Automatic Testing of Real-Time Graphics Systems

Robert Nagy[1], Gerardo Schneider[2], and Aram Timofeitchik[3]

[1] Dfind Redpatch, Sweden
[2] Department of Computer Science and Engineering,
Chalmers | University of Gothenburg, Sweden
[3] DQ Consulting AB, Sweden
ronag89@gmail.com, gerardo.schneider@gu.se, aram.timofeitchik@dqc.se

Abstract. In this paper we deal with the general topic of verification of real-time graphics systems. In particular we present the Runtime Graphics Verification Framework (*RUGVEF*), where we combine techniques from runtime verification and image analysis to automate testing of graphics systems. We provide a proof of concept in the form of a case study, where *RUGVEF* is evaluated in an industrial setting to verify an on-air graphics playout system used by the Swedish Broadcasting Corporation. We report on experimental results from the evaluation, in particular the discovery of five previously unknown defects.

1 Introduction

Traditional testing techniques are insufficient for obtaining satisfactory code coverage levels when it comes to testing real-time graphical systems. The reason for this is that the visual output is difficult to formally define, as it is both dynamic and abstract, making programmatic verification difficult to perform [6]. Inherent properties of real-time graphics, such as non-determinism and time-based execution, make errors hard to detect and reproduce. Furthermore, dependencies such as hardware, operating systems, drivers and other external run-time software also make the task of testing quite difficult, as witnessed by Id Software during the initial release of their video-game Rage, where the game suffered problems with texture artifacts [11]. Even though the software itself performed correctly, the error still occurred when executed on systems with certain graphic cards and drivers.

A common method for verifying real-time graphics is through ocular inspections of the software's visual output. The correctness is manually checked by comparing the subjectively expected output with the output produced by the system. Some disadvantages with this approach are that it requires extensive working hours, it is repetitive, and it makes regression testing practically inapplicable. Moreover, the subjective definition of correctness makes it possible for some artifacts to be recognized as errors by some testers, but not by others [10]. Furthermore, some errors might not be perceptible in the context of specific tests thereby making ocular inspections even more prone to human-error.

In this paper, we present a conceptual model for automatic testing real-time graphics system, with the aim of increasing the probability of finding defects, and making software verification more efficient and reliable. The proposed solution is formalized as the *Runtime Graphics Verification Framework* (*RUGVEF*), based on techniques from runtime verification and image analysis, defining practices and artifacts needed to increase the automation of testing. We implemented and evaluated the framework by using it in the development setting of *CasparCG*, a real-time graphics system used by the *Swedish Broadcasting Corporation* (*SVT*) for producing most of their on-air graphics. We also present an optimized implementation of the image quality assessment technique *SSIM*, which enables real-time analysis of *Full HD* video produced by *CasparCG*. As a result of the application of *RUGVEF* to *CasparCG* we identified 5 previously unknown defects that were not previously detected with existing testing practices at *SVT*, and 6 out of 16 known defects that were injected back into *CasparCG* could be found. This shows that *RUGVEF* can indeed successfully complement existing verification practices by automating the detection of contextual and temporal errors in graphical systems. Using the framework allows for earlier detection of defects and enables more efficient development through automated regression testing. In addition to this, the framework makes it possible to test the software in combination with its external environment, such as hardware and drivers.

In summary our contributions are: i) The framework *RUGVEF* for automating the testing of real-time graphics systems; ii) The implementation of the framework into a tool, and its application to an industrial case study (*CasparCG*), finding 5 previously unknown defects; iii) An optimized implementation of *SSIM*, an image quality assessment technique not previously applicable to the real-time setting of *CasparCG*.

We start with some background in next section, and we outline our conceptual framework *RUGVEF* in section 3 . We present our case study in section 4, of which we show the results in section 5. Related work is presented in section 6.

2 Background

We give here a very short introduction to runtime verification, and provide a description on some image quality assessment techniques.

Runtime Verification (RV) offers a way for verifying systems as a whole during their execution [2]. The verification is performed at runtime by monitoring system execution paths and states, checking whether any predefined formal logic rules are being violated. Additionally, RV can be used to verify software in combination with user-based interaction, adding more focus toward user specific test-cases, which more likely could uncover end-user experienced defects. However, care should be taken as RV adds an overhead potentially reducing system performance. This overhead could also possibly affect the time sensitivity of systems in such way that they appear to run correctly while the monitor is active, but not after it has been removed, a common problem when checking for e.g. data-races in concurrent execution [2].

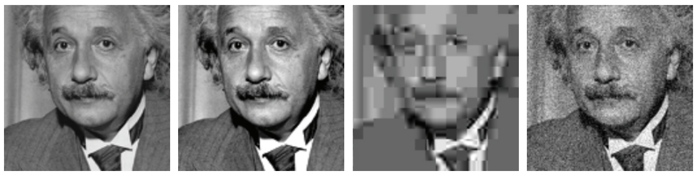


Fig. 1. *Image Quality Assessment* of distorted images using *MSE* and *SSIM* [15]

Image Quality Assessment is used to assess the quality of images or video-streams based on models simulating the *Human Visual System* (*HVS*) [15]. The quality is defined as the *fidelity* or similarity between an image and its reference, and is quantitatively given as the differences between them. Models of the *HVS* describe how different type of errors should be weighted based on their *perceptibility*, e.g. errors in *luminance* are more perceptible than errors in *chrominance* [9]. [1] However, there is a trade-off between the accuracy and performance of algorithms that are based on such models.

Binary comparison is a high performance method for calculating image fidelity, but does not take human perception into account. This could potentially cause problems where any binary differences found are identified as errors even though they might not be visible, possibly indicating false negatives.

Another relatively fast method is the *Mean Squared Error (MSE)*, which calculates the cumulative squared difference between images and their references, where higher values indicate more errors and lower fidelity. An alternative version of *MSE* is the *Peak Signal to Noise Ratio (PSNR)* which instead calculates the peak-error (i.e. noise) between images and their references. This metric transforms *MSE* into a logarithmic decibel scale where higher values indicate fewer errors and stronger fidelity. The *MSE* and *PSNR* algorithms are commonly used to quantitatively measure the performance and quality of lossy compression algorithms in the domain of video processing [6], where one of the goals is to keep a constant image quality while minimizing size, a so-called constant rate factor [7]. This constant rate is achieved, during the encoding process, by dynamically assessing image quality while optimizing compression rates accordingly.

Structural Similarity Index (SSIM) is an alternative measure that puts more focus on modeling human perception, but at the cost of heavier computations. The algorithm provides more interpretable relative percentage measures (0.0-1.0), in contrast to *MSE* and *PSNR*, which present fidelity as abstract values that must be interpreted. *SSIM* differs from its predecessors as it calculates distortions in perceived structural variations instead of perceived errors. This difference is illustrated in Fig. 1, where (b) has a uniform contrast distortion over the entire image, resulting in a high perceived error, but low structural

[1] Luminance is a brightness measure and chrominance is about color information.

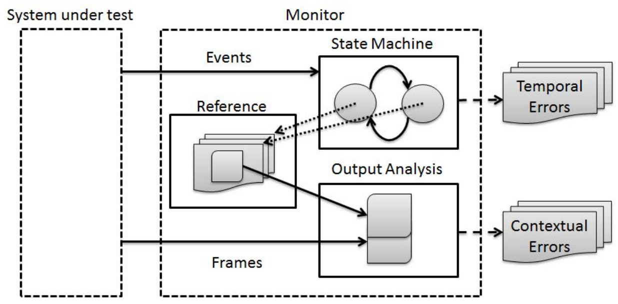


Fig. 2. The *RUGVEF* framework

error. Unlike *SSIM*, *MSE* considers (b), (c), and (d) to have the same image fidelity to the reference (a), but this is clearly not the case due to the relatively large structural distortions in (c) and (d). Tests conducted have shown that *SSIM* provides more consistent results compared to *MSE* and *PSNR* [15]. Furthermore, *SSIM* is also used in some high end applications as an alternative to *PSNR*.[2]

3 The Runtime Graphics Verification Framework

In this section we start by describing the *Runtime Graphics Verification Framework* (*RUGVEF*), for verifying graphics-related system properties. We then state the prerequisites for testing such systems, and finally, we explain how graphical content is analyzed for correctness using image quality assessment.

3.1 The *RUGVEF* Conceptual Model

RUGVEF can be used to enable verification of real-time graphics systems during their execution. Its verification process is composed of two mechanisms: i) checking of execution paths, and ii) verification of graphical output. Together they are used to evaluate *temporal* and *contextual* properties of the system under test. Note that the verification can also include its external runtime environment, such as hardware and drivers.

The verification process, illustrated in Fig. 2, is realized as a monitor application that runs in parallel with the tested system. During this process system and monitor are synchronized through event-based communication where events are used to identify changes in the system's runtime state, thereby verifying the system's temporal correctness. State transitions should always occur when the graphical output changes, allowing legal graphical states to be represented through reference data. These references, either predefined or generated during testing using N-version programming, are in turn used for determining the

[2] `http://www.videolan.org/developers/x264.html`

correctness of state properties through objective comparisons against graphical output produced by the system using appropriate image assessment techniques.

As an example let us consider testing a simple video player having three system control actions (`play`, `pause`, and `stop`), which according to the specification change from 3 different states: from `Idle` to `Playing` with action `play`. From `Playing` it is possible to go to state `Idle` with action `stop` and to `Paused` with action `pause`; and finally from `Paused` to `Playing` with action `play`, and with `stop` to state `Idle`. In this formal definition (the above gives place to a Finite-State Machine — FSM), transitions are used to describe the consequentiality of valid system occurrences that potentially could affect the graphical output. Thus, as the video player is launched the monitor application is started and initialized to the video player's `Idle` state, specifying during this state that only completely black frames are expected. Any graphical output produced is throughout the verification intercepted and compared against specified references, where any mismatches detected correspond to contextual properties being violated. At some instant, when one of the video player's controls is used, an event is triggered, signaling to the monitor that the video player has transitioned to another state. In this case, there is only one valid option and that is the event signaling the transition from `Idle` to `Playing` state (any other events received would correspond to temporal properties being violated). As valid transitions occur, the monitor is updated by initializing the target state, in this case the `Playing` state, changing references used according to that state's specifications.

3.2 Prerequisites

There is a limitation in using comparisons for evaluating graphical output. To illustrate this consider a moving object being frame-independently rendered at three different rendering speeds, showing that during the same time period, no matter what frame rate is used, the object will always be in the same location at a specific time. The problem is that rendering speeds usually fluctuate, causing consecutive identical runs to produce different frame-by-frame outputs. For instance two runs having the same average frame rate but with varying frame-by-frame results, will make it impossible to predetermine the references that should be used. For this reason, the rendering during testing must always be performed in a time-independent fashion. That is, a moving object should always have moved exactly the same distance between two consecutively rendered frames, no matter how much time has passed.

3.3 Image Quality Assessment for Analyzing Graphical Output

Analysis of graphical output is required in order to determine whether contextual properties of real-time graphics systems have been satisfied. *RUGVEF* achieves this by continuously comparing the graphical output against predefined references. We discuss two separate image quality assessment techniques for measuring the similarity of images: one based on absolute correctness, and the other based on perceptual correctness.

Absolute correctness is assessed using binary comparison, where images are evaluated pixel by pixel in order to check whether they are identical. This technique is effective for finding differences between images that are otherwise difficult or impossible to visually detect, which could for instance occur as a result of using mathematically flawed algorithms. However, it is not always the case that non-perceptible dissimilarities are a problem, requiring in such cases that a small tolerance threshold is introduced in order to ignore acceptable differences. One example of this could occur when the monitored system generates graphical output using a Graphical Processing Unit. based runtime platform, conforming to the *IEEE 754* floating point model[3], while its reference generator is run on a *x86 CPU* platform, using an optimized version of the same model[4], possibly causing minor differences in what otherwise should be binarily identical outputs.

Perceptual correctness is estimated through algorithms based on models of the human visual system, and is used for determining whether images are visually identical. Such correctness makes graphics analysis applicable to the output of physical video interfaces which compresses images into lossy color spaces [15,9], with small effects on perceived quality [9], but with large binary differences.

We have evaluated the three common image assessment techniques, *MSE*, *PSNR*, and *SSIM*, which are based on models of the *HVS*. Although *MSE* and *PSNR* are the most computationally efficient and widely accepted in the field of image processing, we have found *SSIM* to be the best alternative. The reason for this is that *MSE* and *PSNR* are prone to false positives and present fidelity as abstract values that need to be interpreted. As an example, when verifying the output from a physical video interface we found that an unacceptably high error threshold was required in order for a perceptually correct video stream to pass its verification. *SSIM* on the other hand was found to be more accurate, also presenting results as concrete similarity measure given as a percentage (0.0-1.0). Additionally, both *MSE* and *PSNR* have recently received critique due to lacking correspondence with human perception [15]. The main problem with *SSIM* is that current implementations are not efficient.

4 Case Study - *CasparCG*

In order to evaluate the feasibility of our framework, a case study was performed in an industrial setting where we created, integrated, and evaluated a verification solution based on *RUGVEF*. We first describe *CasparCG*, and we then show how testing of *CasparCG* was improved using *RUGVEF*.

4.1 *CasparCG*

The development of *CasparCG* started in 2005 as an in-house project for on-air graphics and was used live for the first time during the 2006 Swedish elections [13]. Developing this in-house system enabled *SVT* to greatly reduce costs

[3] `http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf`

[4] `http://msdn.microsoft.com/en-us/library/e7s85ffb.aspx`

by replacing expensive commercial solutions with a cheaper alternative. During 2008 the software was released under an open-source license, allowing external contributions to the project. *CasparCG* 2.0, was released in April 2012 with the successful deployment in the new studios of the show *Aktuellt* [12]. During broadcasts *CasparCG* renders on-air graphics such as bumpers, graphs, news tickers, and name signs. All graphics are rendered in real-time to different video layers that are composed using alpha blending into a single video-stream. The frame rate is regulated by the encoding system used by the broadcasting facility.

The broad range of features offered by *CasparCG* allows the replacement of several dedicated devices during broadcasting (e.g. video servers, character generators, and encoders), making it a highly critical component as failures could potentially disrupt several stages within broadcasts. The system is expected to handle computationally heavy operations during real-time execution, e.g. high quality deinterlacing[5] and scaling of high definition videos. A single program instance can also be used to feed several video-streams to the same or different broadcasting facilities, requiring good performance and reliability.

CasparCG is incrementally developed and is mainly tested through code reviews and ocular inspections. Code reviews are performed continuously throughout the iterative development, roughly every two weeks and also before any new version release. Reviews usually consist of informal walkthroughs where either the full source code or only recently modified sections are inspected, in order to uncover possible defects. Ocular inspections are performed during the later stages of the iterative development, when *CasparCG* is nearing a planned release. The inspections consist of testers enumerating different combinations of system functionalities and visually inspecting that the output produced looks correct.

Whenever an iteration is nearing feature completion, an alpha build is released, allowing users to test the newly added functionality while verifying that all previously existing features still work as expected. Once an iteration becomes feature complete, a beta build is released that further allows users to test system stability and functionality. As defects are reported and fixed, additional beta builds are released until the iteration is considered stable for its final release. Alpha and beta releases are viewed, by the development team, as a cost-effective way for achieving relatively large code coverage levels, where the assumption is that users try more combinations of features, compared to the in-house testing, and that the most commonly used features are tested the most.

4.2 Verifying *CasparCG* with *RUGVEF*

RUGVEF was integrated into the testing workflow of *CasparCG* with the aim of complementing existing practices (particularly ocular inspections), in order to improve the probability of detecting errors, while maintaining the existing reliability levels of its testing process. In this section, we present our contribution

[5] A process where an *interlaced* frame consisting of two interleaved frames (fields) are split into two full *progressive* frames.

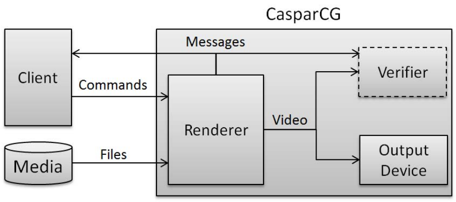


Fig. 3. The verifier is implemented as an output module, running as a part of *CasparCG*

to the testing of *CasparCG*, consisting of two separate verification techniques: local and remote, allowing the system to be verified alternatively on the same and different machine. We also present our optimized *SSIM* implementation, used for real-time image assessment, and a theoretical argumentation on how our approach is indeed an optimization in relation to a reference implementation [4].

Local Verification. During local verification, the verification process is concurrently executed as a plugin module inside *CasparCG*, allowing output to be intercepted without using middleware or code modifications. Fig. 4.2 illustrates that the verifier is running as a regular output module inside *CasparCG*, directly intercepting the graphical output (i.e. video) and the messages produced.

The main difficulty of verifying *CasparCG* is to check its output as it is dynamically composed of multiple layers. Consider the scenario where a video stream, initially composed by one layer of graphics, is verified using references. In this case, the reference used is simply the source of the graphics rendered. However, at some point, as an additional layer is added, the process requires a different reference for checking the stream that now is composed of two graphical sources. The difficulty, in this case, is to statically provide references for each possible scenario where the additional layer has been added on top of the other (as this can happen at any time). As a solution, we instead analyze the graphical output through a reference implementation that mimics basic system functionalities of *CasparCG* (e.g. blending of multiple layers). Using the original source files, the reference implementation generates references at runtime which are expected to be binarily equal to the graphical output of *CasparCG*. The reference implementation only needs to be verified once, unless new functionality is added, as it is not expected to change during *CasparCG*'s development.

Another problem of verifying *CasparCG* lies in defining the logic of the system, where each additional layer or command considered would require an exponential increase in the number of predefined states. For example an FSM representing a system with two layers would only require half the amount of states compared to an FSM representing the same system with three layers. In order to avoid such bloated system definitions, we instead define a generic description of *CasparCG* where one state machine represents all layers which are expected to be functionally equal. This allows temporal properties of each layer to be monitored separately while the reference implementation is used for checking contextual properties of the complete system.

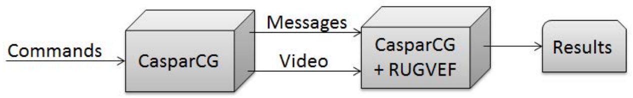


Fig. 4. The remote verification uses two instances of *CasparCG*

The process in local verification is computationally demanding, affecting the system negatively during periods of high load, thus making verification inapplicable during stress-testing. Another limitation identified was that not all components of the system are verifiable; that it is impossible to check the physical output produced by *CasparCG*, which could be negatively affected by external factors (e.g. hardware or drivers). So, in order to more accurately monitor *CasparCG*, with minimal overhead and including its physical output, we further extended our implementation to include remote verification.

Remote Verification. During remote verification, the verifier is executed non-intrusively on a physically different system. Fig. 4 shows this solution, consisting of two *CasparCG* instances running on separate machines, where the first instance receives the commands and produces the output, and the second instance captures the output and forwards it to the *RUGVEF* verification module.

The main problem of remote verification is that the video card interface of *CasparCG* compresses graphical content, converting it from the internal *BGRA* color format to the *YUV420* color format, before transmitting it between the machines. These compressions cause data loss, making binary comparisons inapplicable, instead requiring that the output is analyzed through other image assessment techniques that are based on the human visual system. In this implementation, we chose to use *SSIM*, as it seems to be the best alternative for determining whether two images are perceptually equal. However, the reference *SSIM* implementation [4] is only able to process one frame every few seconds, making real-time analysis of *CasparCG*'s graphical output impossible (as it is produced at a minimum rate of 25 frames per second). In what follows we discuss specific optimizations performed in order to make *SSIM* applicable to the *RUGVEF* verification process of *CasparCG*.

On the Implementation of SSIM. The main challenge of improving the implementation of *SSIM* consisted in achieving the performance that would allow the algorithm to be minimally intrusive while keeping up with data rate of *CasparCG*. The main bottleneck concerning efficiency in current implementations of *SSIM* is the quadratic time complexity, $O(N^2M^2)$, depending on the HDTV resolution (N), and on the window size (M) used in the fidelity measurements. In order to improve the performance, we implemented the algorithm using *Single-Instruction-Multiple-Data* instructions (*SIMD*) [8], allowing us to perform simultaneous operations on vectors of 128 bit values, in this case four 32 bit floating point values using one single instruction. Also, in order to fully utilize *SIMD*, we chose to replace the recommended window size of $M = 11$ in [15] with M=8, allowing calculations to be evenly mapped to vector sizes of four elements (i.e. two vectors per row).

Furthermore, we parallelized our implementation by splitting images into several dynamic partitions, which are executed on a task-based scheduler, enabling load-balanced cache-friendly execution on multicore processors [5]. Dynamic partitions enable the task-scheduler to more efficiently balance the load between available processing units [5], by allowing idle processing units to split and steal sub-partitions from other busy processing units' work queues. Using all processors, we are able to achieve a highly scalable implementation.

The final time complexity achieved by our optimized *SSIM* implementation is $O((N^2\ (M^2+24))/(12p))$, where p is the number of available processing units, allowing *SSIM* calculations to be performed in real-time on consumer level hardware at *HDTV* resolutions.

5 Experimental Results

In this section we show: i) The errors found while verifying *CasparCG* using *RUGVEF*, ii) Previously known defects (injected back into *CasparCG*) we could detect, iii) The improvements in terms of accuracy and performance of our optimized *SSIM* implementation w.r.t the reference implementation.

5.1 Previously Unknown Defects

Using *RUGVEF* we were able to detect five previously unknown defects (presented in the order of their severity, as assessed by the developers), namely: i) Tinted colors, ii) Arithmetic overflows during alpha blending, iii) Invalid command execution, iv) Missing frames during looping, v) Minor pixel errors.

Tinted Colors. Using remote verification, we found a defect where a video transmitted by *CasparCG*'s video interface had slightly tinted colors compared to the original source (i.e. the reference). The error was caused by an incorrect *YUV* to *BGRA* transformation that occurred between *CasparCG* and the video interface. Such problems are normally difficult to detect as both the reference and the actual output looks correct when evaluated separately where differences only are apparent during direct comparisons.

Arithmetic Overflows During Alpha Blending. Using *RUGVEF*, we found that in video streams consisting of multiple layers some small "bad" pixels appear, due to a pixel rounding defect. This defect caused arithmetic overflows during blending operations, producing errors as shown in Fig. 5 (b) (seen as blue pigmentations[6]). Since these errors only occur in certain cases and possibly affecting very few pixels, detection using ocular inspections is a time-consuming process requiring rigorous testing during multiple runs.

[6] In B&W this is seen as the small grey parts in the white central part of the picture.

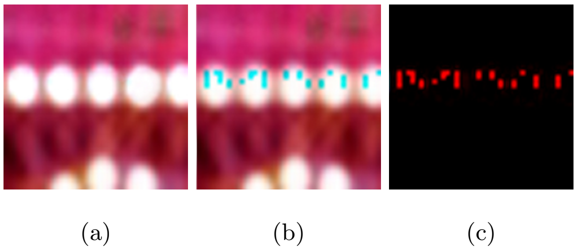


Fig. 5. Pixel rounding defect causing artifact to appear in image (b), highlighted using red color (grey in B&W) in (c), which are not visible in the reference (a)

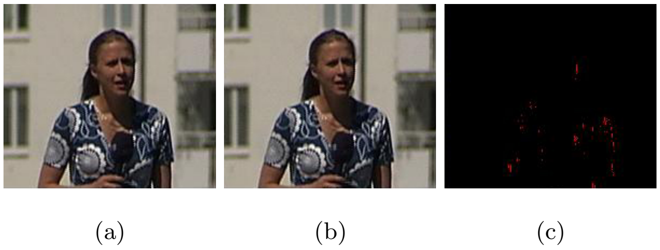


Fig. 6. The output (b) is perceptually identical to the reference (a) while still containing minor pixels errors (c)

Invalid Command Execution. Using *RUGVEF*, we found that the software in certain states accepted invalid commands. For instance it was possible to stop and pause images while in the `Idle` state and to pause while in the `Paused` state. Executing commands on non-existing layers caused unnecessary layers to be initialized, consuming resources in the process. Without *RUGVEF*, this defect would only have been detected after long consecutive system runs, where the total memory consumed would be large enough to be noticed. Furthermore, the execution of these invalid commands produced system responses that indicated successful executions to clients (instead of producing error messages), probably affecting both clients and developers in thinking that this behavior was correct.

Missing Frames During Looping. Using *RUGVEF*, we detected that frames were occasionally skipped when looping videos. The cause of this defect is still unknown and has not been previously detected due to the error being virtually invisible, unless videos are looped numerous times (since only one frame is skipped during each loop).

Minor Pixel Errors. Using local verification, we detected that minor pixel deviations occurred to the output of *CasparCG* that sometimes caused pixel errors of up to 0.8%. These errors are perceptually invisible and could only be detected by using the binary image assessment technique. Fig. 6 shows an example of

Table 1. Previously fixed defects that were injected back into *CasparCG* in order to test whether they are detectable using *RUGVEF*

Rev	Description	Found
N/A	Flickering output due to faulty hardware.	yes
2717	Red and blue color channels swapped during certain runs.	yes
2497	Incorrect buffering of frames for deferred video input.	no
2474	Incorrect calculations in multiple video coordinate transformations.	no
2410	Frames from video files duplicated due to slow file I/O.	yes
2119	Configured RGBA to alpha conversion sometimes not occurring.	yes
1783	Missing alpha channel after deinterlacing.	yes
1773	Incorrect scaling of deinterlaced frames.	no
1702	Video seek not working.	no
1654	Video seek not working in certain video file formats.	no
1551	Incorrect alpha calculations during different blending modes.	no
1342	Flickering video when rendering on multiple channels.	yes
1305	De-interlacing artifacts due to buffer overflows.	no
1252	Incorrect wipe transition between videos.	no
1204	Incorrect interlacing using separate key video.	no
1191	Incorrect mixing to empty video.	no

such a case, where the output in (b) looks identical to the reference in (a) but where small differences have been detected (c).

5.2 Previously Known Defects

In order to evaluate the efficiency of our conceptual model, we injected several known defects into *CasparCG* and tested whether these could be found using *RUGVEF*. The injected defects were mined from the subversion log of *CasparCG* [1] by inspecting the last 12 months of development, scoping the large amount of information while still providing enough relevant defects. In table 1, we present a summary of the gathered defects, where the first column contains the revision id of the log entry, the second a short description of the defect, and the third column indicates whether the defects were possible to detect using *RUGVEF*.

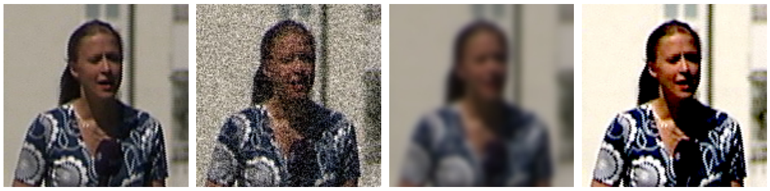
Using *RUGVEF* we were able to detect 6 out of 16 defects that were injected back into *CasparCG*. The defects that could not be found were due to limited reference implementation, which only partially replicated existing functionalities of *CasparCG*. For instance, our reference implementation did not include the scaling of frames or the wipe transition functionalities which made the defects, found in revision 1773 and 1252 respectively, impossible to detect as appropriate references could not be generated.

5.3 Performance of the Optimized SSIM Implementation

We performed our speed improvement benchmarks of our optimized *SSIM* implementation on a laptop computer having 8 logical processing units, each running

Table 2. The optimized *SSIM* implementation compared against a reference implementation at different video resolutions

Implementation	720x576 (SD)	1280x720 (HD)	1920x1080 (Full HD)
Optimized	129 fps	55 fps	25 fps
Reference	1.23 fps	0.55 fps	0.24 fps



(a) R: 1.000 O: 1.000

(b) R: 0.719 O: 0.714

(c) R: 0.875 O: 0.875

(d) R: 0.699 O: 0.686

Fig. 7. The results of performing *SSIM* calculation using our optimized implementation (O) and the reference implementation (R) for an undistorted image (a), noisy image (b), blurred image (c), and an image with distorted levels (d)

at 2.0 GHz[7](which is considerably slower than the target server level computer). Each benchmark consisted of comparing the optimized *SSIM* implementation against the original implementation using the three most common video resolutions, standard definition *(SD)*, high definition *(HD)*, and full high definition *(Full HD)*, by measuring the average time for calculating *SSIM* for 25 randomly generated images. The results of our benchmarks are presented in table 2, showing that our optimized *SSIM* implementation is up to 106 times faster than the original implementation. This increase is larger than the theoretically expected increase of 80 times (calculated using our final time complexity in section 4.2), since our optimized *SSIM* implementation performs all calculations in a single pass, thereby avoiding the memory bottlenecks which existed in the original *SSIM* implementation. Using our implementation, we were able to analyze the graphical output of *CasparCG* in real-time for *Full HD* streams.

Additionally, we also performed an accuracy test by calculating *SSIM* for different distortions in images, comparing the results of our optimized *SSIM* implementation with the results of the original implementation. In Fig. 7, we present the values produced by our optimized *SSIM* implementation “O” and the values produced by the original implementation “R” for the following four types of image distortions: undistorted (a), noisy (b), blurred (c), and distorted levels (d). The result shows that the accuracy of both *SSIM* implementations is nearly identical, as the differences between the values are very small.

[7] Intel Core i7-2630QM.

6 Related Work

The following works address issues related to the testing of graphics: the tool *Sikuli* [16], that uses screenshots as references for automating testing of *Graphical User Interfaces* (*GUI*s); the tool *PETTool* [3], which (semi-) automates the execution of *GUI* based test-cases through identified common patterns; and a conceptual framework for regression testing graphical applications [6]. When it comes to verifying graphical output, the framework in [6] uses a similar approach to *RUGVEF*. However, the tool in [6] focuses on testing system features in isolation, where each test is run separately and targets specific areas of a system (similarly to unit tests). Furthermore, we have also applied our framework to an industrial case study, while there are no indications that something similar has been done in [6], making it difficult to make a detailed comparison.

Finally, the runtime verification tool *LARVA* [2] was used as inspiration source for developing the runtime verification part of *RUGVEF*.

7 Final Discussion

In this paper we have presented *RUGVEF*, a framework for the automatic testing of real-time graphical systems. *RUGVEF* combines runtime verification for checking temporal properties, with image analysis, where reference based image quality assessment techniques are used for checking contextual properties. The assessment techniques presented were based on two separate notions of correctness: absolute and perceptual. We also provided a proof of concept, in the form of a case study, where we implemented and tested *RUGVEF* in the industrial setting of *CasparCG*, an on-air graphics playout system developed and used by *SVT*. The implementation included two separate verification techniques, local and remote, used for verifying the system locally on the same machine with maximal accuracy, and remotely on a different machine, with minimal runtime intrusiveness. Additionally, remote verification allowed the system to be tested as a whole, making it possible to detect errors in the runtime environment (e.g. hardware and drivers). We also created an optimized *SSIM* implementation that was used for determining the perceptual difference between images, enabling real-time analysis of *Full HD* video output produced by *CasparCG*.

When verifying *CasparCG* with *RUGVEF* we identified 5 previously undetected defects. We also investigated whether previously known defects could be detected using our tool, showing that 6 out of 16 injected defects could be found. Lastly, we measured the performance of our optimized *SSIM* implementation, demonstrating a performance gain of up 106 times compared to the original implementation and a negligible loss in accuracy.

Our results show that *RUGVEF* can successfully complement existing verification practices by automating the detection of contextual and temporal errors in graphical systems. Using our framework allows for earlier detection of defects and enables more efficient development through automated regression testing. Unlike traditional testing techniques, *RUGVEF* can also be used to verify the system post deployment. The implementation of the *RUGVEF* tool requires *CasparCG* to run but it should be possible to adapt and apply implementation to other systems as well.[8]

Acknowledgment. We would like to thank Peter Karlsson for his valuable feedback and SVT for giving us the opportunity to develop this work.

References

1. CasparCG (2008), `https://casparcg.svn.sourceforge.net/svnroot/casparcg`
2. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: SEFM, pp. 33–37. IEEE Comp. Soc. (2009)
3. Cunha, M., Paiva, A.C.R., Ferreira, H.S., Abreu, R.: PETTool: A pattern-based GUI testing tool. In: ICSTE 2010, vol. 1, pp. 202–206 (2010)
4. Distler, T.: Image quality assessment (IQA) library (2011), `http://tdistler.com/projects/iqa`
5. Farnham, K.: Threading building blocks scheduling and task stealing: Introduction (August 2007), `http://software.intel.com/en-us/blogs/2007/08/13/threading-building-blocks-scheduling-and-task-stealing-introduction/`
6. Fell, D.: Testing graphical applications. Embedded Sys. Design 14(1), 86 (2001)
7. Li, X., Cui, Y., Xue, Y.: Towards an automatic parameter-tuning framework for cost optimization on video encoding cloud. Int. J. Digit. Multim. Broadc. (2012)
8. Microsoft. Streaming SIMD extensions, SSE (2012), `http://msdn.microsoft.com/en-us/library/t467de55.aspx`
9. Murching, A.M., Woods, J.W.: Adaptive subsampling of color images. In: ICIP 1994, vol. 3, pp. 963–966 (November 1994)
10. Myers, G.J., Sandler, C.: The Art of Software Testing, 2nd edn. John Wiley & Sons (2004)
11. Sharke, M.: Rage PC launch marred by graphics issues (October 2011), `http://pc.gamespy.com/pc/id-tech-5-project/1198334p1.html`
12. S.B.C. (SVT). National news: Aktuellt & Rapport, `http://www.casparcg.com/case/national-news-aktuellt-rapport`
13. S.B.C. (SVT). Swedish election 2006 (2006), `http://www.casparcg.com/case/swedish-election-2006`
14. Timofeitchik, A., Nagy, R.: Verification of real-time graphics systems. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden (May 2012)
15. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: From error visibility to structural similarity. IEEE Trans. on Image Proc. 13(4), 600–612 (2004)
16. Yeh, T., Chang, T.-H., Miller, R.C.: Sikuli: using GUI screenshots for search and automation. In: UIST 2009, pp. 183–192. ACM (2009)

[8] The project can be downloaded from `runtime-graphics-verification.googlecode.com`. See [14] for an extended version of the paper.