An Extensible Framework to Implement Test Oracles for *Non-Testable Programs*

Rafael A. P. Oliveira*†, Atif M. Memon†, Victor N. Gil‡, Fátima L. S. Nunes‡ and Márcio Delamaro*
* Dept. of Computer Systems, University of São Paulo – (ICMC/USP)
São Carlos, SP, Brazil
Email: rpaes@icmc.usp.br delamaro@icmc.usp.br
† Department of Computer Science, University of Maryland – UMD
College Park, MD, USA
Email: atif@cs.umd.edu
‡ Dept. of Computer Systems, University of São Paulo – (EACH/USP)
São Paulo, SP, Brasil
Email: victor.gil@usp.br fatima.nunes@usp.br

***Abstract*—Test oracles evaluate the execution of SUTs (*Systems Under Test*) supporting testers to decide about correct outputs and behaviors. "Non-testable" systems are cases in which the testers must spend extraordinary efforts to judge SUT's outputs. Currently, some contemporary non-testable programs are represented by systems with complex outputs such as GUIs (*Graphical User Interface*), Web applications, and *Text-to-speech* (TTS) systems. Currently, there is a lack of knowledge associated with automated test oracles and SUTs with complex outputs. Extensible testing frameworks are necessary to provide the reuse of components and the sharing of knowledge in this field. This paper presents an extensible framework to support the development of test oracles for non-testable programs with complex outputs. In addition, we present an alternative to reuse software engineering components through plug-ins-based frameworks. The framework adapts CBIR (*Content-Based Image Retrieval*) concepts to enable testers to specify test oracles. The framework matches concepts of signal feature extraction, similarity functions, and object comparisons to obtain a Java program that compares two objects, responding how similar they are, according to a threshold. We performed proofs of concept using two empirical studies and the results showed our framework is useful to alleviate human-oracle efforts supporting human decisions. In addition, the plug-ins-based framework we present is a contribution toward a reusing of components on test oracles for systems with complex outputs.**

***Keywords*—test oracle; software testing; knowledge engineering.**

I. Introduction

Test oracles are instruments to examine particular executions of an SUT (*Systems Under Test*), supporting software testing designers to decide about correct outputs [1]. In testing environments, test oracles must deal with SUT's outputs or behaviors after a determined input. Test oracles may assume several forms: programs, functions, sets of data, tables of values or even the tester's knowledge about the SUT [2]. Regarding complex testing scenarios, including strategies for test case generation and coverage reports, a test oracle is a requisite to insure the testing productiveness. Automated test oracles can increase the test productivity and decrease testing costs [3].

"Non-testable" programs are SUTs in which an oracle does not exist or the tester must expend some exceptional amount of time to determine whether the current output is correct [4]. Despite being a well-known concept, non-testable programs and systems hard to test are still common. For instance, software systems whose outputs represent complex data for test automation. In this context, some examples of contemporary non-testable programs are: (1) systems with complex GUIs (*Graphical User Interface*) whose visual information is reflected in different screen resolution, sizes, and orientations; (2) *Text-To-Speech* (TTS) systems that convert written text in audio files; (3) some Web applications whose result quality depends on the browsers; (4) *Virtual Reality* (VR) environments, and others. Systems with complex outputs are becoming standard and the software industry is developing more attractive and exquisite applications. In addition, contemporary software has a vast array of technologies including the hardware platforms, *Operating Systems* (OS), and programming languages to support complex scenarios.

In the context of software testing, complex outputs make oracle verification difficult. Existent studies on test oracles for systems with complex outputs use manual (human-oracle), semi-automated or ad-hoc techniques [5]. Due to their complexity, test oracle analysis of complex output commonly requires sensory and perceptual aspects of a human being: (1) vision and (2) hearing. For instance, vision is required for checking orientations and visibility of components in GUIs, Web applications, and VR environments. Hearing is needed for checking naturalness, intonations, and pauses in TTS systems, in which audio files are the output. However, manual approaches are unproductive and inappropriate to support the large demand for the software. A vast array of technologies including mobile devices, hardware platforms, OS, and programming languages are in favor of the complex output's systems.

To supply the test industry with supporting strategies to systematize and automate testing techniques for systems with complex outputs, contributions from *Software Engineering* (SE) and *Knowledge Engineering* (KE) are necessary. Flexible and reusable approaches may represent a way to alleviate efforts on testing those systems. Every advancement or increase in knowledge of testing systems with complex outputs have to be adequately reported to enable its reuse.

In this context, knowledge-based approaches enhance the effectiveness of components and procedures reuse. Regarding testing approaches for contemporary non-testable systems, the knowledge has to be shared in order to follow satisfactorily new technologies.

In this paper we present an extensible open-source framework to support the development of flexible test oracles for systems with complex outputs. O-FIm/CO (*Oracle For Images and Complex Outputs*)[1] adapts CBIR (*Content-Based Image Retrieval*) concepts to allow tester defining flexible and adaptable test oracles, as well as reusing plug-ins' implementations. To provide an environment of automated test oracles, the framework matches concepts of signal feature extraction, similarity functions, and object comparisons. CBIR is the application of computer vision and *Image Processing* (IP) techniques to help in organizing digital images using their visual content [6]. Regarding an image database, according to one or more extracted features, CBIR systems locate images that are similar to a query image. Through its reusable plug-ins, O-FIm/CO allows the generalization of these concepts for complex outputs such as GUI screenshots and audio files. The O-FIm/CO flexibility allows testers to obtain a Java program that compares two objects, responding if they are similar or not, according to a threshold.

Besides some reusable testing plug-ins by themselves, these paper's contributions rely on the step-by-step feasible strategy to implement flexible test oracles for systems hard to test. We evaluate our technique using two practical examples of real-world systems with complex outputs: one for graphical outputs (GUIs), and one for audio outputs (TTS). We developed test oracles that use features from outputs as sources of information for comparison between model outputs and current SUT's outputs. In this context, the goals of this paper are: (1) Presenting an extensible framework for testing systems with complex output, using reusable plug-ins; (2) Describing a feasible way to write flexible test oracles for complex output domains; and (3) Presenting two practical examples of the framework's practical effectiveness on evaluating complex outputs.

II. O-FIm/CO

Besides technical and structural details of O-FIm/CO, this section presents overall concepts of the "oracle problem" and CBIR.

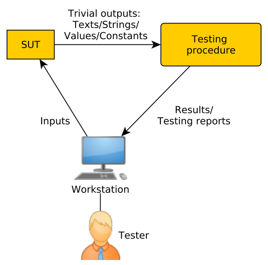
A. *The Oracle Problem*

When it is impossible or too difficult to decide about the correctness of test outputs, the result is a scenario called the oracle problem [4], [7]. Further, the oracle problem is the absence of an accurate test oracle or cases in which it is too expensive to apply the oracle. The oracle problem may occur depending on the SUT. In such cases, the test oracle is not able to support the right decisions. Due to the oracle problem, the tester often classifies the design of the test oracle as a complex and cognitive activity rather than being a routine activity [1]. Approaches to alleviate the oracle problem, in general, must deal with two problems: "false positives" (when a test result incorrectly rejects a true null hypothesis) and "false negatives" (the failure to reject a false null hypothesis).

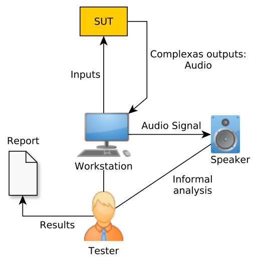
[1]see: *http://ccsl.icmc.usp.br/pt-br/projects/o-fim-oracle-images*

Among several contemporary examples of testing scenarios associated with the oracle problem, our scope in this paper are SUT with graphical (GUIs, processed images, maps, routes) or audio (voice synthesis) outputs. Regarding testing strategies for these systems, often the tester has to play the role of the oracle, making the test activity unproductive and error-prone. However, the complex output domains lead the tester to decide about the correctness of the SUT manually.

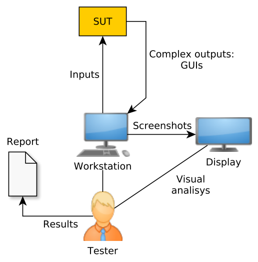
Figure 1 presents different testing scenarios relying on the complexity of the SUT's outputs. In the Figure 1b, the SUT represents a TTS system, in which the tester (human-oracle) has to listen to the result that represents an understandable speech representing a written text. Similarly, Figure 1c represents a testing work-flow for an SUT with a graphical output, where the tester has to check several graphical aspects. For instance, regarding GUIs, the visual aspects must work properly independently of platform, screen resolutions, screen orientation, color systems, monitor settings, and *Look and Feels* (L&Fs). In both cases (Fig. 1b and 1c), the complexity of the SUT's output are considered. On the other hand, Figure 1a presents a testing scenario where the SUT has trivial outputs and automated testing procedures can be implemented without extraordinary efforts.



(a) Trivial-output Scenario.



(b) Audio Scenario.



(c) Graphical Scenario.

Fig. 1: Testing Scenarios Depending on the SUT's Output.

B. *Content-Based Image Retrieval*

CBIR is any technology which helps organize digital image files using its visual content [6]. CBIR systems allow retrieving a finite set of images similar to a reference image. The similarity criteria are obtained by features extraction from the image related to shape, color, and texture. Feature extraction is the method used to obtain some information or particular data from the image. A set of extracted features composes a feature vector that will be considered in its retrieval. The feature vector is not sufficient to determine the result. It is necessary to measure how similar two feature vectors are by using some sort of distance measure called a similarity function. Then, image comparisons are performed using values of features extracted and a similarity function. This process results in numeric values that represent the distance between the images.

C. O-FIm/CO: Oracle For Images and Complex Outputs

O-FIm/CO enables testers to apply test oracle strategies in SUTs with complex outputs. O-FIm/CO extends concepts presented in previous works [8] regarding specific domains, CAD (*Computer-Aided Diagnosis*), and Web applications as SUTs with graphical outputs. In the scope of this paper, O-FIm/CO represents an alternative of testing verification for cases in which the complex outputs limit testers to applying traditional oracle strategies.

A precondition in using O-FIm/CO is the need for a reliable and representative output from which it is possible to establish a baseline for comparing objects under testing. For example, when it is desired to evaluate the appearance of a GUI in a particular L&F, it requires an environment from which it is possible to establish a standard image (screenshot) for comparison during the test. This standard screenshot may represent, for example, a GUI in a default screen resolution and orientation. On the other hand, regarding TTS systems, two empirical strategies are possible: (1) use two different systems and compare their outputs; and (2) compare the output produced by a TTS system with the audio of a person who reproduces the text input.

Figure 2 presents, in detail, the generic structure of O-FIm/CO. Regarding the structure presented, O-FIm/CO has an organization that matches different modules and resources in a common environment. These modules play fundamental rules to provide an oracle setting. These modules are: (1) "core"; (2) "parser"; (3) "wizard"; (4) "plug-ins"; (5) "API" (*Application Program Interface*); and (6) "oracle application".

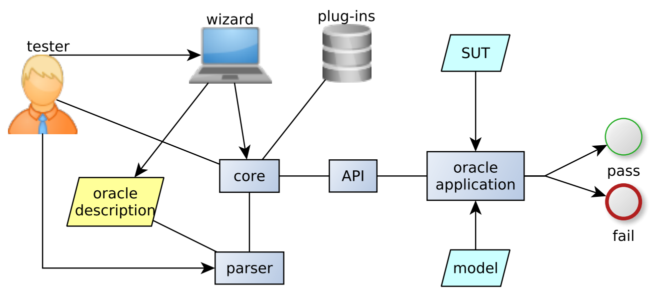


Fig. 2: O-FIm/CO General Structure.

"Plug-ins" may represent feature extractors for complex outputs or similarity functions. In the O-FIm/CO context, plug-ins are the main contribution from testers and they represent a reusable package of Java classes. In these plug-ins, testers must implement algorithms able to quantify features (feature extractors) or to figure out the difference between objects in accordance with their features (similarity functions). The framework is extensible due to its acceptance of plug-ins, providing the apportion of knowledge. In order to associate these plug-ins and O-FIm/CO, testers must implement some specific Java interfaces. Then, the requirement is that when developing an extractor, the tester creates a Java class that implements one of these interfaces. There are different interfaces for audio outputs, graphical outputs, and similarity functions. After that, "testers" are able to install/uninstall plug-ins in O-FIm/CO in two different ways: (1) using line commands that are interpreted by the O-FIm/CO "core"; (2) using a friendly "wizard". Exploring ClassLoaders, O-FIm/CO loads plug-in packages that are able to act over an SUT (audio or graphical output) extracting relevant information to compose test oracles.

Once the setup described above is made, testers may specify their test oracles using the O-FIm/CO "wizard". To do so, feature extractors, extractor parameters, and similarity functions must be specified. The final specification is saved in a text file that represents an "oracle description". Figure 3 presents an example of an oracle description including a similarity function, two extractors, and their parameters. During a test execution, a "parser" analyzes oracle descriptions and sets up the test. Finally, using an intuitive "API", testers must write an "oracle application" that matches framework, oracle descriptions, SUT's outputs, and models. The oracle application is a Java program that analyzes a set of complex SUT outputs, extracting features specified by testers, returning if they are similar to a model or not, according to a threshold (precision).

```
------------------------------------------------------------
similarity Euclidean
extractor Color {rectangle = [100 100 30 40] sc = 1.33}
extractor Connectedcomponent {rectangle = [0 0 128 64]}
precision = 0.87
------------------------------------------------------------
```

Fig. 3: Generic Instance of an Oracle Description.

III. Proofs of Concept

In this section we present empirical analysis whose goals are determining whether test oracles for SUT with complex outputs can benefit from our technique, enabling the reuse of componentss and the knowledge sharing among researchers. Then, the study is designed to answer the following *Research Questions* (RQ):

RQ1: What is the feasibility of knowledge sharing through reusable plug-ins when automating test oracles for SUTs with different complex outputs?

RQ2: Is this approach (adapting CBIR concepts) flexible enough to support oracle automation for different complex SUTs?

To answer the above questions, we have used real system applications for two critical complex scenarios: (1) graphical outputs and (2) audio outputs. Regarding graphical outputs we use O-FIm/CO to verify test results of a system with GUIs, exploring the framework resources on verifying acceptable behaviors of a TTS system.

A. Complex Scenario 1: Complex GUIs

In the first scenario we use the O-FIm/CO approach to support the automated testing of GUI-based programs. This example of usage aims to evaluate the framework resources on alleviating manual verification of GUI visual requirements. These visual requirements include loading the SUT in different

L&Fs, different screen resolutions, and OSs. Then, our strategy implements an oracle application able to evaluate screenshots exposing GUIs from different aspects and to support decisions about their correctness regarding a model.

1) Subject Application: The subject application empirically evaluated is the text editor jEdit[2]. This application is released as free software with open source code. jEdit is a programmer's text editor written in Java and its main GUI has a huge panel for text editing, a tool-bar, and a side including a package explorer. Figure 4a presents the main jEdit GUI and its components. Figure 4b represents this same GUI under a different L&F. jEdit has a mean of 8000 downloads weekly and its development team is always fixing bugs reported by users.

2) Experimental Setup: In order to verify jEdit GUI appearance under critical situations, we have changed the default parameters of its appearance. Further, this proof of concept was designed to access the effectiveness of our technique in detecting possible failures related to visual aspects of GUIs derived from visual settings by users. Using a 20" monitor, we have exposed jEdit's main GUI to the following situations:

- Execution on six different screen resolutions (800x600, 1024x768, 1152x864, 1280x1024, 1366x768, and 1600x900);
- Execution on three different L&Fs (CDE-Motif, Metal, and Nimbus);;
- Execution on two different OS. (Linux and Microsoft Windows®); and
- Execution on two different GUI sizes (Maximized and Regular).

Regarding all possible combinations of visual aspects, we manually took 72 (36 for each OS) screenshots representing the visual information provided by the jEdit GUI. To perform this analysis, we have regarded the jEdit GUI with a resolution of 1366x768 and the Metal L&F as a model. Then depending on the OS, and the GUI size, different models were considered. We assured in advance that each model contained the appropriate requirements for the proper use of the application.

jEdit may have vision problems that can negatively affect its functions. Depending on the screen resolution, toolbar buttons can disappear and, then, become inaccessible for final users, so this constitutes a sort of visual bug. Indeed, this is a common problem for several GUI-based applications. Then, in this empirical evaluation we seek to automate test oracles for identifying this and similar situations. We have adopted a strategy that considers applying feature extractors in different points of jEdit's GUI. In Figure 4a we highlight the four main spots of interest in jEdit's GUI. Additional visual errors are supposed to be detected by this oracle: distorted components, and inoperable or invisible buttons. Then, based on comparisons with model features, our test oracles judge the test as *fail* when these conditions are identified.

To implement test oracles able to check the situations described above, we have developed five O-FIm/CO plug-ins: (1) Color count, (2) Sobel vertical, (3) Sobel horizontal, (4) Connected component, and (5) *Euclidean Distance* (ED). Along the framework website, all of these plug-ins (including scenario 2) are available as open source code software for possible experiments.

[2]see: *http://www.jedit.org/*

Color count is an extractor that, after pre-processing that converts the SUT screenshot into a gray scale image, counts the number of different gray levels in the region set by the parameter area. The number of different colors is normalized by 256 (number of levels). This extractor is useful in complex toolbars and informative panels to quantify their diversity of color.

Sobel vertical and Sobel horizontal are two edge detection algorithms for highlighting image patterns. After binarizing the GUI screenshot and applying the Sobel operator, this extractor counts the number of vertical or horizontal edge pixels. Then, the algorithm normalizes the number of pixels by the perimeter of the GUI. This plug-in quantifies visual patterns in some GUI components, such as combo boxes, toolbars, and information panels.

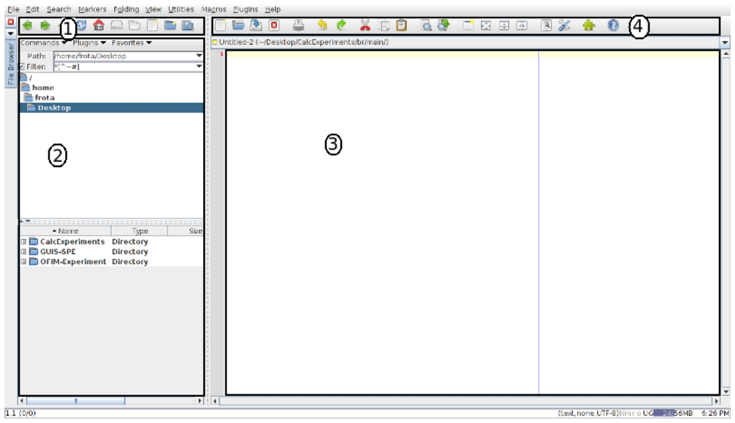
Connected component is able to count the number of different components in a determined region of the GUI. Using a gray scale image, this extractor identifies components by counting the number of similar pixels using an IP technique known as area-point transformation. This number is normalized by image width. This extractor can be used mainly to identify missing components after changing L&Fs and screen resolutions.

Euclidean Distance is a similarity function that corresponds to the function that measures the distance between two vectors of length n. The result 0.0 represents the maximum similarity.

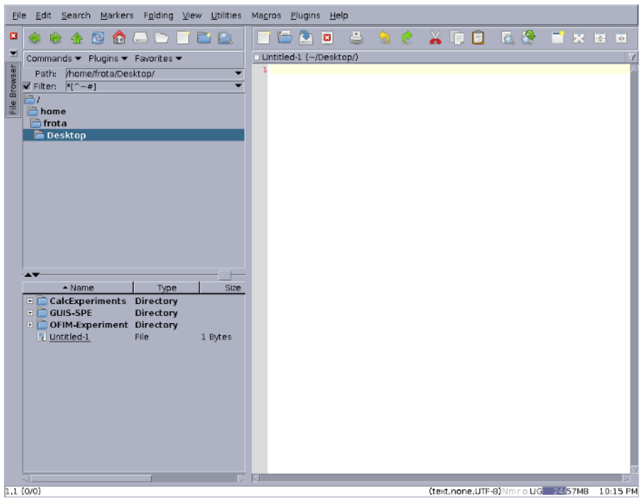
After installing these plug-ins using the framework's wizard, we developed two different oracles. The first oracle (Oracle 1) rule is: to apply color count extractor in Rectangles 1, 3, and 4 of Figure 4a; Sobel vertical and Sobel horizontal are applied in Rectangle 2; Finally, the ED function checks the difference between the vectors (SUT versus Model). The second test oracle (Oracle 2) rule is: to apply the connect components extractor in Rectangles 1 and 4; in addition, the color count extractor acts in Rectangle 3; the ED function is used to obtain the final difference between the SUT and model.

For both oracles, the threshold is set using the distance of the model and the screenshot of the SUT on a 1280x1024-Nimbus GUI. This threshold (precision) is due to a visual analysis that reveals most of the screenshots on resolutions 1280x1024, 1600x900, and 1366x768 always maintaining the necessary functional requirements to be considered as approved ("pass"). In the same line, Nimbus is the L&F that remains the minimal visual conditions of the GUI.

3) Experimental Results: After the test conduction, a manual detailed analysis of false (positive or negatives) results was carried out. Table I presents the false alarms noticed by our automated oracle regarding the whole proof of concept, including oracles, false positives, false negatives, and error rate. Each row represents one oracle in a specific OS under 36 different visual conditions. The error rate represents the sum of false positives and false negatives for each row. Broadly, for the Linux OS, considering the Maximized screenshots, the 1280x1024, 1366x768 e 1600x900 resolutions did not present any visual problem of loss of components or inactivity of buttons. However, the 800x600, 1024x760 e 1152x864 have presented some loss or distortion of components and therefore it was expected that the oracles fail their tests. For regular sizes GUIs, model images were considered in regular sizes and thus, it was expected that none of the screenshots presented problems. Analyzing of the screenshots of the Windows operating system, we note that the only difference to be considered in


File Edit Search Markers Folding View Utilities Macros Plugins Help
Commands Plugins Favorites
Path: /home/frota/Desktop
Filter: *[^~#]
home
frota
Desktop
Name Type Size
CalcExperiments Directory
GUIS-SPE Directory
OFIM-Experiment Directory
Untitled-2 (~/Desktop/CalcExperiments/br/main/)
1
2
3
4

(a) Default jEdit GUI


File Edit Search Markers Folding View Utilities Macros Plugins Help
Commands Plugins Favorites
Path: /home/frota/Desktop/
Filter: *[^~#]
home
frota
Desktop
Name Type Size
CalcExperiments Directory
GUIS-SPE Directory
OFIM-Experiment Directory
Untitled-1 File 1 Bytes
Untitled-1 (~/Desktop/)

(b) jEdit GUI: Motif L&F

Fig. 4: jEdit GUIs.

comparison to previous analyzes is that the Maximized size screenshots with 1152x864 did not present visual problems.

TABLE I: Error Summary for Complex Scenario 1.

# Oracle	OS	False positives	False Negatives	Error rate
1	Linux	3	0	8.3% (3/36)
1	Windows	1	2	8.3% (3/36)
2	Linux	2	0	5.5% (2/36)
2	Windows	1	0	2.8% (1/36)
Total		**7**	**2**	**6.25% (9/144)**

B. Complex Scenario 2: TTS Systems

This second scenario aims to demonstrate how to explore the framework O-FIm/CO to develop automated test oracles for SUTs with complex outputs given in audio format. This scenario's scope is limited to TTS systems. Many embedded systems use TTS applications to read e-mails or social network updates, to read books or headlines for blind people, to read traveling directions, news, weather forecasts, and other written materials. However, the mainstream adoption of TTS is severely limited by its quality. Pronunciation and intonation problems make the speech synthesized highly unnatural. Despite their importance, TTS systems have their quality assessed by manual and unproductive processes. Informal human interpretations and different manual approaches are the most common procedures to evaluate the output of TTS systems [5]. In this empirical evaluation, using audio files in Wave format as the source of information for verifications, we automate test oracles to support testers' decisions about the correctness of TTS's outputs through comparison of two TTS systems.

1) Subject Application: Outputs from two real TTS systems are considered in this empirical analysis: (1) *TTS CPqD* (version 3.3)[3]; and (2) *Google®Translate Text-to-Speech*[4]. CPqD is the major Brazilian provider of telecommunications and solutions. TTS CPqD is a system able to synthesize texts written in Portuguese in speech signals near human speech. Then, Portuguese is the idiom considered in this evaluation.

[3]see: http://www.cpqd.com.br/

[4]see: https://code.google.com/p/java-google-translate-text-to-speech/

2) Experimental Setup: The aim of this analysis is to check two different TTS systems and compare their outputs. Typically, outputs from one TTS have to be regarded as model. TTS CPqD is a commercial application, then we have considered it as our model. Consequently, Google Translate TTS represents our SUT. In this context, in order to generate a data set, 100 Portuguese words were selected. These words were generated by the CPqD system loaded with three different texts from random popular news website. The choice of words did not follow any restriction, and it was obtained following the order of their occurrence. At the end of the process, 100 audio files were generated. Finally, the same words were used as input to our SUT.

We have implemented an automated test oracle exploring two test oracles: (1) Vowel extractor; and (2) Phoneme extractor. "Vowel extractor" is able to analyze an audio signal identifying the presence and the instant in which a Portuguese phoneme (A [/a/, /ɐ/], E [/e/, /ɛ/], I [/i/], O [/o/, /ɔ/], and U [/u/]) is identified. Then, from a file representing a voice signal, the algorithm is able to return useful testing information. This information is specified through an XML (*eXtensible Markup Language*) file. "Phoneme Extractor" analyzes an audio signal, reporting the occurrences of three most common Portuguese phonemes (K [/t/], T [/t/], and M [/m/]). Similar to the vowel extractor, testers are able to set parameters for the phoneme extractor using an XML file. Then, the extractor identifies the presence and the moment of specified phonemes in an audio file.

Both extractors went through a process of parameter calibration to work properly in the context of our TTS system using our data set including more than 100 words. After installing the plug-ins into the framework, we designed a test oracle using the extractors aforementioned to evaluate in order to assess the accuracy of the identification of the phonemes. This experience also aims to estimate the possibility of using oracle mechanisms for TTS systems test.

3) Experimental Results: We manually analyzed the results regarding the oracle acting in the set of data. This analysis aimed to measure the hits achieved in the identification of phonemes of interest. Table II and Table III present a summary of errors and hits of the vowel extractor and the phoneme extractor, respectively. The tables reveal the hits regarding the number of occurrences of the feature extracted. Through a detailed evaluation of the results, we identify the differences between speed and voices of both systems as the main cause of errors.

TABLE II: Summary for the Vowel Feature Extractor.

Vowel		Number of occurrences					
		CPqD system			Google TTS		
		0	**1**	**> 1**	**0**	**1**	**>1**
A	/a/, /ɐ/	71%	85%	57%	60%	68%	64%
E	/e/, /ɛ/	65%	76%	60%	73%	61%	50%
I	/i/	89%	72%	50%	74%	59%	25%
O	/o/, /ɔ/	75%	69%	40%	61%	72%	40%
U	/u/	68%	68%	100%	93%	40%	100%
hits		**74%**	**74%**	**61%**	**72%**	**60%**	**56%**

TABLE III: Summary for the Phoneme Feature Extractor.

Phoneme		Number of occurrences			
		CPqD system		Google TTS	
		0	**1**	**0**	**1**
K	/k/	74%	53%	81%	63%
T	/t/	85%	48%	68%	52%
M	/m/	73%	47%	52%	20%
hits		**77%**	**49%**	**67%**	**45%**

IV. Result Discussions

Regarding the research questions aforementioned (Sec. IV), our results contributed to clarify some points. We consider our approach to check SUT's outputs of distinct non-testable programs feasible. Some efforts from testers are necessary to assimilate the entire technique, however elementary concepts of programming and IP are enough to adopt our strategy and automate test oracles, as well as answer RQ1. Then, we consider O-FIm/CO an extensible framework, which enables testers to use pre-implemented plug-ins, as a potential form of knowledge sharing on automating test oracles for SUTs with complex outputs.

Regarding RQ2, as in most SE studies, we cannot be sure that the subject programs and the empirical analysis we adopted are representative to answer this question completely. Two scenarios are not enough to generalize our results for another complex scenario such as Web applications and RV environments. We need further analyses and more research focused on getting shared knowledge from other studies.

In addition to the pre-defined RQs, through the conduction of this study, we noticed that several research investments are necessary from testers. We consider these non-experimental findings as trade-offs on using O-FIm/CO. Among these trade-offs we highlight: (1) implementing or finding plug-ins; (2) seeking reliable output models; and (3) using the API to implement oracle applications. However, once the testers are done with investments 1 and 2, the technique is useful to alleviate oracle efforts on different contexts. In this context, all these investments address the usage of O-FIm/CO to a concept known as "amortization cost". Generally this concept refers to cases in which the testers spend an amount of time and effort to set a favorable scenario to be explored again and again.

Among the limitations associated with the wide usage of O-FIm/CO, we highlight that the correctness and precision of its oracles actually depend on the quality of the oracle programs (threshold and plug-ins), which are manually developed by domain engineers. Then, testers must to select adequate plug-ins to achieve accurate results. In addition, the approach as whole needs an assumption – some reliable expected output must be given. In practice, testers and developers have to decide, based on system specifications, about reliable sources of information to be considered in testing. This decision evolves important features that have to remain apart of environments variations.

V. Conclusion

This paper proposes an extensible and reusable framework to support test oracle automation for non-testable programs, alleviating human-oracle efforts through reusable testing plug-ins and enabling the sharing of knowledge in this field. Based on two research questions, we conducted empirical analysis on two different complex scenarios: graphical and audio outputs. Our findings showed the feasibility of our approach and the possibility of reuse of plug-ins. In addition, we highlight that complementary experiments are necessary to consider this approach in wider scenarios such as, the usage of the framework to evaluate visual aspects of more than five GUI-based systems. A trade-off analysis reveals that tester has to dedicate extra effort in the very beginning of the testing project to achieve a favorable automated scenario. We believe extensible frameworks set possible solutions to the lack of knowledge associated with automated test oracles and SUT with complex outputs. The framework and all plug-ins developed and explored to collect our results are available as free software along with its documentation. This opens channels for technology transfer and reveals a solid contribution of the research reported in this paper.

Acknowledgment

FAPESP – Grant Numbers 2012/06474-1 and 2013/01775-6. Much of this work was conducted at the University of Maryland at College Park, MD, USA, during an one-year visit by the first author.

References

[1] W. Chan and T. Tse, "Oracles are hardly attain'd, and hardly understood: Confessions of software testing researchers," in *Proceedings of the 13th International Conference on Quality Software (QSIC 2013)*, Boston, USA, 2013, pp. 245–252.

[2] P. Mateo and P. Usaola, "Bacterio oracle: An oracle suggester tool," in *Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE 2013)*, Boston, USA, 2013, pp. 300–305.

[3] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012, pp. 870–880.

[4] E. J. Weyuker, "On Testing Non-Testable Programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, Nov. 1982.

[5] P. Taylor, *Text-to-Speech Synthesis*, 1st ed. Cambridge University Press, 2009.

[6] R. Datta, D. Joshi, J. Li, and J. Z. Wang, "Image Retrieval: Ideas, Influences, and Trends of the New Age," *ACM Computing Surveys*, vol. 40, no. 2, pp. 1–60, Apr. 2008.

[7] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proceedings of the ACM '81 Conference*, ser. ACM '81. New York, NY, USA: ACM, 1981, pp. 254–257.

[8] M. E. Delamaro, F. L. S. Nunes, and R. A. P. Oliveira, "Using concepts of content-based image retrieval to implement graphical testing oracles," *Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 171–198, 2013.