Automatic Testing of Real-Time Graphic 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 graphic 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 graphic 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 not been detected before.

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 [7]. 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 [13]. Even though the software itself performed correctly, the error still occurred when executed on systems with certain graphic cards and drivers [1]. To this day, 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. There are several disadvantages with this approach, such as that it requires extensive working hours, is repetitive, and makes regression testing practically inapplicable. Additionally, the subjective definition of

* The length of the paper is 15 pages according to the requirements; the additional pages conform the appendix which is only for reviewing purpose.

correctness induces the possibility that some artifacts might be recognized as errors by some testers, but not by others [12]. 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. This model has the purpose of increasing the probability of finding defects, making verification more efficient and reliable throughout the systems development. 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 detected 5 previously unknown defects that were not previously detected with existing testing practices at *SVT*, and 6 out of 16 known defect 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*.

After starting with some background in next section, we outline our developed conceptual framework *RUGVEF* in section 3 . We present our case study in section 4, of which we show the results in section 5. We discuss related work in section 6, and finally conclude in the last section.

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 [3]. 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 [3].

Image Quality Assessment is used to assess the quality of images or video-streams based on models simulating the *Human Visual System* (*HVS*) [16]. 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*[4] are more perceptible than errors in *chrominance*[5] [11]. 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 [9]. 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 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*

[4] Brightness measure.
[5] Color information.

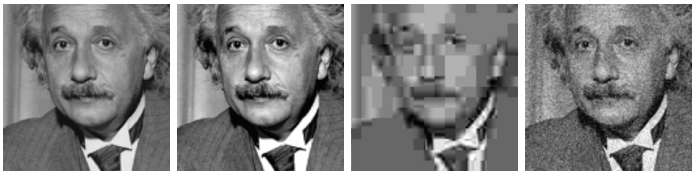


Fig. 1. *Image Quality Assessment* of distorted images using *MSE* and *SSIM* [16].

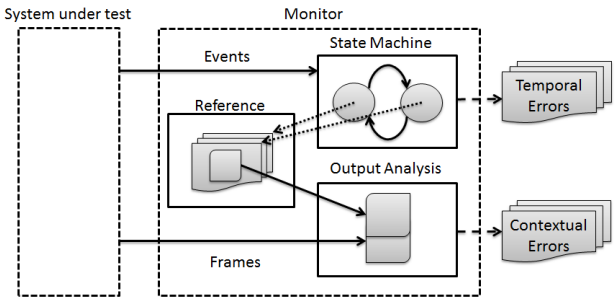


Fig. 2. The *RUGVEF* framework, used for automatically monitoring temporal and contextual properties of the system under test.

provides more consistent results compared to *MSE* and *PSNR* [16]. Furthermore, *SSIM* is also used in some high end applications as an alternative to *PSNR*.[6]

3 The Runtime Graphics Verification Framework

In this section we start by describing the *Runtime Graphics Verification Framework* (*RUGVEF*), that combines runtime verification with image quality assessment in order to create a verification process that is capable of verifying graphics related system properties. We then state the prerequisite 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. In conjunction they are used to evaluate *temporal* and *contextual* properties of the system under

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

test. Note that the verification can also include its external runtime environment, such as hardware and drivers.

The verification process is realized as a monitor application that runs in parallel with the tested system. During this process the system is synchronized with the monitor through event-based communication, where events sent are used to identify changes in the systems runtime state, thereby verifying the systems temporal correctness. State transitions should always occur when the graphical output changes allowing legal graphical states to be represented by reference data. These references are in turn used for determining the correctness of state properties through objective comparisons against graphical output produced by the system using appropriate image assessment techniques.

To show this concept, we consider an example where we test a simple video player having three system control actions (`play`, `pause`, and `stop`), which according to the specification change from 3 different states: from state `Idle` to state `Playing` with action `play`, and with `pause` to state `Paused`. From `Paused` it is possible to go to state `Playing` with action `play` and to state `Idle` with action `stop`; and finally from `Playing` to `Idle` with action `stop`. 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. (See Appendix A for more details.)

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 *GPU*[7] based runtime platform, conforming to the *IEEE 754* floating point model[8], while its reference generator is run on a *x86 CPU* platform, using an optimized version of the same model[9], 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 [16, 11], with small effects on perceived quality [11], 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 [16]. 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

[7] Graphical Processing Unit.

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

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

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 [15]. Developing this in-house system enabled *SVT* to greatly reduce costs 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* [14].

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. (See Appendix B for more details.)

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[10] 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 releases. 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. (See Appendix C for more details.)

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

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

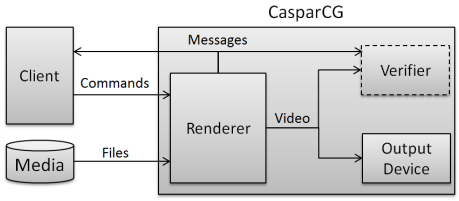


Fig. 3. The verifier is implemented as an input module, running as a part of *CasparCG*.

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 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 [5].

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. 3 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 ex-

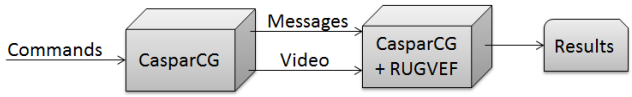


Fig. 4. The remote verification uses two instances of *CasparCG*.

ponential 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. (See Appendix D.)

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. (See Appendix F for more details.)

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 [5] is only able to process one frame every few second, 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*. We have formally specified the algorithm behind SSIM in order to identify the bottlenecks concerning efficiency in current implementations. We have identified that the main problem was that the time complexity was quadratic on the HDTV resolution (N), and on the size of windows used in the fidelity measurements (M). In order to fully utilize modern processor capabilities, we implemented the algorithm using *Single-Instruction-Multiple-Data* instructions (*SIMD*) [10], allowing us to perform simultaneous operations on vectors of 128 bit values, in this case four 32 bit floating point values using a single instruction. Also, in order to fully utilize *SIMD*, we chose to replace the recommended window size of $M = 11$ in [16] 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 [6]. Dynamic partitions enable the task-scheduler to more efficiently balance the load between available processing units [6], 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 $O((N^2\ (M^2+24))/(12p))$, allowing *SSIM* calculations to be performed in real-time on consumer level hardware at *HDTV* resolutions, where p is the number of available processing units. (See Appendix G for more details.)

5 Experimental Results

We present here the errors found while verifying *CasparCG* using *RUGVEF*. We then show which previously known defects (injected back into *CasparCG*) we could detect. Finally, we present 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 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. (See Appendix I.)

(a) (b) (c)

Fig. 5. Pixel rounding defect causing artifact to appear in image (b) which are not visible in the reference (a).

(a) (b) (c)

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

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[11]). 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.

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

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

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

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

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 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* [2] 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.

(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).

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 at 2.0 GHz[12](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.

6 Related Work

The following works address issues related to the testing of graphics: the tool *Sikuli* [18], that uses screenshots as references for automating testing of *Graphical*

[12] Intel Core i7-2630QM

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

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

User Interfaces (*GUI*s); the tool *PETTool* [4], which (semi-) automates the execution of *GUI* based test-cases through identified common patterns; and a conceptual framework for regression testing graphical applications [7]. When it comes to verifying graphical output, the framework in [7] uses a similar approach to *RUGVEF*. However, the tool in [7] 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 [7], making it difficult to make a detailed comparison.

Finally, the runtime verification tool *LARVA* [3] 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.

The results above show that *RUGVEF* can indeed successfully complement existing verification practices by automating the detection of contextual and tem-

poral errors in graphical systems. That using the 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, similarly to traditional runtime verification, something that previously was impossible. 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.[13]

References

1. J. Carmack. Quakecon 2011 - John Carmack keynote Q&A, Aug 2011. `http://www.quakecon.org/2011/08/catch-up-on-quakecon-2011/`.
2. CasparCG, 2008. `https://casparcg.svn.sourceforge.net/svnroot/casparcg`.
3. C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *SEFM*, pages 33–37. IEEE Comp. Soc., 2009.
4. M. Cunha, A. C. R. Paiva, H. S. Ferreira, and R. Abreu. PETTool: A pattern-based GUI testing tool. In *ICSTE'10*, volume 1, pages 202–206, 2010.
5. T. Distler. Image quality assessment (IQA) library, 2011. `http://tdistler.com/projects/iqa`.
6. K. Farnham. Threading building blocks scheduling and task stealing: Introduction, Aug 2007. `http://software.intel.com/en-us/blogs/2007/08/13/threading-building-blo%cks-scheduling-and-task-stealing-introduction/`.
7. D. Fell. Testing graphical applications. *Embedded Sys. Design*, 14(1):86–86, 2001.
8. I. C. U. (ICU). BT.709 : Parameter values for the HDTV standards for production and international programme exchange, Apr 2002. `http://www.itu.int/rec/R-REC-BT.709/en`.
9. X. Li, Y. Cui, and Y. Xue. Towards an automatic parameter-tuning framework for cost optimization on video encoding cloud. *Int. J. Digit. Multim. Broadc.*, 2012.
10. Microsoft. Streaming SIMD extensions (SSE), 2012. `http://msdn.microsoft.com/en-us/library/t467de55.aspx`.
11. A. M. Murching and J. W. Woods. Adaptive subsampling of color images. In *ICIP'94*, volume 3, pages 963–966, Nov 1994.
12. G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, second edition, 2004.
13. M. Sharke. Rage PC launch marred by graphics issues, Oct 2011. `http://pc.gamespy.com/pc/id-tech-5-project/1198334p1.html`.
14. S. B. C. (SVT). National news: Aktuellt & Rapport. `http://www.casparcg.com/case/national-news-aktuellt-rapport`.
15. S. B. C. (SVT). Swedish election 2006. `http://www.casparcg.com/case/swedish-election-2006`.
16. Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Trans. on Image Proc.*, 13(4):600–612, 2004.
17. M. Wright. The Open Sound Control 1.0 specification, Mar 2002. `http://opensoundcontrol.org/spec-1_0`.

[13] The project can be downloaded from `runtime-graphics-verification.googlecode.com`.

18. T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: using GUI screenshots for search and automation. In *UIST'09*, pages 183–192. ACM, 2009.

A The *RUGVEF* Conceptual Model *CasparCG*

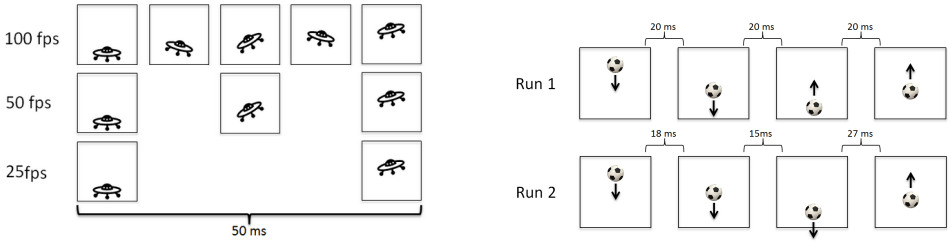
The Runtime Graphics Verification Framework (*RUGVEF*) can be used to enable verification of real-time graphics systems during their execution. Its verification process is composed of two mechanisms: i) checking of execution paths, and ii) verification of graphical output. In conjunction they are used to evaluate *temporal* and *contextual* properties of the system under test.

To illustrate an example of such properties, we consider a simple video player that displays video files on a screen. This application has three controls through which users can start, stop, or pause video playback. In this case, a contextual property might be *that actual video file content is always displayed during playback or that otherwise only empty frames are produced*, and a temporal property might be *that it is only possible to use the pause control during video playback.*

The correctness of properties in real-time graphics is determined through continuous analysis of graphical output (e.g. content displayed by the video player). Traditionally, this type of analysis is performed intuitively through ocular inspections, where testers themselves observe content displayed in order to determine whether system properties are satisfied. However, since ocular inspections heavily rely on the subjective opinions of testers, the process is very difficult (or maybe even impossible) to be programmatically replicated. Thus, in order to automate the verification of graphics related properties, we propose that correctness is determined through objective comparisons against references using appropriate image assessment techniques.

However, there is a limitation in using comparisons for evaluating graphical output. To illustrate this consider the example in Fig. 8-(a), where we show a moving object being frame-independently 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 in Fig. 8-(b), we show two runs having the same average frame rate but with varying frame-by-frame results, making 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.

The analysis of graphical output through comparisons requires that references are provided together with relevant information on when and how they should be used. To show this, we consider the same video player that was previously described. This video player has three system control actions (`play`, `pause`, and `stop`), which according to the specification change from 3 different states: from state `Idle` till state `Playing` with action `play`, and with `pause` to state `Paused`. From `Paused` it is possible to go to state `Playing` with action `play` and to state `Idle` with action `stop`; finally from `Playing` to `Idle` with action `stop`. 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



(a) (b)

Fig. 8. (a) The frame-independent rendering technique renders a moving object at different frame rates, but where the displaced distance during identical periods is the same; (b) Frame-independent rendering produces different results depending on the time elapsed between each rendered frame.

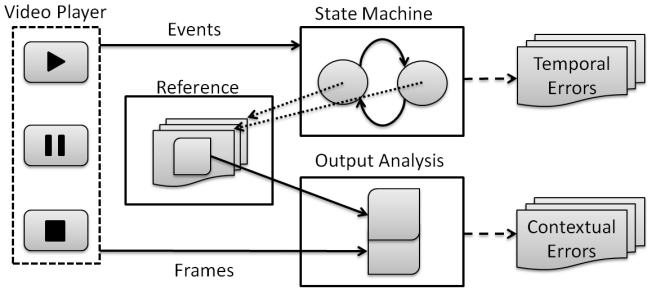


Fig. 9. An implementation of *RUGVEF* used for automatically monitoring the simple video player's temporal and contextual properties.

that potentially could affect the graphical output. For instance describing that it is only possible to pause an already playing video. Transitions are labelled with actions (`play`, `pause`, and `stop`) that, when executed, trigger the transitions. States (`idle`, `playing`, and `paused`) represent legal output variations possibly occurring during runtime execution, that is, what we expect to be displayed by the video player. For instance defining the reference that must be used while in the `idle` state in order to check that only empty frames are produced.

We have so far focused on describing the artifacts needed in order to automate the verification of temporal and contextual properties in real-time graphics. We have stated that the correctness of such properties is most easily determined through objective comparisons with references, and that these references must be provided together with a formal definition of the system to be tested. To put these artifacts into context, we consider the example illustrated in Fig. 9, showing an automated verification process that can be used for checking properties of the same video player mentioned throughout this section.

In this example, the verification process has been realized as a monitor application that runs in parallel with the video player. During this process the

video player is synchronized with the monitor through event-based communication, where events sent by the video player are used to identify changes to its runtime state. As previously mentioned, such state transitions should always occur when the output of the video player changes, requiring in this case that the controls `play`, `pause`, and `stop` are used as the triggering points for transmitted events. Legal states and transitions must also be known by the monitor before event-based communication can be used for monitoring the video player's activities. In this case, we provide the same states and transitions as defined by the FSM describe above. Events can thereby be used for representing transitions between states, making it possible for the monitor to track the video player's runtime state and to check its temporal activities. The graphical context during each state is monitored by intercepting the graphical output and comparing it against provided references.

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.

One important issue emerges when the input of a system is handled asynchronously from the graphical output: whenever a command is issued, its effect is delayed by an undefined amount of discrete output intervals before actually being observable. The problem with this non-deterministic delay is that it could affect the event-based communication between a monitor application and the tested system, causing events to arrive prematurely from what is output, thereby making the reference data used in the verification process to become out of sync.

A.1 Solving the Synchronization Problem

We suggest two approaches for solving the synchronization problems presented above. The first is based on tagging the graphical output with unique identifiers, and the second on using comparisons with already provided references.

Frame Tagging One possible method for solving the synchronization problem is by tagging the output produced by the monitored system with unique identifiers. The synchronization is achieved by pairing events with specific frames, in which the triggering commands effect can be observed, and by delaying these until frames tagged with matching identifiers have been received. For instance in Fig. 10, we show a wrapper intercepting and delaying an event tagged with the

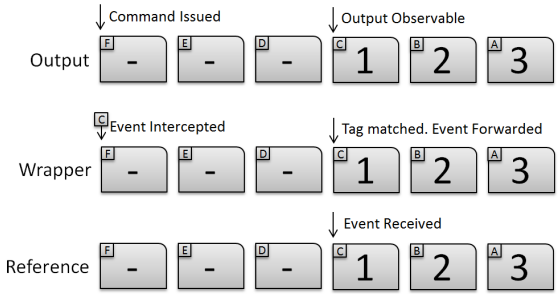


Fig. 10. Synchronization using a wrapper delays events until frames with correct tags are received.

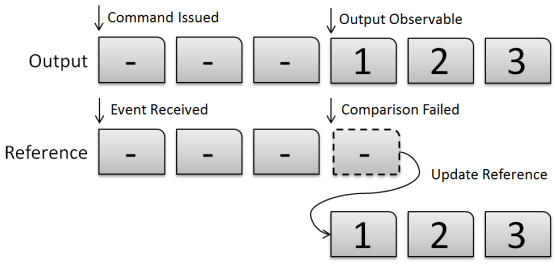


Fig. 11. Synchronization using a reference.

unique identifier `C` until the frame carrying the same identifier has been received, indicating that the change to the output is observable.

The disadvantage of this technique is that it requires the tagging capability of frames to be implemented. This solution might be too intrusive as it may require fundamental changes in the software. It may also prove difficult in keeping the associated metadata correctly tagged while propagating throughout the complete system's software and hardware chain, before finally reaching the verifier.

Reference Based Comparisons As an alternative to frame tagging, the synchronization can be performed by only using reference based comparisons. This process is illustrated in Fig. 11, and is as follows: when an event is received the verifier continues comparing the output against provided references, unchanged in relation to the received event. As a frame mismatch occurs and the comparison fails, the reference is updated and a re-comparison is performed. If the re-comparison succeeds then the synchronization has been successful or otherwise an error has been detected.

However, an issue emerges when additional events are received while still in the synchronization process, causing difficulties in deducing which references to update when a mismatch occurs, requiring that all possible combination of pending changes are tried before proceeding with the updates.

Another issue arises whenever changes to the output cannot be observed (i.e. where changes are outside the scene or totally obscured), causing the re-synchronization to be prolonged or, in the worst case scenario, to never finish. In such cases, the solution is to postpone all synchronization until later stages where mismatches have been detected, but requiring that all possible combinations of pending changes are tried before updates to references can be performed.

Thus, the main issue with using reference based synchronization is that the testing of all combinations requires a much higher computational demand. This could affect the total outcome of the verification negatively, making some defects to not occur until the verifier has been removed. In order to avoid such complications, we propose that the verification is instead scoped so that changes to the output should always be observable and that no additional commands should be issued, while still in the process of synchronizing references.

In what follows we show how reference based image quality assessment techniques are used to determine the correctness of graphical output of systems.

B *CasparCG*

These frequencies are regulated through a hardware reference clock connected to the system, usually consisting of a sync pulse generator that provides a highly stable and accurate pulse signal. This signal is used globally in the broadcasting facility in order to ensure that all devices are synchronized.

CasparCG takes advantage of modern multi-core and heterogeneous computer architectures by implementing a pipelined processing design, allowing the system to run different interdependent processing stages in parallel, and possibly on different processing devices. This design increases the throughput of the system, but at the cost of increased latency that is proportional to the number of active pipeline stages. In order to increase the performance reliability of *CasparCG*, the system uses buffers for effectively hiding transient performance drops within its rendering pipeline. This buffering technique further increases the latency between the input and the output,

C *CasparCG*

Parts of this process has been automated using two different tools: *JMeter*, an application that automatically triggers commands (possibly in a random order) according to predefined schedules; and *Log Repeater*, an application that repeats previous system runs through runtime logs produced by *CasparCG*. These tools allow testers to run the software in the background, checking stability during longer periods of runs. As defects are found, the *Log Repeater* is used for reproducing errors, debugging the system, and verifying possible fixes.

D Local Verification

The runtime state of *CasparCG* is tracked by connecting to its existing messaging capability, which is implemented according to the *Open Sound Control* (*OSC*) protocol [17]. Transmitted messages consist of two fields where the first field carries the name and the origin of the message (similarly to hyperlink addresses), while the second carries its arguments. In the context of *CasparCG*, the first field corresponds to the system's capability of handling multiple outputs (i.e. channels), where each output is composed of several graphical streams (i.e. layers), and where the second field carries any additional information, such as the file currently playing. For instance, the event of playing the file `movie.mp4` on layer two, channel one, would result as the transmitted *OSC* message ⟨`channel/1/layer/2/play`, `movie.mp4`⟩ where the first part is the address (with `channel/1/layer/2` the origin, and `play` the name), and the second the data (the mp4 file). This allows source files to be determined dynamically at runtime.

We formally define the logic of *CasparCG* through *Extensible Markup Language* (*XML*) scripts (cf. Appendix E, listing 1.1). Events are described as regular expressions that are mapped to *OSC* messages and paired with predefined properties. Properties are used for controlling the behavior of the reference implementation; each property corresponds to a replicated functionality of *CasparCG*.

As already mentioned, *CasparCG* implements a buffered rendering pipeline in order to increase the performance and the reliability of the system. However, this architectural design introduced a latency between received commands and the visible output, causing *OSC* messages to be transmitted prematurely and thereby making the reference implementation to become out of sync. As a solution to this problem, we implemented the frame tagging scheme presented in appendix A.1, describing how synchronization is achieved by pairing frames with events, in this case *OSC* messages, using unique identifiers.

E Example XML script for *CasparCG*

In listing 1.1, the event `pause` at line 13 is paired with the property `suspend="true"`, defining that the reference implementation should pause/suspend the playback on layers corresponding to the address defined by the regular expression `channel/0/layers/[0-9]+/` of received *OSC* messages with the name `pause`.

```
<?xml version="1.0" encoding="utf-8"?>                          1
<state-machine start="idle">                                    2
    <states>                                                    3
        <state name="idle">                                     4
            <transition event="channel/0/layer/[0-9]+/play"     5
                        target="playing"/>                      6
        </state>                                                7
        <state name="playing">                                  8
            <transition event="channel/0/layer/[0-9]+/stop"     9
```

```
                    target="idle" reset="true"/>                    10
            <transition event="channel/0/layer/[0-9]+/eof"          11
                    target="paused" reset="true"/>                  12
             <transition event="channel/0/layer/[0-9]+/pause"       13
                    target="paused" suspend="true"/>                14
        </state>                                                    15
        <state name="paused">                                       16
            <transition event="channel/0/layer/[0-9]+/stop"         17
                    target="idle" reset="true"/>                    18
            <transition event="channel/0/layer/[0-9]+/play"         19
                    target="playing" suspend="false"/>              20
        </state>                                                    21
    </states>                                                       22
</state-machine>                                                    23
```

Listing 1.1. XML script defining the generic state machine of *CasparCG*.

F Remote Verification

Remote verification requires that *OSC* messages are transmitted between the machines. We originally tried to address this by using the *TCP* network protocol. However, the latency introduced by the overhead of this protocol (i.e. buffering, error correction, and packet acknowledgements) caused a synchronization problem that made remote verification impossible (i.e. the opposite of the synchronization problem described in section 3.1). Thus, we instead used the *UDP* protocol allowing us to attain sufficiently low transmission latencies.

Additionally, the implemented synchronization by frame tagging scheme is unusable during remote verification. The problem lies in that this synchronization method requires tags to always be transmitted together with frames, but where tags received by the second machine are impossible to forward together with frames through *CasparCG*'s rendering pipeline (to the verification module), without making significant architectural changes to *CasparCG*. Thus, to be able to synchronize the remote verification with the output of *CasparCG*, we have also implemented the reference based synchronization scheme (cf. appendix A.1).

G On the implementation of SSIM

In this appendix we first describe the basics of *SSIM* in order to show identified bottlenecks, followed by showing how these are reduced by mapping the algorithm to modern processor architectures. We finally present our algorithmic optimization in the form of a preprocessing step that reduced the amount of data processed by *SSIM*, while maintaining sufficient accuracy.

In order to determine fidelity, *SSIM* decomposes the image similarity measurement into three independent components that the human visual system is more sensitive to [16]: i) *luminance*, the mean pixel intensity between images;

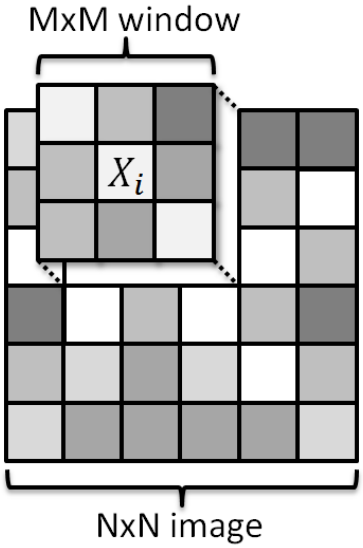


Fig. 12. The M by M window for the pixel X_i in the N by N image.

ii) *contrast*, the variance of pixel intensity between images; iii) *structure*, the pixel intensity after subtracting the mean intensity and normalizing the variance between images. These metrics are estimated for each pixel by sampling neighboring pixels in windows of predetermined sizes, as shown in Fig. 12. The pixel values inside the windows are weighted according to some form of distribution (e.g. linear or *Gaussian* distribution), where samples closer to the center are considered more important.

In order to show the bottlenecks, we provide the following specification of *SSIM*.

Specification 1 *Let us consider two images represented by two distinct N by N sequences of pixels, $X = \{X_i|1, 2, ..., N^2\}$ and $Y = \{Y_i|1, 2, ..., N^2\}$, where X_i and Y_i are the values of the ith pixel sample in X and Y. Let $x_i = \{x_{ij}|1, 2, ..., M^2\}$ and $y_i = \{y_{ij}|1, 2, ..., M^2\}$ be windows of M by M sequences centered around the ith pixel in X and Y, where x_{ij} and y_{ij} are the values of the jth pixel samples in x_i and y_i, and where these windows are evaluated using the weighting function w. Then* SSIM *is specified as [16]:*

$$SSIM(X,Y) = N^{-2}\sum_{i=1}^{N^2} luminance(X_i,Y_i)\times constrast(X_i,Y_i)\times structure(X_i,Y_i) \tag{1}$$

$$luminance(x,y) = (2\mu_x\mu_y + C_1)/(\mu_x^2 + \mu_y^2 + C_1) \tag{2}$$

$$constrast(x,y) = (2\sigma_x\sigma_y + C_2)/(\sigma_x^2 + \sigma_y^2 + C_2) \tag{3}$$

$$structure(x, y) = (\sigma_{xy} + C_3)/(\sigma_x + \sigma_y + C_3) \tag{4}$$

$$E[x] = \mu_x = \sum_{j=1}^{M^2} w_j x_{ij} \tag{5}$$

$$\sigma_x = \sqrt{\sum_{j=1}^{M^2} w_j (x_{ij} - \mu_x)^2} \tag{6}$$

$$\sigma_{xy} = \sum_{j=1}^{M^2} w_j (x_{ij} - \mu_x)(y_{ij} - \mu_y) \tag{7}$$

We found that the main bottlenecks of implementing this specification are equations (5), (6) and (7) each having respectively the time complexity of $O(N^2M^2)$, where $N^2 = 1920 \times 1080$ for *HDTV* resolutions [8] and M is the size of the windows that are used in the fidelity measurements.

In order to fully utilize modern processor capabilities, we implemented the algorithm using *Single-Instruction-Multiple-Data* instructions (*SIMD*) [10], allowing us to perform simultaneous operations f on vectors of 128 bit values, in this case four 32 bit floating point values $x_j = \{x_1, x_2, x_3, x_4\}$, using a single instruction f_{SIMD} as illustrated by Specification 2. Also, in order to fully utilize *SIMD*, we chose to replace the recommended window size of $M = 11$ in [16] with M=8, allowing calculations to be evenly mapped to vector sizes of four elements (i.e. two vectors per row).

Specification 2

$$f_{SIMD}(\boldsymbol{x}, \boldsymbol{y}) = \{f(x_1, y_1), f(x_2, y_2), f(x_3, y_3), f(x_4, y_4)\} \tag{8}$$

By utilizing *SIMD*, we are able to reduce the time complexity of equations (5), (6) and (7) from $O(N^2M^2)$ to $O(N^2M^2/4)$.

Additionally, we implemented *SSIM* using a cache friendly single pass calculation, allowing **??** to be calculated in one pass. This implementation greatly improved the performance, in comparison with the reference implementation [5], where entire images had to be processed in multiple passes and thereby requiring multiple memory loads and stores for each pixel [14].

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 [6]. This parallelization is illustrated in Fig. 13, where an image is split into four partitions, mapping execution on all available processing units, and where the result for each partition is merged into the cumulative *SSIM* measure. Dynamic partitions enables the task-scheduler to more efficiently balance the load between

[14] Processing large amounts of data that does not fit into the cache memory requires the same data to be transferred between the cache and the main memory multiple times, which is considerably slower than performing the actual calculations.

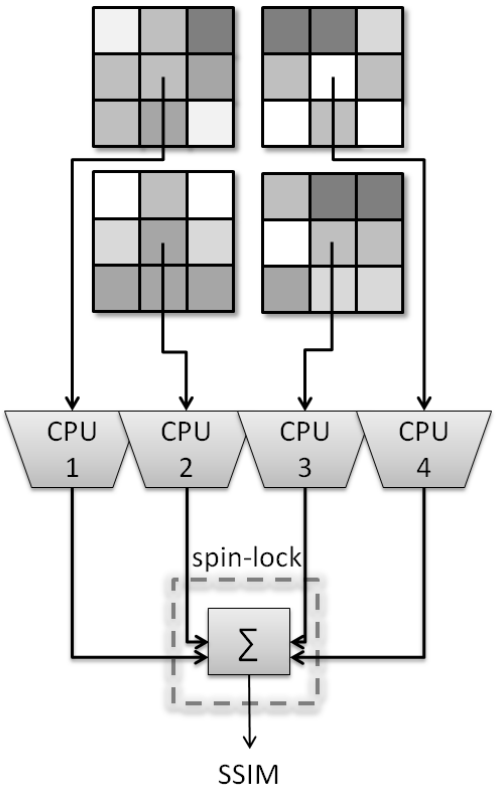


Fig. 13. Images are divided into partitions which are processed on different central processing units. The results of the partitions are then summed in a critical section using a low contention spinlock.

available processing units [6], 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 with the total time complexity of $O((N^2M^2)/(4p))$, where p is the number of available processing units.

Lastly, we further reduced the time complexity of *SSIM* by discarding chrominance and calculating results based solely on luminance (i.e. instead of calculating *SSIM* based on all three, red, green, and blue, color channels), where this optimization is possible due to the assumption that the *HVS* is more sensitive to luminance than chrominance [11]. Using only luminance, time complexity of the *SSIM* algorithm is $O((N^2M^2)/(4p3) + 10N^2)$, where $O(5N^2)$ is the time complexity of the luminance transformation specified by the *BT.709* standard for *HDTV* [8], which we specify below.

(a) (b) (c)

Fig. 14. Three images are produced when errors are detected, where (a) is the reference, (b) the actual output, and (c) the highlighted pixel error.

Specification 3 *Let* $\boldsymbol{x} = \{x_i|1, 2, 3, 4\}$ *be a pixel sample where each component respectively evaluate the 8 bit components red, green, blue, and alpha of the pixel sample* x_i*. Then the* BT.709 *luma transformation* Y' *is specified as:*

$$Y'_{BT.709}(\boldsymbol{x}) = 0.2125x_1 + 0.7154x_2 + 0.0721x_3 \tag{9}$$

In this transformation, a weight is assigned for each component in a vector x, where each weight represents the relative sensitivity between colors according to the *HSV*. For example, the blue component is considered least visible by the *HSV* and is, therefore, weighted by the relatively low value of 0.0721. 9 can also be realized using a four component dot product[15] between the pixel sample and a weighting vector. Using the dedicated dot product *SIMD* instruction we are able to reduce the time complexity of 9 from $O(5N^2)$ to $O(N^2)$. Additionally, by parallelizing the equation, similarly to the parallelization of *SSIM* earlier described, we are able to further reduce the time complexity of the luminance transformation to $O(N^2/p)$, where p is the number of available processing units..

The final time complexity achieved by our optimized *SSIM* implementation is $O((N^2\ (M^2+24))/(12p))$, allowing *SSIM* calculations to be performed in real-time on consumer level hardware at *HDTV* resolutions, which is demonstrated in section 5.

H Previously Unknown Defects

The results produced by *RUGVEF* when errors occur are images describing the errors found. For instance, Fig. 14 illustrates one example of such images produced when an error was detected.

I Tinted Colors

[15] An algebraic operation that by multiplying and summing the corresponding elements in two vector returns a single value result called the dot/scalar product, $a \cdot b = \sum_{i=1}^{n} a_i b_i$.

[17] In black and white this is seen as slightly brighter background.

(a) (b)

Fig. 15. The tinting error where the output (b) has a yellowish tint[17] in relation to the reference in (a).

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). This error is visible in Fig. 15, where the actual output (b) has slightly different colors than the reference (a). 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 (see Fig. 15), where differences only are apparent during direct comparisons.

J Invalid Command Execution

Using *RUGVEF*, we found that the software in certain states accepted invalid commands. Fig. 16 illustrates these errors as dashed lines in the automaton describing the expected interactions, showing for instance that it was possible to incorrectly stop idle or non existing layers.

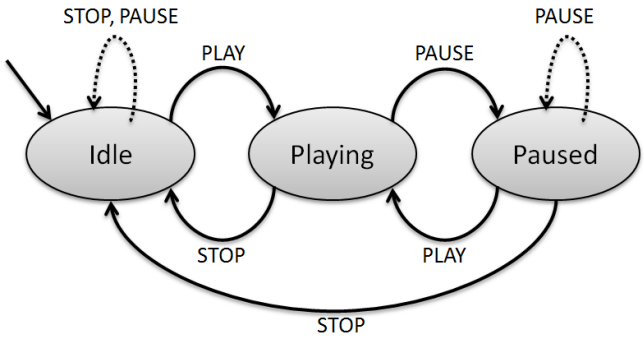


Fig. 16. A state machine showing parts of the formal definition used during the verification of *CasparCG*, where the invalid transitions detected are shown using dashed lines.