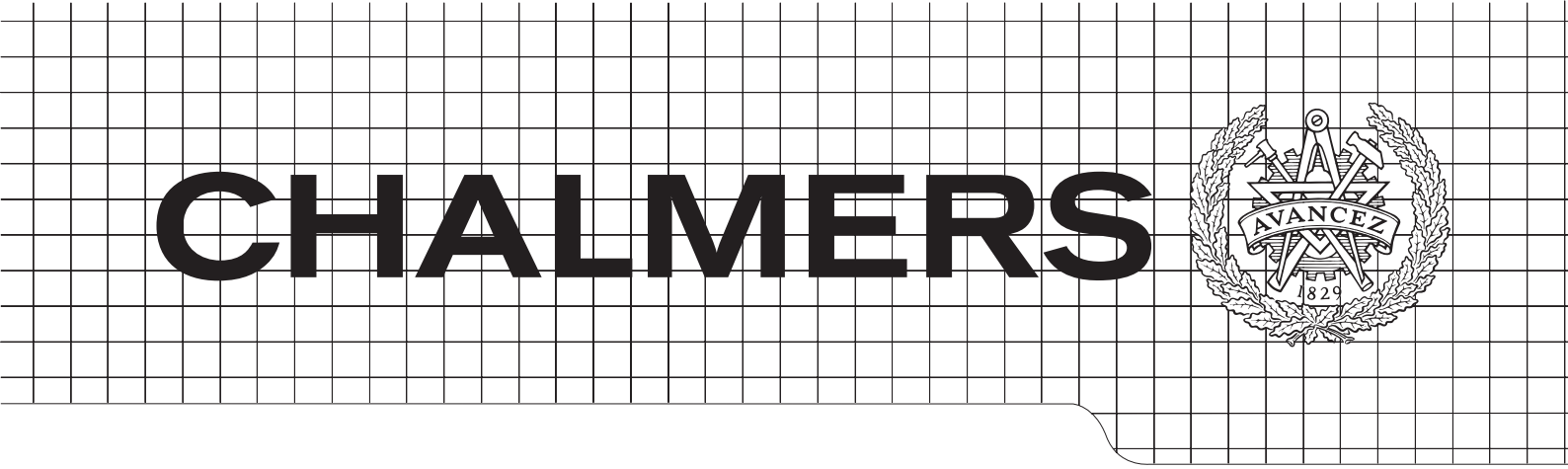


Verification of Real-Time Graphics Systems

Master of Science Thesis in Software Engineering and Technology

Aram Timofeitchik and Robert Nagy

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 15/05/2012

Verification of Real-Time Graphics Systems

Aram Timofeitchik
Robert Nagy

Supervisor: Gerardo Schneider
Examiner: Rogardt Heldal

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden, May 2012

Abstract

Today, software quality assurance is an important part of software development, where increasingly more research is conducted in order to achieve more efficient and reliable verification processes. However, little progress has been made when it comes to verification of real-time graphics systems. To this day, most verification of such applications is performed through repetitive manual labour, where graphical content is ocularly inspected and subjectively verified. In this thesis, we present the Runtime Graphics Verification Framework (RUGVEF), where we have combined runtime verification with image analysis in order to automate the verification of graphical content. We also provide a proof of concept in the form of a case study, where RUGVEF is evaluated in an industrial setting by verifying CasparCG, an on-air graphics playout system. The results of the case study are five previously unknown defects, together with statistics of previously known defects (which were injected back into the system) that could be detected by our tool.

Acknowledgements

First, we would like to express our deepest gratitude to Dr. Gerardo Schneider for his continuous support and guidance throughout our work. We would also like to thank him for his valuable ideas that helped us to better approach the problem of this thesis.

Next, we would like to thank Peter Karlsson for his valuable feedback and the Swedish Broadcasting Corporation (SVT) for giving us the opportunity to develop and test our ideas.

Finally, we would like to thank our families and friends for their continuous support and encouragement throughout the duration of this project.

Contents

1

Introduction

Software verification plays an important role in the software development life-cycle. It provides the means for detection and correction of defects and thereby improving software quality [1]. One of the main challenges of software verification lies in ensuring that sufficient code coverage is achieved [1]. However, with increasingly larger and more advanced software being developed, the sheer number of program states and possible user interactions makes adequate verification, especially when carried out manually, both difficult and expensive to achieve [1]. This is one of the main reasons why automated software verification (such as automatic unit[1] and state-of-the-art model-based testing [2]), which allows faster and more efficient verification, is increasingly seen as an important part of the software verification life-cycle.

With manual testing, defects are found by checking as many combinations of user interactions as possible. Usually, during manual testing, a checklist of tasks is used for making the process more structured and reproducible [1, 3]. However, as tasks must be carried manually, such testing becomes impractical to perform on large systems as often as required. Automatic testing can assist testers by replacing some of the tedious tasks applied by manual testing, thereby reducing costs and improving software quality [1, 4]. Additionally, automation can also be used to improve the software development process, e.g. automated regression tests can enable continuous refactoring and integration [5, 6], and thereby providing the means for iterative development through functional self-testing.

Research in software verification seems to have been mostly focused towards automating various white-box testing methodologies (e.g. unit testing). The drawback of these methodologies is that they focus on the internal parts of the application, requiring both knowledge and access to the source-code and possible third-party components. This type of verification also lacks the means for testing the software in integration with its external components (e.g. operative system, hardware, and drivers). One such problem was encountered by Id Software during the initial release of their video-game Rage, where the game suffered problems with texture artifacts [7]. Even though the software itself performed correctly [8], the error still occurred when executed on systems with certain graphic cards and drivers.

[1] http://www.junit.org/

Today, focus seems to have started to shift towards automated grey- and black-box testing. Areas, such as memory[2] and concurrency[3] and also graphical user-interfaces [9, 10], are starting to receive more attention, while others, such as testing of real-time graphics (e.g. games and dynamic video renderers), still remain mostly unexplored.

Traditional testing techniques are insufficient for obtaining satisfactory code coverage levels in real-time graphics. 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 [11]. Dependencies such as hardware, operating systems, drivers, and external run-times (e.g. *OpenGL* and *Direct3D*) are also generally outside the scope of white-box testing. While properties common to real-time graphics, such as non-determinism and time-based execution, make errors difficult to detect and reproduce.

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 [3]. Additionally, the subjective definition of correctness induces the possibility that some artifacts might be recognized as errors by some testers, but not by others [1]. Furthermore, some errors might not be perceptible, thereby making ocular inspections even more prone to human-error.

In this thesis, we present a conceptual model for automated testing of properties in 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*), defining practices and artifacts needed in order to perform automated verification in coherence with system development. A proof of concept is provided in the form of a case study, where our framework is implemented and evaluated 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*.

The results of the case study are five previously unknown defects that the testing practices at *SVT* had not yet detected, and 6 out of 16 known defect that were injected back into *RUGVEF* 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. Furthermore, we also show that our optimizations of the image assessment technique *SSIM* improved the performance of the algorithm by approximately a factor of 100, while maintaining accuracy.

[2] http://software.intel.com/en-us/articles/intel-inspector-xe/
[3] http://software.intel.com/en-us/articles/intel-parallel-inspector/

In summary our contributions are:

- The conceptual *Runtime Graphics Verification Framework* (*RUGVEF*) for automating the testing of real-time graphics systems.
- A proof of concept demonstrating the feasibility, applicability, and strengths of this framework in the context of the real-time graphics system *CasparCG* at the *Swedish Broadcasting Corporation* (*SVT*).
- Experimental results concerning our proof of concept that was used to test *CasparCG* at *SVT*, where we found five previously unknown defects and demonstrated which of the previously known defects could be detected.
- The optimized implementation of *SSIM*, an image quality assessment technique which previously was not applicable to the real-time setting of *CasparCG*.

This thesis is organized as follows: We start by describing the used research methodologies in Chapter 2. We then provide the preliminaries required to understand the essential concepts in Chapter 3, and outline *RUGVEF*, our developed conceptual framework, in Chapter 4. We present our proof of concept, in the form of a case study of *CasparCG*, in Chapter 5, and show the results of the case study and the optimizations of *SSIM* in Chapter 6. We discuss related respectively future work in Chapters 7 and 8, and finally conclude the work of this thesis, with a short summary and final remarks, in Chapter 9.

2

Research Method

The goal of this thesis was to design a verification framework that was aimed at improving and extending existing software verification techniques, making them applicable to computer graphics, in order to enhance the accuracy and effectiveness of software quality assurance processes of real-time graphics systems.

Research was conducted using the design-science research method [12] with the purpose of contributing, according to [13], "a purposeful IT artifact created to address an important organizational problem", which was evaluated in the context of a holistic case study that is described in [14].

Data collection was performed using: 1) informal interviews with the main tester and the two developers of *CasparCG*, in order to gain knowledge and understanding of the current testing practices used in the project; 2) observations of the current verification workflow that is practiced by *CasparCG*'s project members; and 3) data mining of *CasparCG's* subversion log for statistical analysis of previously known and corrected defects, and their correlated time within the project's software development cycle.

The typical subjects addressed during the interviews were:

- What types of software testing techniques are used?
- How and by whom is the testing performed?
- What types of errors are testers looking for and which defects are more commonly found?
- What types of errors are harder to detect?
- When in the software development cycle are defects mostly detected?
- What are the common problems experienced during testing?
- What improvements could be made to the current verification process?

A hypothesis was then reached that a combination of run-time verification and objective image analysis would allow for improved coverage and effectiveness in the verification of real-time graphics systems. The hypothesis was conceptualized into a framework that described the key components required for enabling this type of verification.

Through a holistic experimental case study, where the framework was implemented and used in an industrial project, the value of the developed framework is shown and whether it has any value to the software engineering community. The study was conducted in a fixed design setting, as the projects current testing practices did not change throughout its studied period. During the case study the framework was retrospectively formalized and extended into a set of required tools and techniques. The results of the case study are presented as quantitative data of actual previously known and unknown defects found, together providing a qualitative indication of whether the original hypothesis was accurate.

3

Background

In this chapter, we provide the relevant background information regarding concepts used in the verification framework of this thesis. We start by describing the runtime verification technique that is used for verifying systems during their execution, in Section 3.1, followed by describing various image quality assessment techniques that are based on measuring the similarity between images, in Section 3.2.

3.1 Runtime Verification

Runtime Verification can be used as an alternative or a complement to traditional verification techniques such as model-based checking and testing [15]. Instead of testing individual components, runtime verification offers a way for verifying systems as a whole during their execution [16, 17]. The verification is performed at runtime by monitoring system execution paths and states, checking whether any predefined formal logic rules are being violated [16]. Additionally, runtime verification can be used to verify software in combination with user-based interaction [16], adding more focus toward user specific test-cases, which more likely could uncover end-user experienced defects. However, care should be taken as runtime verification adds an overhead potentially reducing system performance. This overhead could possibly affect the time sensitivity of systems in such way that a system appears to run correctly while the verification is active, but not after it has been removed [17] (a common problem when checking for data-races).

3.2 Image Quality Assessment

Image Quality Assessment is used to assess the quality of images or video-streams based on models simulating the *Human Visual System* (*HVS*) [18]. 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[1] are

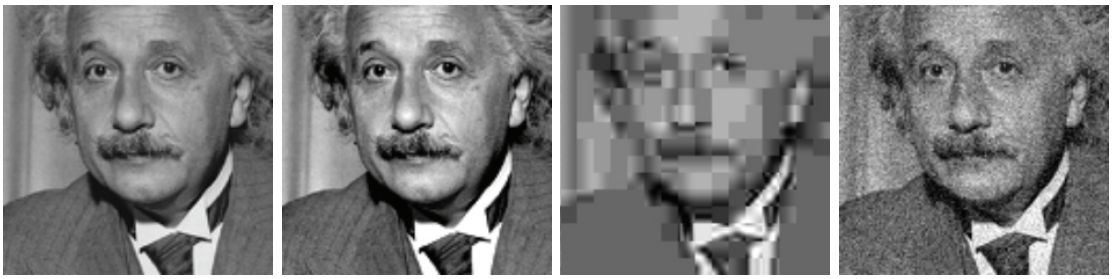
[1] brightness measure

more perceptible than errors in chrominance[2] [19]. 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 it 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 [20]. 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 Figure 3.1, 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). Conducted tests [18] have shown that *SSIM* provides more consistent results compared to *MSE* and *PSNR*. Furthermore, *SSIM* is also used in some high end applications[3] as an alternative to *PSNR*.



(a)	**(b)**	**(c)**	**(d)**
MSE:0 SSIM:1.000	MSE:306 SSIM:0.928	MSE:309 SSIM:0.580	MSE:309 SSIM:0.576

Figure 3.1: *Image Quality Assessment* of distorted images using *MSE* and *SSIM* [18].

[2] color information

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

4

RUGVEF

This chapter outlines the *Runtime Graphics Verification Framework* (*RUGVEF*), which we have developed with the purpose of complementing existing verification techniques that are programmatically incapable of interpreting the graphical states in real-time graphics systems.

This chapter is organized as follows: we start by describing how runtime verification is combined with image quality assessment in order to create a verification process that is capable of verifying graphics related system properties. We then address the issue of synchronizing the verification process with the graphical output of monitored systems in Section 4.2. Finally, we explain how graphical content is analyzed for correctness using image quality assessment in Section 4.3.

4.1 The RUGVEF Conceptual Model

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, checking of execution paths and verification of graphical output, which are used to evaluate temporal and contextual properties of the monitored systems.

To illustrate an example of such properties, we consider a simple video player that takes video files as input and displays them 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.*

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 programmatically replicate. 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.

Nevertheless, there is a limitation in using comparisons for evaluating graphical output. To illustrate this consider the example in Figure 4.1, where we show 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 lies in that rendering speeds usually fluctuate, causing consecutive identical runs to produce different frame by frame outputs. For instance in Figure 4.2, 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.

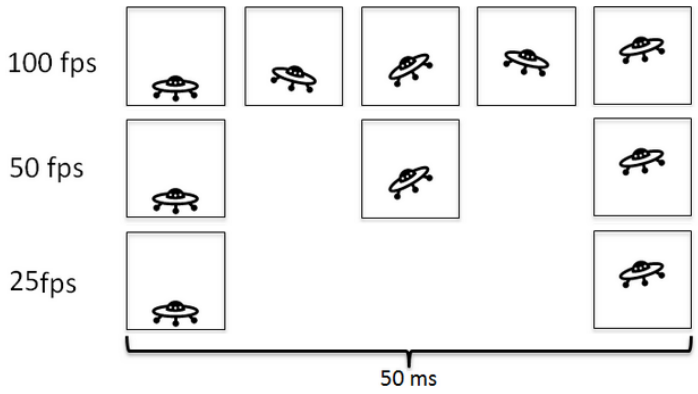


Figure 4.1: The frame-independent rendering technique is used for rendering a moving object at different frame rates, but where the displaced distance during identical periods is the same.

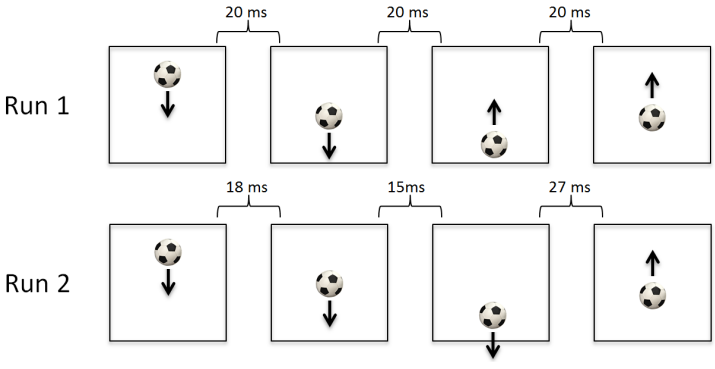


Figure 4.2: Frame-independent rendering produces different results depending on the time elapsed between each rendered frame.

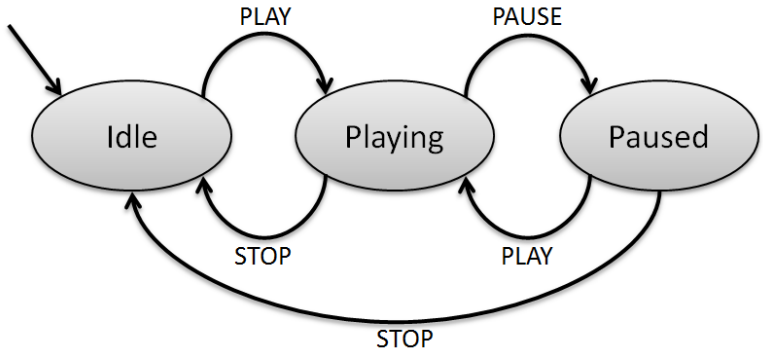


Figure 4.3: The nondeterministic finite state machine formally describing the system characteristics of the simple video player.

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 described in a previously mentioned example. This video player has three system controls (`play`, `pause`, and `stop`) and two system properties which the video player must satisfy. All these system characteristics are represented in the nondeterministic finite state machine shown in Figure 4.3. In this formal definition, transitions are used to describe the consequentiality of valid system occurrences 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 (i.e. `play`, `pause`, and `stop`) that, when executed, trigger the transitions. States (i.e. `idle`, `playing`, and `paused`) collectively 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 Figure 4.4, 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 signify 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.

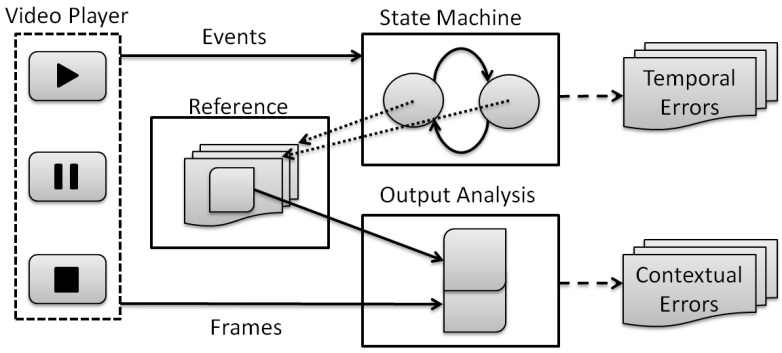


Figure 4.4: An implementation of *RUGVEF* used for automatically monitoring the simple video player's temporal and contextual properties.

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 state machine of the previous example illustrated in Figure 4.3. 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. For instance in Figure 4.5, we show an issued command triggering an event, which is received three frames before the observable output, causing the corresponding reference to be used too early.

In the following section, we propose two possible methods for synchronizing the graphical output with references.

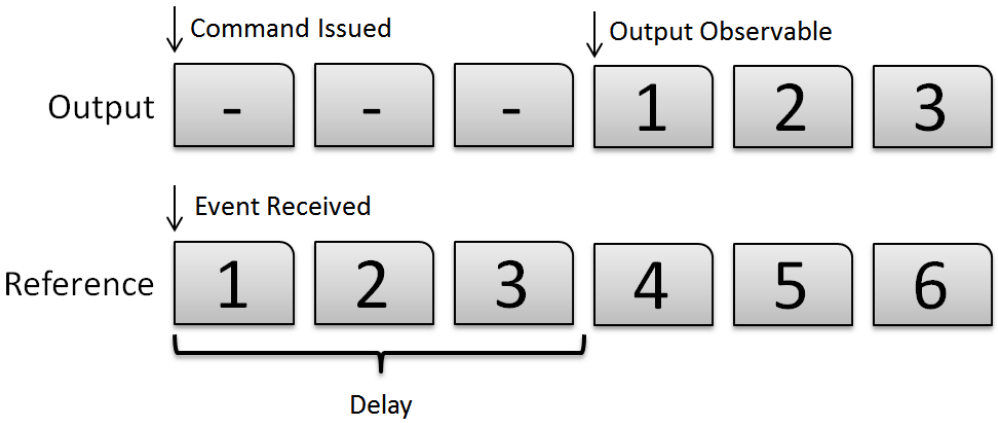


Figure 4.5: The asynchronous relationship between the input and the output causes references and the graphical output to become out of sync.

4.2 Solving the Synchronization Problem

We suggest two approaches for solving the synchronization problem that was described in the previous section. The first apporach 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 Figure 4.6, we show a wrapper intercepting and delaying an event tagged with the 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 Figure 4.7, 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.

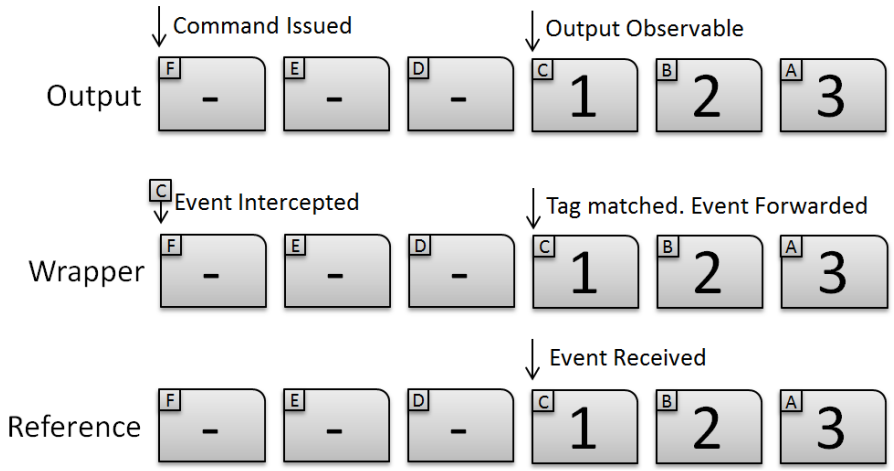


Figure 4.6: Synchronization using a wrapper that delays events until frames with correct tags are received.

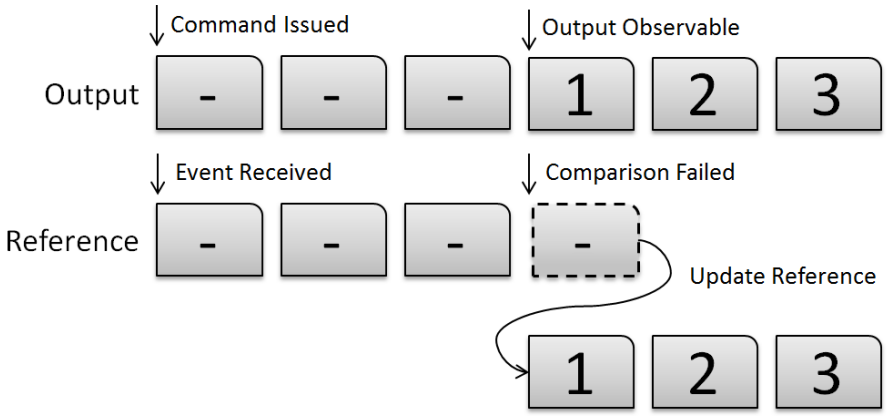


Figure 4.7: Synchronization achieved by delaying the update of references until a mismatch occurs.

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 combinations 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 it will also require 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 the following section, we show how reference based image quality assessment techniques are used in order to determine the correctness of graphical output of systems.

4.3 Using 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 or that separate incompatible arithmetic models are used. However, it is not always the case that non-perceptible dissimilarities are an issue, requiring in such cases that a small tolerance threshold is introduced in order to ignore acceptable differences. One example of this issue could occur when the monitored system generates graphical output using a *GPU*[1] based runtime platform, conforming to the *IEEE 754* floating point model[2], while its reference generator is run on a *x86 CPU* platform, using an optimized version of the same model[3], possibly causing minor differences in what otherwise should be binarily identical outputs.

Perceptual correctness is estimated through algorithms that are 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 [18, 19], with small effects on perceived quality [19], 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 better 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.

[1] Graphical Processing Unit

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

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

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 [18, 21, 22].

5

Case Study - CasparCG

In order to evaluate the feasibility of our conceptual model, a case study was performed in an industrial setting where we created, integrated, and evaluated a verification solution based on *RUGVEF*. The industrial setting used was *CasparCG*, an open-source video and graphics play-out system funded and developed by the *Swedish Broadcasting Corporation* (*SVT*) for on-air graphics, used in most of its national and regional live broadcasts. *CasparCG* is a business critical system, producing content that is daily seen by millions of viewers, and has therefore high requirements in terms of reliability and performance.

This chapter is outlined as follows: we start by describing *CasparCG* in Section 5.1; we then give a description of the existing testing practices used for verifying *CasparCG* at *SVT* in Section 5.2; and finally, we show how the testing of *CasparCG* was improved using *RUGVEF* in Section 5.3.

5.1 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 [23] (see Figure 5.1). 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, thereby reducing development costs. Since then, a lot of development has been done and the public major version, *CasparCG 2.0*, was released in April 2012, after a successful deployment in the new studios of the show *Aktuellt* [24] (see Figure 5.2).

During broadcasts *CasparCG* renders on-air graphics such as bumpers, graphs, news tickers, name signs, and much more. All graphics are rendered in real-time to different video layers (Figure 5.3) that are composed using alpha blending into a single video-stream. The framerate of the stream is regulated by the encoding system used by the broadcasting facility. In Europe the *Phase Alternating Line* (*PAL*) encoding system is used which specifies frame intervals of 20 or 40 milliseconds (i.e. 25 or 50 frames per second). 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. Figure 5.4 shows a pipeline consisting of three stages, where tokens are processed synchronously and travel diagonally between stages, allowing each stage to perform parallel processing on different tokens, increasing the processing throughput (i.e. three tokens per time slot) but at the cost of a two slot delay.

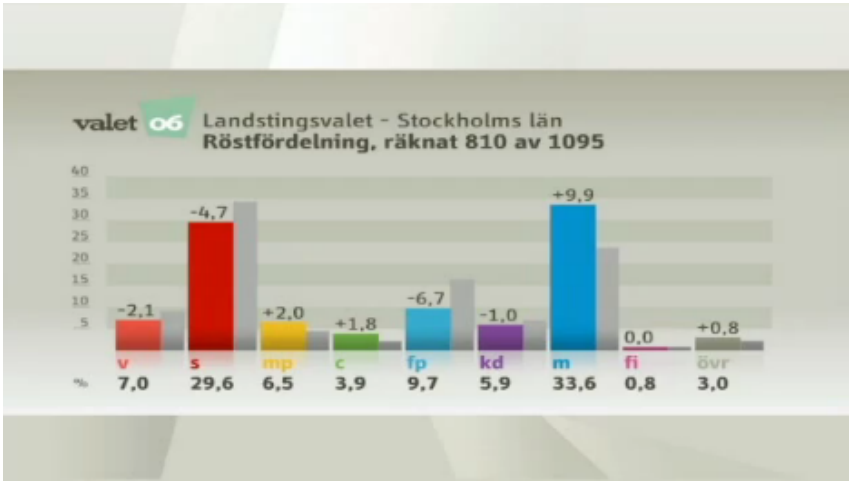


Figure 5.1: On-air graphics rendered by *CasparCG* during the 2006 Swedish elections.

Figure 5.2: On-air graphics rendered by *CasparCG* during the 2012 news show Aktuellt.

Figure 5.3: The output is composed of multiple video layers in *CasparCG*.

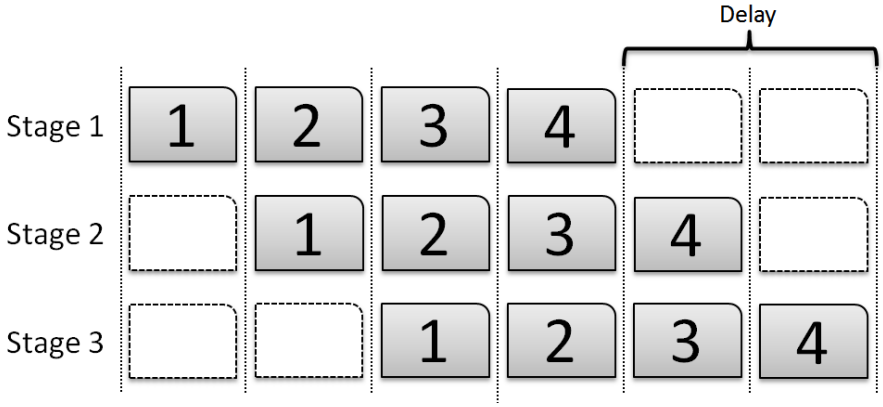


Figure 5.4: Parallel processing stages in a pipeline.

In order to increase the performance reliability of *CasparCG*, the system uses buffers for effectively hiding transient performance drops within its rendering pipeline. For instance, Figure 5.5 shows a buffer with the capacity of three items, but where only two items are enqueued. The buffer is not full because, at some point, its enqueuing stage was unable to keep up, and has not yet caught up with its dequeuing stage; however, this is not externally observable due to the redundancy of items within the buffer. In extreme cases, when a dequeuing stage is starved, the system avoids stalling the pipeline by reusing the most recently dequeued item. This buffering technique further increases the latency between the input and the output of *CasparCG*, which is proportional to the combined size of all buffers within the pipeline.

CasparCG offers a broad range of features allowing it to act as a replacement for several dedicated devices in the broadcasting facility (e.g. video servers, character generators, encoders, and much more). This versatility makes the system a highly critical component where a failure could potentially disrupt several stages within broadcasts.

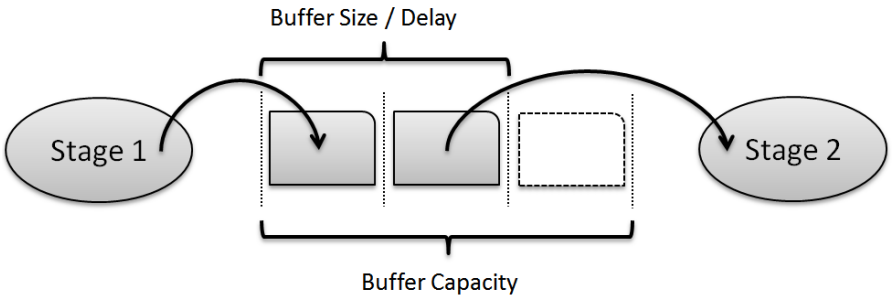


Figure 5.5: Buffering between pipeline stages.

The system is expected to handle computationally heavy operations, e.g. high quality deinterlacing[1] and scaling of high definition videos, during real-time execution. A single program instance can also be used to feed several video-streams to the same or different broadcasting facilities, further adding focus on performance and reliability of *CasparCG*. Quality assurance is, therefore, crucial, requiring rigorous verification processes in order to achieve a high system reliability (i.e. improving $MTBF$[2] and $MTTR$[3]).

In the following section, we describe how verification of *CasparCG* is performed at *SVT* (without using *RUGVEF*) in order to achieve required levels of reliability.

5.2 Current Verification Practices of CasparCG at SVT

CasparCG is incrementally developed and is mainly tested through code reviews and ocular inspections. The 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. 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. In Figure 5.6, we show one verification process of *CasparCG*, where *JMeter* or *Log Repeater* automatically produces input commands while the video output produced is ocularly inspected for correctness.

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

[2] Mean-Time Between Failures.

[3] Mean-Time To Repair.

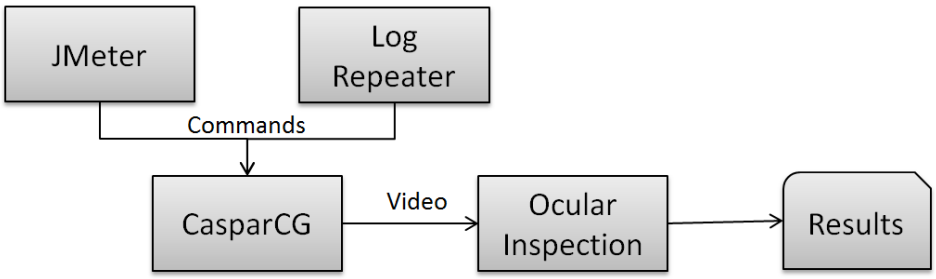


Figure 5.6: *JMeter* and *Log Repeater* are used for automating parts of the testing process.

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.

In our assessment of the testing processes, we have observed the following areas with room for improvement:

- During stability testing, where the system is executed in the background using the tool *JMeter* or *Log Repeater*, the output of the system is only occasionally inspected, causing errors occurring between inspection to possibly remain undetected.
- The ocular inspection process is highly subjective and also limited to the capabilities of the human visual system.
- Regression testing the graphical output is difficult and is, therefore, seldomly applied, limiting the possibilities for refactoring and testing possible fixes.
- Apart from code reviews, public releases have been the main source for bug reports, during the development of *CasparCG 2.0*, indicating a late discovery of defects.
- Debugging is difficult, as there is often inadequate information in order to reproduce the errors that have been reported by users.

In the following section, we describe how *RUGVEF* was implemented and used for improving the verification process of *CasparCG* at *SVT*.

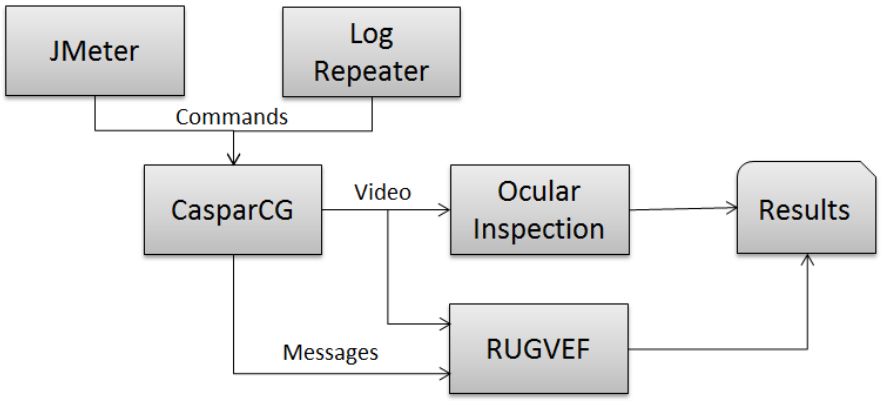


Figure 5.7: *RUGVEF* is integrated into the existing testing workflow of *CasparCG*.

5.3 Verifying CasparCG with RUGVEF

The *RUGVEF* conceptual model 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. Figure 5.7 illustrates *RUGVEF* being non-intrusively added to the testing workflow (intercepting messages and output video transmitted by *CasparCG*), where it is used in parallel with ocular inspections (and possibly other existing practices).

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 on a 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 [25].

5.3.1 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. Figure 5.8 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 that 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

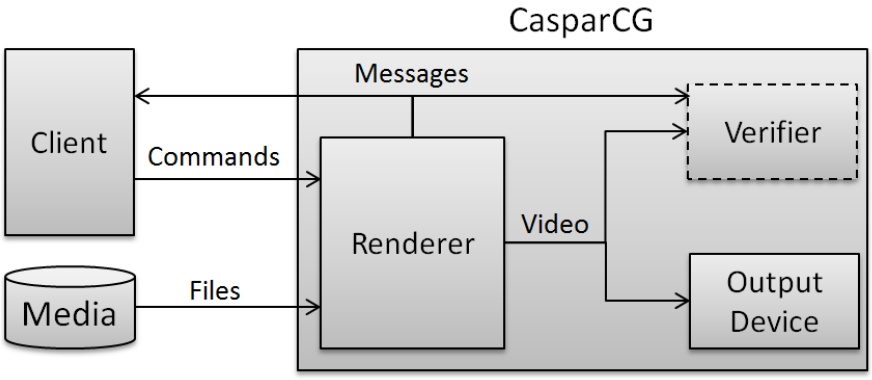


Figure 5.8: The verifier is implemented as an input module, running as a part of *CasparCG*.

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. As an example consider Figure 5.9, showing that a state-machine representing a system with two layers (a) only requires half the amount of states compared to the state-machine representing the same system with three layers (b). 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 solution allows temporal properties of each layer to be monitored separately while the reference implementation is used for checking the contextual properties of the complete system.

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 [26]. Transmitted messages consist of two fields (see Figure 5.10), 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 shown in Figure 5.10; thus, allowing source files to be determined dynamically at runtime.

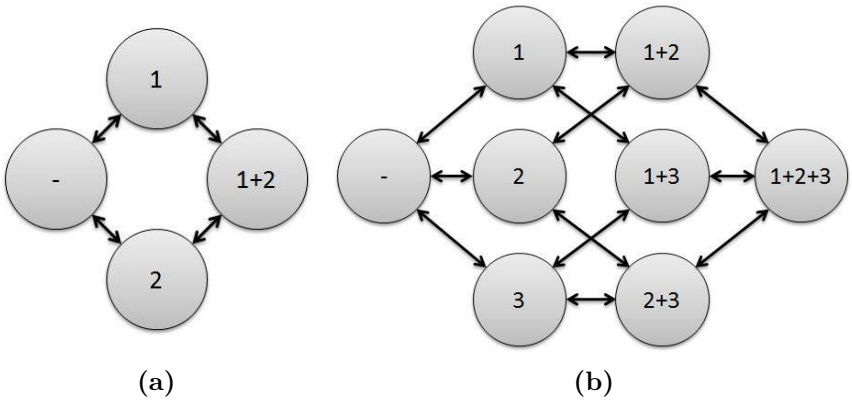


Figure 5.9: Illustrating the problem where the number of states is exponentially increased with each additional layer defined. Showing a state-machine defining a system of two layers (a), and a state-machine defining the same system with three layers (b).

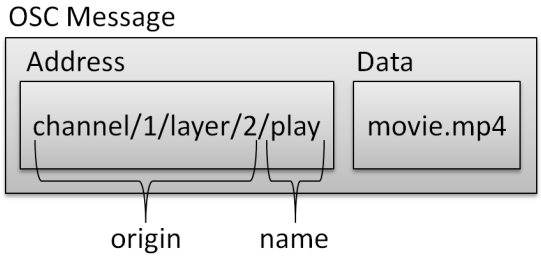


Figure 5.10: The OSC message representing that the file "movie.mp4" is playing on channel 1, layer 2.

We formally define the logic of *CasparCG* through *Extensible Markup Language* (*XML*) scripts (see Listing 5.1). Events are described as regular expressions [27] that are mapped to *OSC* messages and paired with predefined properties. Properties are used for controlling the behavior of the reference implementation, where each property corresponds to a replicated functionality of *CasparCG*. For instance in Listing 5.1, 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`.

As mentioned in Section 5.1, *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 Section 4.2, describing how synchronization is achieved by pairing frames with events, in this case *OSC* messages, using unique identifiers.

```
<?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 5.1: An XML script defining the generic state machine of *CasparCG* where events are paired with properties in order to control the behaviour of the reference implementation.

While evaluating local verification, we found that the process is computationally demanding, possibly affecting the system negatively during periods of high load, making verification inapplicable during stress-testing. Another limitation identified was that all components of the system are not 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). Thus, in order to more accurately monitor *CasparCG*, with minimal overhead and including its physical output, we further extended our implementation of *RUGVEF*, as described in the following subsection.

5.3.2 Remote Verification

During remote verification, the verifier is executed non-intrusively on a physically different system. Figure 5.11 shows this solution, consisting of two *CasparCG* instances running on separate machines, where the first instance receives the commands and produces the output, and where the second instance captures the output and forwards it to the *RUGVEF* verification module.

Remote verification requires that *OSC* messages are transmitted between the machines and where we originally attempted to solve this problem 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

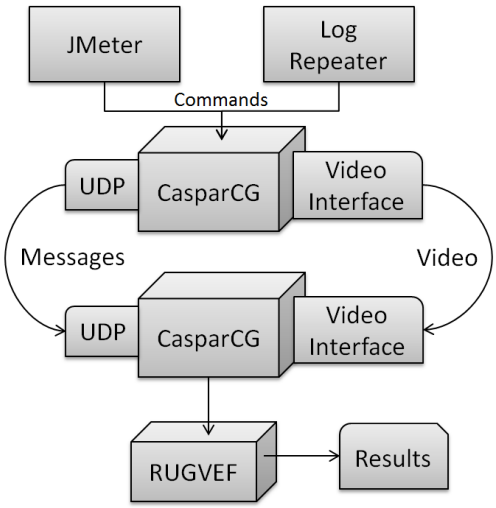


Figure 5.11: The remote verification uses two instances of *CasparCG* that are running on two separate machines, where the first instance renders the video content, and where the second instance captures the rendered content and forwards it to the verification module.

problem that made remote verification impossible (i.e. the opposite of the synchronization problem described in Section 4.1). Thus, we instead use the *UDP* protocol through which we were able to attain sufficiently low transmission latencies.

Additionally, the already 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 also implemented the reference based synchronization scheme that is described in Section 4.2.

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 [4], 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*, which we found to be the best alternative for determining whether two images are perceptually equal.

Nevertheless, the reference *SSIM* implementation [25] 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 the following subsection, we show the algorithmic and implementation specific optimizations performed in order to make *SSIM* applicable to the *RUGVEF* verification process of *CasparCG*.

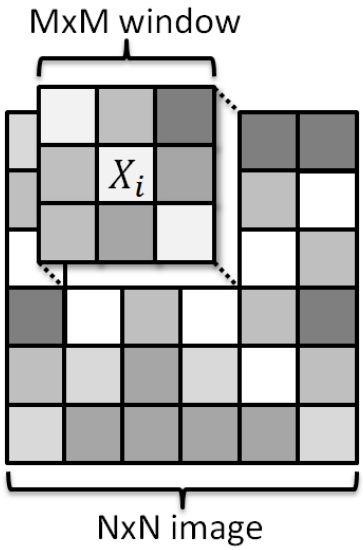


Figure 5.12: The M by M window for the pixel X_i in the N by N image.

5.3.3 On the implementation of SSIM

The main challenge of improving the implementation of *SSIM* consisted of achieving the performance that would allow the algorithm to be minimally intrusive, while keeping up with data rate of *CasparCG*.

In this subsection, 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 [18]:

- *luminance*, the mean pixel intensity between images.
- *contrast*, the variance of pixel intensity between images.
- *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 Figure 5.12. 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 [18]:*

$$SSIM(X,Y) = N^{-2}\sum_{i=1}^{N^2} luminance(X_i,Y_i)\times constrast(X_i,Y_i)\times structure(X_i,Y_i) \quad (5.1)$$

$$luminance(x,y) = (2\mu_x\mu_y + C_1)/(\mu_x^2 + \mu_y^2 + C_1) \quad (5.2)$$

$$constrast(x,y) = (2\sigma_x\sigma_y + C_2)/(\sigma_x^2 + \sigma_y^2 + C_2) \quad (5.3)$$

$$structure(x,y) = (\sigma_{xy} + C_3)/(\sigma_x + \sigma_y + C_3) \quad (5.4)$$

$$E[x] = \mu_x = \sum_{j=1}^{M^2} w_j x_{ij} \quad (5.5)$$

$$\sigma_x = \sqrt{\sum_{j=1}^{M^2} w_j (x_{ij} - \mu_x)^2} \quad (5.6)$$

$$\sigma_{xy} = \sum_{j=1}^{M^2} w_j (x_{ij} - \mu_x)(y_{ij} - \mu_y) \quad (5.7)$$

We found that the main bottlenecks of implementing this specification are Equations (5.5) to (5.7)), each having respectively the time complexity of $O(N^2M^2)$, where $N^2 = 1920 \times 1080$ for *HDTV* resolutions [28] 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*) [29], 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 [18] with M=8, allowing calculations to be evenly mapped to vector sizes of four elements (i.e. two vectors per row).

Specification 2

$$f_{SIMD}(\vec{x},\vec{y}) = \{f(x_1,y_1), f(x_2,y_2), f(x_3,y_3), f(x_4,y_4)\} \quad (5.8)$$

By utilizing *SIMD*, we are able to reduce the time complexity of Equations (5.5) to (5.7)) from $O(N^2M^2)$ to $O(N^2M^2/4)$.

Additionally, we implemented *SSIM* using a cache friendly single pass calculation, allowing Equations (5.5) to (5.7) to be calculated in one pass. This implementation greatly improved the performance, in comparison with the reference implementation [25], where entire images had to be processed in multiple passes and thereby requiring multiple memory loads and stores for each pixel [4].

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 [30]. This parallelization is illustrated in Figure 5.13, 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 available processing units [30], 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 [19]. 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* [28], which we specify below.

Specification 3 *Let* $\vec{x} = \{x_i|1,2,3,4\}$ *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:*

$$Y'_{BT.709}(\vec{x}) = 0.2125x_1 + 0.7154x_2 + 0.0721x_3 \tag{5.9}$$

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. Equation (5.9) can also be realized using a four component dot product[5] between the pixel sample and a weighting vector. Using the dedicated dot product *SIMD* instruction we are able to reduce the time complexity of Equation (5.9) from $O(5N^2)$ to $O(N^2)$. Additionally, by parallelizing the equation,

[4]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.

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

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 we demonstrate in the following chapter.

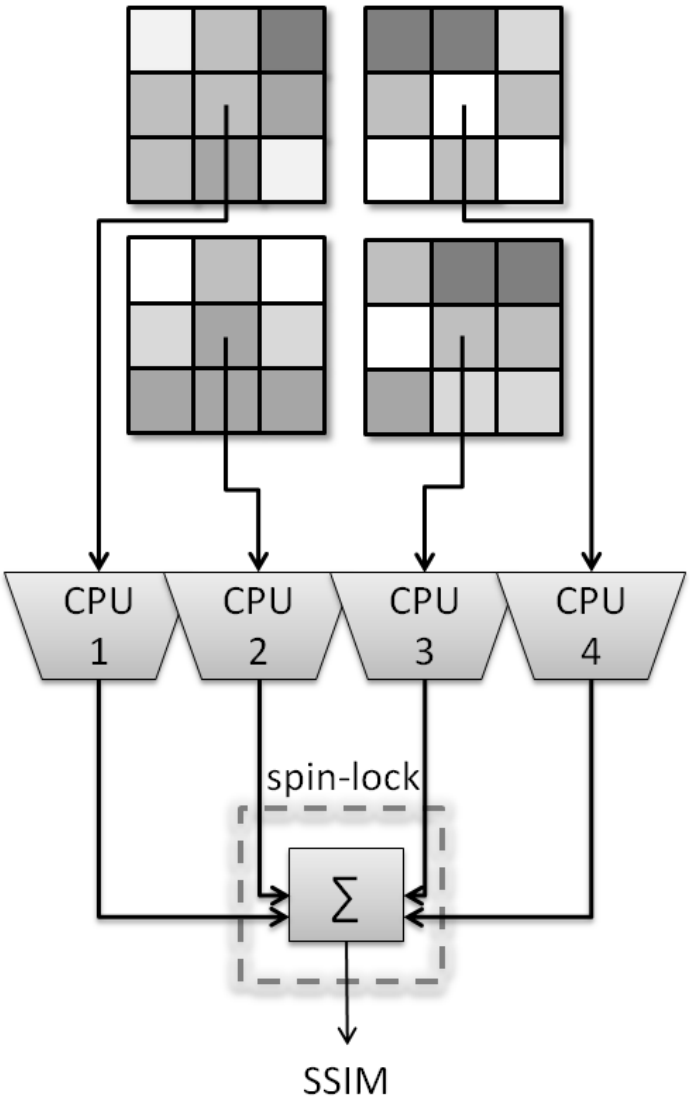


Figure 5.13: Images are divided into partitions which are processed on different central processing units. The results of the partitions are then summed in a critical section using a low contention spinlock.

6

Results

In the previous chapter, we discussed our implementation of *RUGVEF* and its integration into the testing workflow of *CasparCG*. In this chapter, we begin by presenting the quantitative results of new errors found, while verifying *CasparCG* using *RUGVEF*. We then show improvements to the testing process, by checking whether previously known defects (that were injected back into *CasparCG*) could be detected. Finally, we present the improvements in terms of accuracy and performance of our optimized *SSIM* implementation in relation to the reference implementation.

6.1 Previously Unknown Defects

The results produced by *RUGVEF* when errors occur are images describing the errors found. For instance, Figure 6.1 illustrates one example of such images produced when an error was detected, illustrating the reference expected in (a), the actual output in (b), and the pixel errors in (c).

Using *RUGVEF* we were able to find several previously unknown defects which we present in the order of their severity (as determined by the developers of the system). We also discuss the difficulties in detecting these defects through the existing practices at *SVT* (e.g. ocular inspections).

(a) **(b)** **(c)**

Figure 6.1: Three images are produced when errors are detected, where (a) is the reference, (b) the actual output, and (c) the highlighted pixel error.

6.1.1 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). This error is visible in Figure 6.2, 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 Figure 6.2), where differences only are apparent during direct comparisons.

(a) **(b)**

Figure 6.2: The tinting error where the output (b) has a yellowish tint[1]in relation to the reference in (a).

6.1.2 Arithmetic Overflows During Alpha Blending

Using *RUGVEF*, we found that artifacts sometimes occurred in video streams consisting of multiple layers, due to a pixel rounding defect. This defect caused arithmetic overflows during blending operations, producing errors as shown in Figure 6.3 (b) (seen as blue pigmentations[2]). 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.

6.1.3 Invalid Command Execution

Using *RUGVEF*, we found that the software in certain states accepted invalid commands. Figure 6.4 illustrates these errors as dashed lines, showing for instance that it was possible to incorrectly stop idle or non existing layers. Executing commands on non-existing layers caused unnecessary layers to be initialized, consuming resources in the process.

[1]In black and white this is seen as slightly brighter background.
[2]In black and white this is seen as the small grey parts in the white central part of the picture.

(a) (b) (c)

Figure 6.3: Pixel rounding defect causing artifact to appear in image (b) which are not visible in the reference (a). Note that the images have been magnified in order to make the errors more visible.

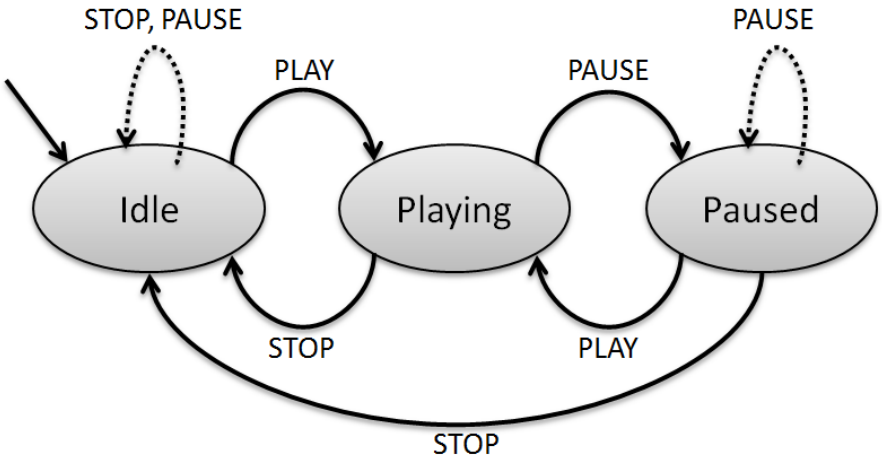


Figure 6.4: A state machine showing parts of the formal definition used during the verification of *CasparCG*, where the invalid transitions detected are shown using dashed lines.

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.

6.1.4 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).

6.1.5 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. Figure 6.5 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).

(a) **(b)** **(c)**

Figure 6.5: The output (b) is perceptually identical to the reference (a) while still containing minor pixels errors (c).

6.2 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 our *RUGVEF* implementation. The injected defects were mined from the subversion log of *CasparCG* [31] by inspecting the last 12 months of development, scoping the large amount of information while still providing enough relevant defects. In Table 6.1, 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 our *RUGVEF* implementation.

Using our *RUGVEF* implementation, 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.

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

6.3 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[3](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 6.2, 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 5.3.3), 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 optimized *SSIM* implementa-

[3]Intel Core i7-2630QM

tion, we are able to analyze the graphical output of *CasparCG* in real-time for *Full HD* streams, which was our goal.

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 Figure 6.6, 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.

(a)	**(b)**	**(c)**	**(d)**
R: 1.000 O: 1.000	R: 0.719 O: 0.714	R: 0.875 O: 0.875	R: 0.699 O: 0.686

Figure 6.6: 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).

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 6.2: The optimized *SSIM* implementation compared against a reference implementation at different video resolutions.

7

Related Work

Although previous research seems scarce, we have found the following work addressing the issues related to the testing of graphics in applications: the tool *Sikuli* [9, 32], that uses screenshots as references for automating testing of *Graphical User Interfaces* (*GUI*s); the tool *PETTool* [10], which (semi-) automates the execution of *GUI* based test-cases through identified common patterns; and a conceptual framework for regression testing graphical applications [11].

We found that the framework presented in [11] was most relevant to our subject. The concept presented is based on verifying graphical applications by combining traditional aspects of testing (e.g. heap integrity and assertion checks) with analysis of graphical output. The basic outline of the verification process is to utilize a harness for executing tests, where tests are registered as function pointers within the harness. The correctness of graphical output is checked through comparisons with previously captured screen contents, which has been subjectively qualified by an operator (done the first time a test is run).

When it comes to verifying graphical output, the framework in [11] uses a similar approach to our proposal in *RUGVEF*. However, the concept differs by focusing 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 also provide a proof of concept in the form of a case study, testing our framework in an industrial setting, while there are no indications that something similar has been done in [11].

Additionally, we also looked at *LARVA* [16, 17, 33], a tool that is based on the conventional runtime verification technique and used for checking runtime properties of *JAVA* applications. Although this tool is not aimed at verifying graphics related system properties, it was used as an inspiration source for developing the *RUGVEF* conceptual model, where we combined runtime verification with image quality assessment techniques.

8

Future Work

We have in this thesis focused mostly on adapting our concept for verifying systems that more or less produce graphical output using already existing video content (e.g. video files). Hence, it would be interesting to investigate whether *RUGVEF* can be adapted to other areas, such the game industry, where output produced is much more dynamic in nature. One idea is to test the possibility of regression testing such systems through recorded references, generated using the existing features of the system, incrementally adding more references as more features are added (similarly to making a movie, where new scenes are recorded continuously).

Another idea, which is more directly related to the work in this thesis, is to use an already developed system as the reference generator while making graphics related optimizations. For instance, reducing the polygon count of objects rendered in games, increasing the performance while checking that changes are perceptually invisible. A different industrial related example could be *CasparCG*, where during our case study of the system, a decision was taken to develop a *CPU* based fallback implementation for much of the *GPU* based functionality. In this case, the current *GPU* based implementation could serve as the reference generator for verifying the *CPU* based implementation during its development, as both are expected to produce identical outputs.

During the implementation of our proof of concept, the capability of transmitting system events already existed in *CasparCG*, allowing us to avoid making changes to the system. However, the existence of such capabilities is ordinarily not the case, instead requiring that the functionality is explicitly added. An interesting idea is to investigate whether it is possible to make *RUGVEF* more generic, e.g. by using function attributes that are automatically translated into code using compiler extensions, thereby reducing the intrusiveness of the framework.

Also, during our case study of *CasparCG*, some audio related errors occurred which are out of scope for the *RUGVEF* verification process. However, it might be possible to detect such errors using perceptual based audio analysis techniques, similarly to our image analysis proposal in *RUGVEF*.

Lastly, we discuss the possibility for improving the reference implementation of *RUGVEF*, created specifically for testing *CasparCG*. This implementation was scoped to only mimic the very basic functionalities (i.e. play, pause, stop, and alpha blending)

which was enough for proving our concept. As a result of this scoping, not all previously known defects could be found while evaluating our proof of concept implementation. Thus, in order to increase the testing coverage, we recommend that the reference implementation is extended so that it supports other basic features of *CasparCG*.

9

Final Discussion

In this thesis, we have investigated the possibility of automating the verification of real-time graphics systems. As a result, we have developed *RUGVEF*, a conceptual framework, combining 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, or 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*.

The results presented show five previously unknown defects detected using our proof of concept implementation. 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 shows that *RUGVEF* can indeed successfully complement existing verification practices by automating the detection of contextual and temporal 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. Additionally, the *RUGVEF* tool created for verifying *CasparCG* was presented at *SVT*, receiving positive responses from the development team.

Bibliography

[1] G. J. Myers and C. Sandler, *The Art of Software Testing.* John Wiley & Sons, second ed., 2004.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[3] D. Galin, *Software Quality Assurance: From Theory to Implementation.* Addison-Wesley, first ed., 2003.

[4] E. Dustin, T. Garrett, and B. Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality.* Addison-Wesley Professional, first ed., 2009.

[5] P. Krill, "Why the time is now for continuous integration in app development," July 2011. `http://www.infoworld.com/d/application-development/why-the-time-now-continuous-integration-in-app-development-941` (5 May 2012).

[6] M. Fowler, "Continuous integration," May 2006. `http://martinfowler.com/articles/continuousIntegration.html` (5 May 2012).

[7] M. Sharke, "Rage PC launch marred by graphics issues," October 2011. `http://pc.gamespy.com/pc/id-tech-5-project/1198334p1.html` (5 May 2012).

[8] J. Carmack, "Quakecon 2011 - John Carmack keynote Q&A," August 2011. `http://www.quakecon.org/2011/08/catch-up-on-quakecon-2011/` (5 May 2012).

[9] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," in *UIST* (A. D. Wilson and F. Guimbretière, eds.), pp. 183–192, ACM, 2009.

[10] M. Cunha, A. C. R. Paiva, H. S. Ferreira, and R. Abreu, "PETTool: A pattern-based GUI testing tool," in *Software Technology and Engineering ICSTE 2010 2nd International Conference on*, vol. 1, pp. 202–206, 2010.

[11] D. Fell, "Testing graphical applications," *Embedded Systems Design*, vol. 14, no. 1, pp. 86–86, 2001.

[12] G. Dodig-Crnkovic, "Scientific methods in computer science," in *Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, p. 126–130, April 2002.

[13] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[14] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, pp. 131–164, April 2009.

[15] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[16] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *FMICS* (D. D. Cofer and A. Fantechi, eds.), vol. 5596 of *Lecture Notes in Computer Science*, pp. 135–149, Springer, 2008.

[17] C. Colombo, G. J. Pace, and G. Schneider, "LARVA — safer monitoring of real-time java programs (tool paper)," in *SEFM* (D. V. Hung and P. Krishnan, eds.), pp. 33–37, IEEE Computer Society, 2009.

[18] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE TRANSACTIONS ON IMAGE PROCESSING*, vol. 13, no. 4, pp. 600–612, 2004.

[19] A. M. Murching and J. W. Woods, "Adaptive subsampling of color images," in *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, vol. 3, pp. 963–966, November 1994.

[20] X. Li, Y. Cui, and Y. Xue, "Towards an automatic parameter-tuning framework for cost optimization on video encoding cloud," *Int. J. Digital Multimedia Broadcasting*, vol. 2012, 2012. Article ID 935724, 11 pages.

[21] Z. Wang and A. Bovik, "Mean squared error: Love it or leave it? a new look at signal fidelity measures," *Signal Processing Magazine, IEEE*, vol. 26, pp. 98–117, January 2009.

[22] S. Winkler and P. Mohandas, "The evolution of video quality measurement: From PSNR to hybrid metrics," *Broadcasting, IEEE Transactions on*, vol. 54, pp. 660–668, September 2008.

[23] S. B. C. (SVT), "Swedish election 2006." `http://www.casparcg.com/case/swedish-election-2006` (5 May 2012).

[24] S. B. C. (SVT), "National news: Aktuellt & Rapport." `http://www.casparcg.com/case/national-news-aktuellt-rapport` (5 May 2012).

[25] T. Distler, "Image quality assessment (IQA) library," 2011. `http://tdistler.com/projects/iqa` (5 May 2012).

[26] M. Wright, "The Open Sound Control 1.0 specification," March 2002. `http://opensoundcontrol.org/spec-1_0` (5 May 2012).

[27] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, second ed., August 2006.

[28] I. C. U. (ICU), "BT.709 : Parameter values for the HDTV standards for production and international programme exchange," April 2002. `http://www.itu.int/rec/R-REC-BT.709/en` (5 May 2012).

[29] Microsoft, "Streaming SIMD Extensions (SSE)," 2012. `http://msdn.microsoft.com/en-us/library/t467de55.aspx` (5 May 2012).

[30] K. Farnham, "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/` (5 May 2012).

[31] CasparCG, "CasparCG subversion repository," 2008. `https://casparcg.svn.sourceforge.net/svnroot/casparcg` (5 May 2012).

[32] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *CHI* (E. D. Mynatt, D. Schoner, G. Fitzpatrick, S. E. Hudson, W. K. Edwards, and T. Rodden, eds.), pp. 1535–1544, ACM, 2010.

[33] C. Colombo, G. J. Pace, and G. Schneider, "Safe runtime verification of real-time properties," in *FORMATS* (J. Ouaknine and F. W. Vaandrager, eds.), vol. 5813 of *Lecture Notes in Computer Science*, pp. 103–117, Springer, 2009.