the images in our data set have solid and single-colored backgrounds. Hence, we can represent backgrounds by the following straightforward model: Background pixels are the pixels that have the highest frequency in a specific image. This applies for all the images in our data set except a few where text is rendered on a still background picture (e.g. Text layer is transparent and it is rendered on a flower photograph). We did not increase our background segmentation model's complexity for these few cases. If we had more samples with transparent text layer and photographic backgrounds, we could apply a PCA model and still have high accuracy in background detection.

In order to achieve background segmentation, we convert both images to 8-bit single luminance channel images, and create a 256-bin (8-bit) grayscale histogram. We set the most frequent bin of

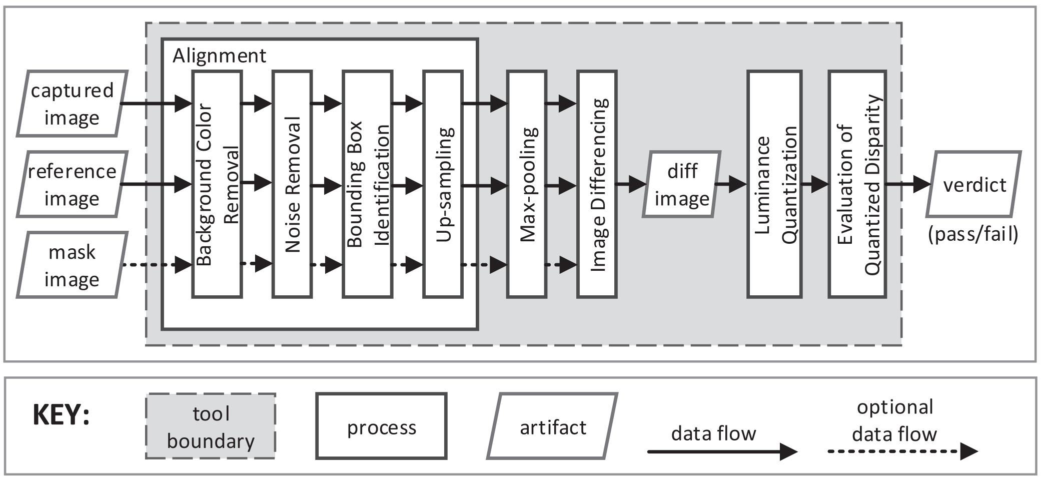


Fig. 3. The general structure and the pipeline of VISOR.

the histogram as the background color, and replace all its occurrences in the images with a sentinel color value. Our subsequent filters use this sentinel value for distinguishing between real foreground and background pixels.

3.2. Noise removal

As a simplification, we assume an SSR to be the maximum enclosing bounding box (MEBB). For this assumption to be useful, no noisy pixels should reside outside the SSR. Hence, as the next step in our pipeline, we perform *noise removal*.

Most of the studies in the literature regarding noise removal deal with complex scenarios where severe noise occurs (Motwani et al., 2004). This is seen, for instance, in scanned historical documents, and also when reconstructing a high resolution image by using a learned mapping between low resolution images and high resolution ground-truth counterparts. We do not see these problems in our data set. We have Full HD (1080p) images captured directly on the TV or with a high quality camera. In our case, we observe "salt and pepper" noise where noise frequency is low but speckle size is big. In other words, we occasionally see areas with slight color variations around the detected background color that are overlapping background pixels. We address this problem by finding the connected components that are inside background regions and erasing them by replacing with the background color. We could alternatively apply a non-linear filter, such as a median filter, that is known for its effectiveness against salt and pepper noise (Gonzalez and Woods, 2008). However, this would impede the speed and damage the text regions since our speckle size is big. That is why we use a more concentrated approach for our data set instead.

Our approach involves creating a *binary* version of the input image first, and then sweeping the noise. A binary image is constructed from a color image by setting a pixel on when the pixel's RGB value is different than the background color and its luminance is less than a threshold T_{lum}. Setting even $T_{lum} = 0$ can be a feasible choice at this step. Choosing a value slightly greater than zero further filters probable noisy artifacts. One should be careful about increasing T_{lum} too much, because this would cause valuable details to be erased along with the noisy pixels. Noise detection is done by finding the connected components of the binarized image, and erasing the regions with area below a certain threshold T_{noise}. The threshold can be adjusted for eliminating noise that can affect the accuracy of MEBB detection.

3.3. Bounding box identification

After we erase the background and remove the noise, we perform *bounding box identification*. This is a straightforward step where we find the MEBB of each image by finding a bounding sub-region that contains all the foreground pixels. This relies on the fact that we segmented all the foreground pixels correctly by finding the background color and erasing the noise in the previous steps.

We apply the detected bounding box on the *original* images; that is, background and noise removal effects are reverted once we find the MEBB.

Following the bounding box identification step, we may consider comparing the regions inside the MEBB's of the reference and the captured images. Although we expect the major scaling and shifting problems to have been cleared out, there can always be minor scaling and translation problems that would cause the FP rate to still remain at an unacceptable level. To remedy such problems, we perform *up-sampling* followed by *max-pooling*.

3.4. Up-sampling

In this step, the MEBB's of the reference and the captured images are scaled to the full image resolution for roughly satisfying the translation and scale invariance in one easy step. For up-sampling, smooth interpolation methods such as bicubic or Lanczos (Fadnavis, 2014) should be used. Nearest neighbor approximation produces artifacts unsuitable for direct image differencing.

3.5. Max-pooling

The last translation step, *max-pooling*, decomposes the input image into a grid of small rectangular blocks of size *B*. Each block is replaced with the maximum pixel value that it contains. (Standard down-sampling replaces a block with the *mean* value.) Max-pooling is a widely used technique in deep learning pipeline of recent state-of-the-art machine learning algorithms. It has been effectively incorporated in various settings such as fast image scanning (Giusti et al.), object recognition (Scherer et al., 2010), and hand gesture recognition (Nagi et al., 2011). Boureau *et al.* discuss a detailed theoretical analysis of performance gains of max-pooling compared to average pooling (Boureau et al., 2010). Although max-pooling causes some information loss in the input images, it provides further translation and scale invariance. It even adds robustness against slight elastic deformations. Perceptual image differencing algorithms (Yee et al., 2001) are based on evalu-

Table 2
The set of VISOR parameters.

Parameter	Description	Range
T_f	Disparity threshold	0–400
T_{lum}	Foreground luminance threshold	0–10
T_{noise}	Noise threshold	10–60
B	Max-pooling block size	8 × 8–40 × 40
Q	Quantization factor	10–60

ating the difference of two aligned images. Max-pooling produces images that are more suitable to be used in image differencing by removing unnecessary detail. Furthermore, because the image size is reduced, this step makes the differencing algorithm run faster. Selecting a proper block size is key to the effectiveness of max-pooling. A large block size erases too much information whereas a small one may not be sufficient for removing sensitivity to translation and scaling.

3.6. Image differencing

This step simply takes the pixel-by-pixel difference of the images received from the max-pooling filter.

3.7. Luminance quantization

The difference image may be histogram-equalized or contrast-stretched for removing possible illumination difference between the reference and captured images. For our data set, we did not directly modify the histograms. Slight illumination differences between the max-pooled reference and captured images are handled by quantizing the difference image. Namely, we divide the per-pixel difference values by a quantization factor, Q. We use integer division; therefore, this step intentionally eliminates difference values below Q. For representing the total difference between images, we sum the quantized per-pixel difference values over the whole difference image, and use this value as our *disparity feature*, $F_{disparity}$.

3.8. Evaluation of the quantized disparity

Image pairs whose disparity value is above a certain threshold, T_f, are deemed “different”, while those with smaller disparity are considered “same”.

3.9. Summary

A running example that shows the stages of the pipeline is shown in Fig. 4. Hereby, the first row shows the original reference and captured images that are supplied as input to VISOR. The last column shows the difference between the two images. The second row shows the transformed images after the alignment step. We can see that the difference between the images turns out to be significantly low compared to the difference between the original images. The last row shows the images after the application of the max-pooling step, at which point almost no difference can be observed.

VISOR has 5 parameters in total as listed in Table 2. Hereby, the T_f parameter is determined based on a training step prior to tests (discussed in the next section) and its range might be increased depending on the resolution of input images. Recall that the amount of difference between two images is quantified as the $F_{disparity}$ value in the last step of the VISOR pipeline. VISOR evaluates the disparity against the threshold, T_f. Optimal values for the other 4 parameters are dependent on the test setup and environment rather than input images. We performed a grid search to determine values of these parameters. We uniformly sampled each of the 4 parameters in their range, and evaluated all the combinations of the sampled values (see Section 4.2.1). The MATLAB code regarding an implementation of our pipeline is available in public domain at http://srl.ozyegin.edu.tr/tools/visor/.

4. Evaluation and results

In this section, we evaluate our approach based on the industrial case study for testing DTV systems. First, we introduce our research questions. Then, we describe the experimental setup. Third, we present the results and interpret them for answering the research questions. Finally, we discuss validity threats for our evaluation and the known limitations of our approach.

4.1. Research questions

Our first dimension of concern regarding VISOR’s performance is the improvement in *accuracy*. We take the accuracy of a test oracle that uses exact image comparison as the baseline for improvement. Our second concern is the *speed*. Because the full test suite processes thousands of images, the efficiency of the test oracle plays a key role in running the test suite in a reasonably short time. We therefore implemented VISOR in C++ and parallelized it, instead of using an interpreted language. Our final concern is the calibration and training time required for VISORbefore it can be used for the actual tests. Although this is a one-time effort, we would like to know the cost of this initial investment. The latter two concerns are related to the acceptability of the tool in the real industrial scenario.

Based on the concerns, we defined the following research questions:

RQ1: To what extent does VISOR improve accuracy with respect to exact image comparison?
RQ2: How much time is required for VISOR to compare two images and provide a verdict?
RQ3: What is the parameter calibration and training overhead?

4.2. Experimental setup

As explained in Section 2, we collected a total of 2659 reference and captured image pairs. These image pairs were actually used during the regression testing of a real TV system in the company. A test engineer within the company manually examined these image pairs prior to our experiment. He labeled each pair as either a *correct execution* or a *failure case*. We used these labeled image pairs as the data set (i.e., objects) of our experiment.

Our experiment does not involve any human subjects. We used the collected image pairs as input to two different tools. First, we used an automated test oracle that employs exact image comparison. As listed in Table 1, 96 image pairs led to TP verdicts, whereas 2563 pairs led to FP verdicts. We took these results as the baseline. Then, we supplied the same inputs to VISOR. To evaluate the improvement in accuracy attained by VISOR, we define the following metrics:

- P_{TP} (TP performance): Ratio of original TP cases that are retained as TP when using VISOR.
- P_{FP} (FP performance): Ratio of original FP cases that became TN when using VISOR.
- *Accuracy*: The ratio of correct verdicts to the total number of image comparisons (i.e., 2659).

We desire both P_{TP} and P_{FP} to be as high as possible. However, there is a trade-off between the two. To understand why, first recall that the amount of difference between two images is quantified as the $F_{disparity}$ value in the last step of the VISOR pipeline.

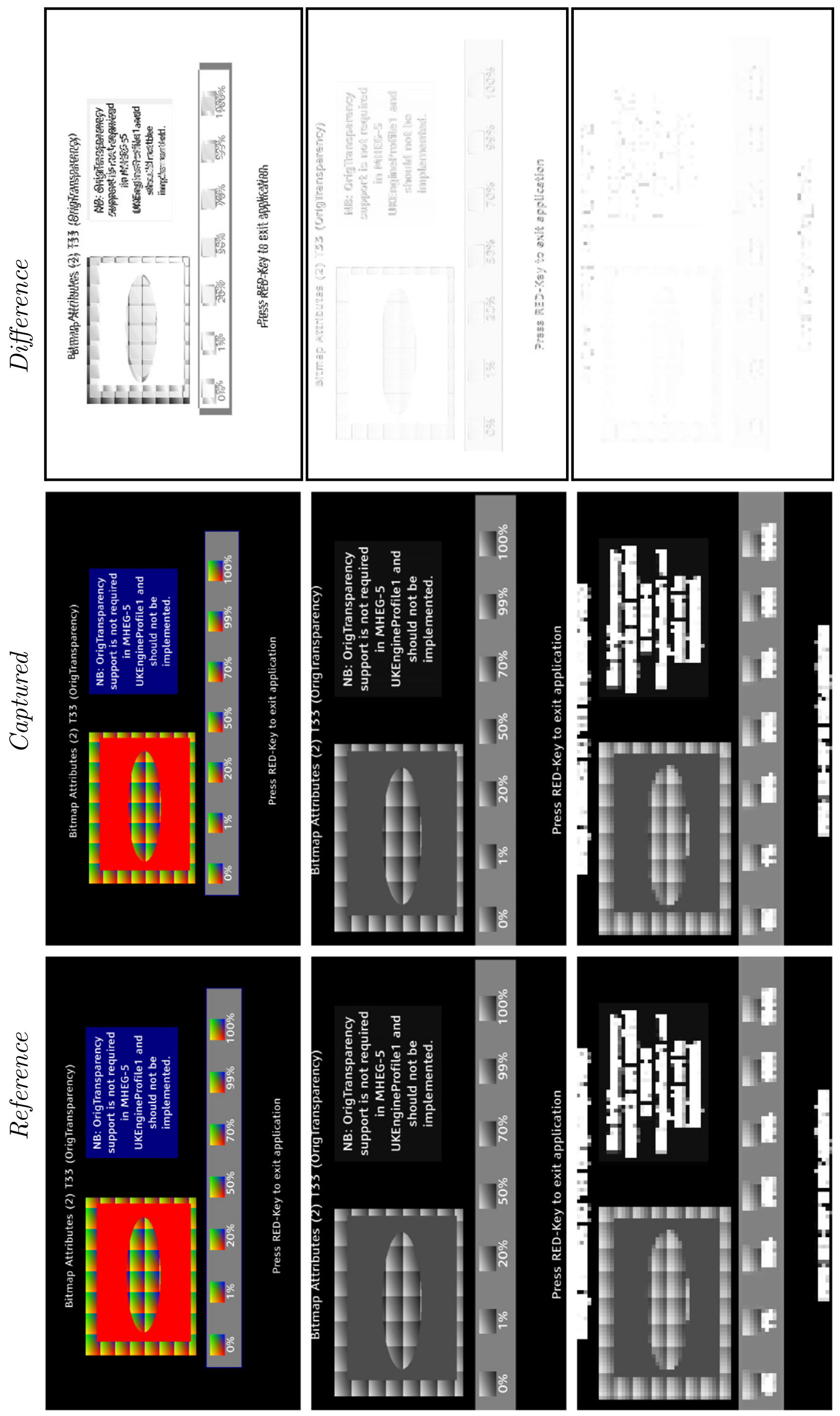

Reference
Captured
Difference
Bitmap Attributes (2) T33 (OrigTransparency)
NB: OrigTransparency support is not required in MHEG-5 UKEngineProfile1 and should not be implemented.
0% 1% 20% 50% 70% 99% 100%
Press RED-Key to exit application

Fig. 4. An image pair showing the major stages in VISOR's pipeline. Top row: original inputs and their difference; middle row: after the alignment phase; bottom row: after the max-pooling step.

VISOR evaluates the disparity against the threshold, T_f, to reach a verdict as *pass* or as *fail*. A large threshold value would improve P_{FP} (by turning more FP cases to TN), but hurt P_{TP} (by turning more TP cases to FN). A small threshold value would yield high P_{TP}, but low P_{FP}. We aim at obtaining parameter settings that balances this trade-off, where $P_{TP}/P_{FP} \approx 1$.

4.2.1. Parameter settings

Recall from Section 3.9 that VISOR has five parameters. We do a grid search to set the T_{lum}, T_{noise}, B, and Q parameters. Our grid search involved the following sampled parameter values:

- T_{lum}: 0, 2, 4, 6, 8, 10
- T_{noise}: 10, 20, 30, 40, 50, 60
- B: 8 × 8, 16 × 16, 24 × 24, 32 × 32, 40 × 40
- Q: 10, 20, 30, 40, 50, 60

Table 3
Results obtained with 10-fold cross validation.

Fold	Best threshold	Validation			
#	value (T_f)	Accuracy	P_{TP}	P_{FP}	P_{TP}/P_{FP}
1	42	91.35%	100.00%	91.05%	1.09
2	43	92.11%	88.89%	92.11%	0.97
3	42	91.11%	100.00%	91.80%	1.09
4	43	92.48%	90.00%	92.58%	0.97
5	41	91.73%	100.00%	91.41%	1.09
6	44	92.86%	80.00%	93.36%	0.86
7	43	93.23%	90.00%	93.36%	0.97
8	43	92.11%	90.00%	92.19%	0.98
9	43	92.08%	88.89%	92.19%	0.97
10	42	92.08%	100.00%	91.80%	1.09
	Average	92.11%	92.78%	92.19%	1.01

We evaluated all the combinations of these sampled values. Hence, in total, $6 \times 6 \times 5 \times 6 = 1080$ combinations are evaluated. The combination that gave the best results for our setup is (T_{lum}: 6, T_{noise}: 40, B: 24 × 24, Q: 40).

We set the fifth parameter, T_f, via a training step before the use of VISOR for actual testing. This requires a one-time effort before regression testing of a new system. The parameter fine-tuning by grid search has to be repeated for each different application/system. The data set training step should also be repeated if the user interface and/or corresponding reference images change. Repeating this step for each and every possible parameter setting is also required after a major system change such as changing of a camera, angle of the camera and illumination of the environment. Normally, a test environment is fixed and this step is required only once during the whole lifecycle of the test oracle system.

To evaluate the impact of training for setting the T_f parameter and the data set on accuracy, we applied 10-fold cross validation. That is, we partitioned our data set into 10 randomly-selected, equally-sized, disjoint segments. Then, we trained and applied VISOR for tests 10 times. Each time, we used a different combination of 9 disjoint segments for training, and used the remaining disjoint segment for testing. We measured the accuracy, P_{TP}, and P_{FP} for each test.

4.3. Results and discussion

In this section, we first introduce the results. Then, we elaborate on these results to answer each of the three research questions.

The overall results are listed in Table 3. Hereby, the first column denotes which data segment is used for testing as part of 10-fold cross validation. Recall that each segment corresponds to a randomly selected, disjoint subset of the data set that contains 10% of the experiment objects. The union of the 9 other segments is used for training. The second column lists the best threshold values that are learned after the training step. The third column lists the accuracy of VISOR. The measured P_{TP} and P_{FP} values, and their ratio are listed in the fourth, fifth, and sixth columns, respectively. In each training step, the "best" threshold value is the value that leads the P_{TP}/P_{FP} ratio closest to 1 in the training set.

4.3.1. Accuracy

We can see in Table 3 that the learned threshold values (second column) do not show a high degree of fluctuation; they are in a narrow range, indicating high consistency of the training step under changing segments of the data set that are used for training. This means that our threshold parameter is not dependent on the data partition selected for training our system. This is also the case for the measured accuracy, P_{TP}, and P_{FP} values, which are depicted on a box-plot in Fig. 5. Despite the fact that the y-axis of the plot is narrowed down to the range 75% – 100%, we can see negligible variance. This is especially the case for the accuracy and the P_{FP} values. The measured P_{TP} values have relatively larger variance, especially caused by the 6th test, where P_{TP} is measured to be 80%. This might be caused by the size of the data set. Recall that we had only 96 image pairs in the TP category, whereas we had 2563 pairs in the FP category. As a result, the training step can utilize only ~86 image pairs (90% of the pairs) from the TP category each time. We believe that the measured P_{TP} values would be more consistent if we had more samples in the TP category for training.

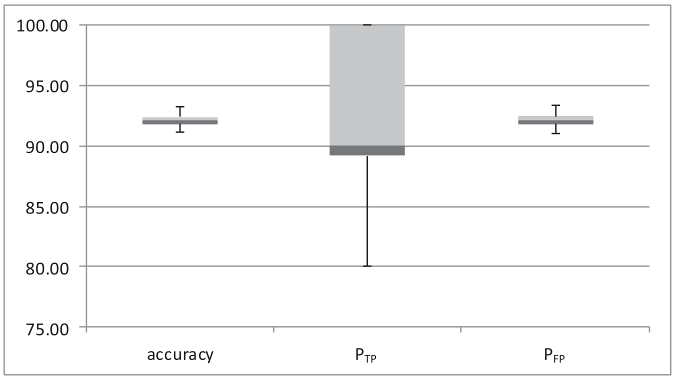


Fig. 5. The box plot that shows the distribution of the accuracy, P_{TP}, and P_{FP} values obtained with 10-fold cross validation.

4.3.2. Speed

In order to evaluate the image processing speed and the training overhead, we measured the time it takes to process one image pair, the time it takes to learn the threshold value T_f, and the time it takes to perform grid search to set the remaining four parameters. At the beginning of our study, we were informed by Vestel test engineers that it takes around 5 s to manually compare two images and reach a decision.

For evaluating speed, we measured the image processing throughput on a laptop computer that has 16GB of memory and a 2.8 GHz quad-core Intel i7 CPU with 2 hyper-threads on each core. VISOR processes 60 image pairs per second, excluding file I/O, when executed in single-thread mode. The throughput is 195 image pair comparisons per second when using 8 threads (again excluding file I/O). In other words, it takes about 45 s to process all the cases in our data set in single thread mode, and about 14 s in multi-thread mode. As a comparison, the perceptual image differencing tool, pdiff (Yee et al., 2001), which was previously used as a test oracle (Mahajan and Halfond, 2015), takes 9.3 s on the average *per* image pair comparison.

We measured the time it takes to perform a data set training session. This is also the time spent in a single step of our grid search for fine-tuning system parameters. Overall, a full fine-tuning of our system takes approximately 4 h. We believe that this time cost is acceptable considering that (1) this level of fine-tuning is rarely necessary, and (2) if needed, it can be carried out overnight as part of a nightly-build system. Employees involved in the study also affirmed that this duration is acceptable and the need for re-calibration of the tool is seldom.

4.4. Threats to validity

Our evaluation is subject to external validity threats (Wohlin et al., 2012) since it is based on a single case study. More case studies can be conducted in different contexts to generalize the results. The type of image effects that take place in other applications might be different than those addressed by VISOR. Hence, some of the steps/filters in the current pipeline of VISOR can be

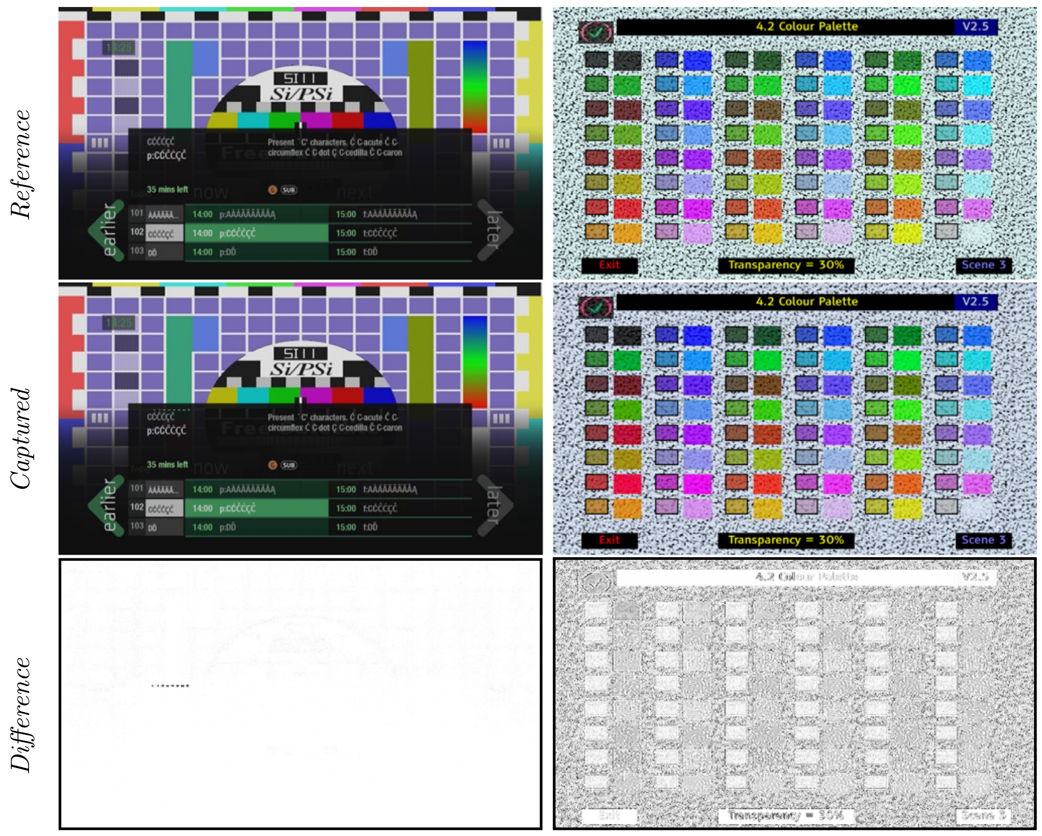


Fig. 6. Sample "tough" cases from the TP data set missed by VISOR.

skipped or new filters might have to be added to handle image effects other than scaling and translation. The level of detail in GUI screens may also be different from DTV screens. As a result, parameters of VISOR might have to be fine-tuned for other applications.

Internal validity threats are either associated with participants or measurements. Our study does not involve human subjects. Internal threats imposed by measurements are mitigated by using real test cases that are applied for a product being subjected to regular regression tests in the industry. Our work did not involve any change of the data set throughout the measurements. We directly used the test cases that are being used in production as is. We also kept the system unchanged throughout the case study.

Conclusion and construct validity threats are mitigated by performing 10-fold cross validation on our data set.

4.5. Limitations of the approach

Results show that the ratio of FP cases that became TN is high (above 90% in general) when we use VISOR. This is due to the fact that the majority of the FP cases are caused by scaling, shifting and/or anti-aliasing differences. Sample image pairs that are subject to these issues are provided in Fig. 1. These image pairs are not evaluated as failures by VISOR and as such they do not lead to FP verdicts anymore.

The ratio of TP cases that are retained as TP is also high (above 80% in general). However, some of these cases became FN when we used VISOR. In Fig. 6, we give two image pairs taken from the TP data set. These are "tough" cases in which VISOR could not detect the error. The first row contains an accent character rendering problem. This causes a very small difference between the reference and captured images. VISOR is designed to tolerate scaling and translation differences between images. Thus, the difference triggered by the character rendering issue erodes in VISOR's pipeline. The second row is a test that checks color saturation and transparency. VISOR is not designed to look for such changes. Hence, this case goes undetected and becomes a false negative as well.

There are also test cases that remain in the FP category. VISOR reported an error for each of these test cases, where in fact, an error does not exist according to the test engineer. Fig. 2(a) shows one of these cases. Hereby, the GUI theme is different between the reference image and the captured image. Such cases are considered out-of-scope for our work. Likewise, there exist test cases in the FP category due to color and transparency differences. We consider the handling of such differences as future work.

5. Related work

Testing of graphical user interfaces (GUI) has been studied for more than two decades (Banerjee et al., 2013). An analysis of the existing work (Banerjee et al., 2013) reveals that only a few studies focus on testing GUIs of embedded devices such as mobile phones (Kervinen et al., 2006). These studies are mainly concerned with the modeling and verification of functional behavior rather than GUI appearance. In fact, the majority of the GUI testing approaches aim at testing the behavior of the system based on captured GUI elements and related event sequences at runtime (Banerjee et al., 2013).

Test oracle automation based on image comparisons was mentioned as an alternative approach (Takahashi, 2001); however, it did not gain attention until recently (Delamaro et al., 2013; Mahajan and Halfond, 2014, 2015). We adopted this approach due to the constraints imposed by the working context for testing consumer electronics, in particular Digital TVs. In this context, the tester does not have any access to the internal events during the execution of these systems. The GUI components (e.g. buttons, labels) or a document object model (e.g. as in HTML) are not available, either. The visual output that is observed on the screen is validated in a black-box fashion. Hence, we employ image processing techniques to automate test oracles by evaluating snapshots of the GUI only. On the other hand, existing visual test automation tools (Alégroth et al., 2015; Leotta et al., 2016) run on the same machine as the system under test and they have access to the GUI components of the system.

Test oracle automation has been extensively studied for testing Web applications, especially for detecting cross-browser issues (Choudhary et al., 2013); however, the employed techniques mainly involve the analysis of the HTML code (Sprenkle et al., 2007). An image comparison technique was proposed by Selay et al. for detecting layout failures in Web applications (Selay et al., 2014). The technique is domain-specific as it assumes the existence of contiguous regions that are subject to failure. This assumption is based on the observed failure patterns in browser layout issues. The study evaluates the effectiveness of adaptive random testing for selecting regions to be compared in image pairs. The mean time to detect a failure is decreased by comparing these regions only, rather than comparing all the pixels. Any difference in the selected regions is interpreted as a failure. Scaling and translation effects are not considered. In our problem context, however, we consider two images to be equivalent even when they differ, if the differences are caused by scaling and translation effects.

Previously, pixel-to-pixel comparison was used by Mahajan and Halfond to detect HTML presentation failures (Mahajan and Halfond, 2014). This technique relies on a very strict comparison and as such it is fragile with respect to scaling, shifting, and color saturation issues. In our context, these issues come up often. In a successive study, Mahajan and Halfond employed *perceptual image differencing* (Yee et al., 2001) to compare images by taking spatial, luminance sensitivity into account (Mahajan and Halfond, 2015). The amount of sensitivity is specified as threshold parameters. They used an external tool, *pdiff*,[5] for performing image comparison. This tool was originally introduced for assessing the quality of rendering algorithms. In our work, we implemented a complementary set of techniques for addressing the oracle automation problem in particular. Rather than utilizing an external tool for this purpose, we implemented a series of image transformations and image comparison as part of VISOR. In addition, VISOR runs substantially faster than *pdiff* (see Section 4).

Image processing and visual testing techniques have also been employed in automative industry for test oracle automation by Amalfitano et al. (2014). This is performed by taking snapshots of the automobile driver's interactive display at specific times during test execution. Amalfitano et al. focus on the Model-in-the-Loop step during software development. Therefore, their test execution takes place in a simulation environment, where captured images are not subject to scaling and translation variations. We, on the other hand, focus on black-box testing of the actual products. In their work, each image is verified with respect to a specification. This specification defines the layout of the display and the information expected to be provided at defined areas of this layout. The expected information is extracted from a captured image with specialized techniques such as pixel-to-pixel comparison for icons, optical character recognition for textual parts of the display, or custom visual feature extraction for more complex display items, such as the level of a gauge. In contrast, VISOR requires a reference image and possibly a mask image for verification rather than a layout description. VISOR is designed to explicitly address scaling and translation variations in captured images, which do not take place in simulation environments.

[5] http://pdiff.sourceforge.net.

Content-based image retrieval techniques have been previously used for test oracle automation (Delamaro et al., 2013; Oliveira et al., 2014). These techniques are used for measuring the similarity of images with respect to a reference image. The similarity measurement is defined based on a set of features extracted from the images. These features are mainly concerned with the color, texture, and shape of objects taking part in the images. The approach has been applied for desktop applications and Web applications. The consumer electronics domain introduces additional challenges that we previously mentioned. Our approach involves several image transformation steps followed by a comparison for detecting differences rather than relying on a similarity measure only.

Sub-image searching was used in a visual testing tool called Sikuli (Chang et al., 2010), where test scripts and assertions can be specified via a set of keywords and images of GUI elements. These images are searched within a Web page, and assertions can lead to failure based on their (non-)existence.

Based on a classification provided in a recent survey (Barr et al., 2015) on test oracles, our approach uses so-called *specified test oracles*. In our case, we use a visual specification, involving a reference image and optionally a mask image for specifying regions within the reference image that should be included or excluded for comparison. Likewise, according to a previously made classification regarding test oracles for GUI (Xie and Memon, 2007), our approach can be considered as the adoption of *visual assertions*. Hereby, snapshots from the interface are recorded for known correct executions of the system first. Then, the tester visually specifies parts of this interface that should hold for all executions.

6. Conclusion

We introduced a test oracle automation approach that employs image processing techniques. Our approach is applicable to any system that produces visual output. We do not assume the existence of any access to the internals of the system. We aim at a context where a reference image is compared with a visual output that is possibly captured with an external camera. Hence, the output might be subject to several distortions due to scaling, shifting, and light reflections. Such differences result in too many false positives when an exact image matching approach is used. To this end, we implemented a tool, named VISOR, that employs an image processing pipeline to compare images in the presence of scaling and translation differences. Our tool is also designed to be very fast. We performed an industrial case study for automated regression testing of Digital TVs. We collected thousands of real-world test cases to form our data set. We obtained promising results indicating that our approach can significantly reduce false positives while causing a negligible decrease in the number of true positives. The evaluation of all the test cases finishes within seconds on a laptop computer. The full fine-tuning of the system takes 4 h on the same computer. This involves training based on a data set of thousands of images and an exhaustive search of all the combinations of sampled parameter settings.

Our future work aims at addressing the limitations listed in Section 4.5. In addition, application of the approach can be evaluated for different types of systems. VISOR is generically applicable for any type of system since it is based on images regarding the outer look of the tested system. However, we should note that we are just focusing on the test oracle automation. A dedicated test harness might have to be built for the tested device to be able to drive the test execution and capture screenshots at predefined points of execution. VISOR just compares the captured images with respect to the reference pictures assuming that they are supposed to be similar.

VISOR can also be coupled with other (complementary) tools that can run in sequence or parallel. On one hand, this would allow joining the strengths of these tools to further improve the accuracy. On the other hand, this would prolong the test execution and analysis process. In this work, our goal was to provide a time-efficient tool that is also accurate at the same time.

Acknowledgements

We thank test engineers at Vestel Electronics, in particular İrfan Can and Ömer Faruk Erdil, for supporting our case study.

References

Alégroth, E., Feldt, R., Ryrholm, L., 2015. Visual GUI testing in practice: challenges, problems and limitations. Empir. Softw. Eng. 20 (3), 694–744.

Amalfitano, D., Fasolino, A., Scala, S., Tramontana, P., 2014. Towards automatic model-in-the-loop testing of electronic vehicle information centers. In: Proceedings of the 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering, pp. 9–12.

Banerjee, I., Nguyen, B., Garousi, V., Memon, A., 2013. Graphical user interface (gui) testing: systematic mapping and repository. Inf. Softw. Technol. 55 (10), 1679–1694.

Barr, E., Harman, M., McMinn, P., Shahbaz, M., Yoo, S., 2015. The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. 41 (5), 507–525.

Beizer, B., 1990. Software Testing Techniques, second Ed. Van Nostrand Reinhold Co., New York, NY, USA.

Benezeth, Y., Jodoin, P.-M., Emile, B., Laurent, H., Rosenberger, C., 2010. Comparative study of background subtraction algorithms. J. Electron. Imaging 19 (3). 033003-033003-12

Berner, S., Weber, R., Keller, R.K., 2005. Observations and lessons learned from automated testing. In: Proceedings of the 27th International Conference on Software Engineering, pp. 571–579.

Boureau, Y.-L., Ponce, J., LeCun, Y., 2010. A theoretical analysis of feature pooling in visual recognition. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp. 111–118.

Chang, T., Yeh, T., Miller, R., 2010. GUI testing using computer vision. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1535–1544.

Choudhary, S., Prasad, M., Orso, A., 2013. X-PERT: Accurate identification of cross-browser issues in web applications. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 702–711.

Delamaro, M., dos Lourdes, F., de Oliveira, R.A.P., 2013. Using concepts of content-based image retrieval to implement graphical testing oracles, software testing. Verif. Reliab. 23 (3), 171–198.

Eaton, C., Memon, A., 2007. An empirical approach to evaluating web application compliance across diverse client platform configurations. Int. J. Web Eng. Technol. 3 (3), 227–253.

Fadnavis, S., 2014. Image interpolation techniques in digital image processing: an overview. J. Eng. Res. Appl. 4, 70–73.

Garousi, V., Mntyl, M., 2016. When and what to automate in software testing? a multi-vocal literature review. Inf. Softw. Technol. 76, 92–117.

Gebizli, C.S., Metin, D., Sozer, H., 2015. Combining model-based and risk-based testing for effective test case generation. In: Proceedings of the 9th Workshop on Testing: Academic and Industrial Conference - Practice and Research Techniques, pp. 1–4.

Gebizli, C.S., Sozer, H., 2014. Improving models for model-based testing based on exploratory testing. In: Proceedings of the 6th IEEE Workshop on Software Test Automation, pp. 656–661.

Gebizli, C.S., Sozer, H., Ercan, A., 2016. Successive refinement of models for model-based testing to increase system test effectiveness. In: Proceedings of the 10th Workshop on Testing: Academic and Industrial Conference - Practice and Research Techniques, pp. 263–268.

Giusti, A., Cireşan, D. C., Masci, J., Gambardella, L. M., Schmidhuber, J., Fast image scanning with deep max-pooling convolutional neural networks. arXiv: 1302.1700.

Gonzalez, R.C., Woods, R.E., 2008. Digital Image Processing, third ed. Prentice Hall.

Howden, W., 1978. Theoretical and empirical studies of program testing. IEEE Trans. Softw. Eng. 4 (4), 293–298.

Kervinen, A., Maunumaa, M., Pääkkönen, T., Katara, M., 2006. Model-based testing through a gui. In: Grieskamp, W., Weise, C. (Eds.), Formal Approaches to Software Testing: 5th International Workshop, 2005, Revised Selected Papers. Springer Berlin Heidelberg, pp. 16–31.

Lee, D.-S., 2005. Effective gaussian mixture learning for video background subtraction. IEEE Trans. Pattern Anal. Mach. Intell. 27 (5), 827–832.

Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2016. Approaches and tools for automated end-to-end web testing, vol. 101 of advances in computers. Elsevier 193–237.

Lin, Y.D., Rojas, J.F., Chu, E.T.H., Lai, Y.C., 2014. On the accuracy, efficiency, and reusability of automated test oracles for android devices. IEEE Trans. Softw. Eng. 40 (10), 957–970.

Mahajan, S., Halfond, W., 2014. Finding HTML presentation failures using image comparison techniques. In: ACM/IEEE International Conference on Automated Software Engineering, pp. 91–96.

Mahajan, S., Halfond, W., 2015. Detection and localization of html presentation failures using computer vision-based techniques. In: Proceedings of the 8th International Conference on Software Testing, Verification and Validation, pp. 1–10.

Memon, A., Pollack, M., Soffa, M., 2000. Automated test oracles for GUIs. SIGSOFT Softw. Eng. Notes 25 (6), 30–39.

Meyer, B., 1988. Eiffel: a language and environment for software engineering. J. Syst. Softw. 8 (3), 199–246.

Motwani, M.C., Gadiya, M.C., Motwani, R.C., Harris, F.C., 2004. Survey of image denoising techniques. In: Proceedings of GSPX, pp. 27–30.

Myers, G., Badgett, T., Sandler, C., 2012. The Art of Software Testing, third ed. John Wiley and Sons Inc., Hoboken, NJ, USA.

Nagi, J., Ducatelle, F., Caro, G.A.D., Cireşan, D., Meier, U., Giusti, A., Nagi, F., Schmidhuber, J., Gambardella, L.M., 2011. Max-pooling convolutional neural networks for vision-based hand gesture recognition, in: Signal and image processing applications (ICSIPA), 2011. In: IEEE International Conference on, IEEE, pp. 342–347.

Oliveira, R., Memon, A., Gil, V., Nunes, F., Delamaro, M., 2014. An extensible framework to implement test oracle for non-testable programs. In: Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering, pp. 199–204.

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

Scherer, D., Müller, A., Behnke, S., 2010. Evaluation of pooling operations in convolutional architectures for object recognition. In: International Conference on Artificial Neural Networks. Springer, pp. 92–101.

Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cort, A., 2016. A survey on metamorphic testing. IEEE Trans. Softw. Eng. 42 (9), 805–824.

Selay, E., Zhou, Z.Q., Zou, J., 2014. Adaptive random testing for image comparison in regression web testing. In: Proceedings of the International Conference on Digital Image Computing: Techniques and Applications (DICTA), pp. 1–7.

Sivaraman, G., Csar, P., Vuorimaa, P., 2001. System software for digital television applications. In: IEEE International Conference on Multimedia and Expo, pp. 784–787.

Sprenkle, S., Pollock, L., Esquivel, H., Hazelwood, B., Ecott, S., 2007. Automated oracle comparators for testingweb applications. In: Proceedings of the 18th IEEE International Symposium on Software Reliability, pp. 117–126.

Takahashi, J., 2001. An automated oracle for verifying GUI objects. SIGSOFT Softw. Eng. Notes 26 (4), 83–88.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A., 2012. Experimentation in Software Engineering. Springer-Verlag, Berlin, Heidelberg.

Wold, S., Esbensen, K., Geladi, P., 1987. Principal component analysis. Chemometrics Intell. Lab. Syst. 2 (1–3), 37–52.

Xie, Q., Memon, A., 2007. Designing and comparing automated test oracles for gui-based software applications. ACM Trans. Softw. Eng. Method. 16 (1), 1–36. Article No. 4

Yee, H., 2004. Perceptual metric for production testing. J. Graphics Tools 9 (4), 33–40.

Yee, H., Pattanaik, S., Greenberg, D., 2001. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. ACM Trans. Graphics 20 (1), 39–65.

Zhou, Z.Q., Xiang, S., Chen, T.Y., 2016. Metamorphic testing for software quality assessment: a study of search engines. IEEE Trans. Softw. Eng. 42 (3), 264–284.