FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Automated testing framework implementation for post-production and broadcast HW/SW

Miguel Nabuco

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Ramada Paiva Pimenta (PhD)

July 16, 2012

Automated testing framework implementation for post-production and broadcast HW/SW

Miguel Nabuco

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor Raul Fernando de Almeida Moreira Vidal
External Examiner: Doctor José Francisco Creissac Freitas Campos
Supervisor: Doctor Ana Cristina Ramada Paiva Pimenta
July 16, 2012

Abstract

Testing is a fundamental part of software development process. It allows developers to continuously improve their software quality and certify that the final user has a flawless experience using it.

This research work was made in partnership with MOG, a broadcast and post-production company. MOG's products perform several video related tasks, such as file-based ingest and SDI capture. Due to the products complexity, it is necessary to test them thoroughly, in order to validate them for the final customer.

Testing in the company is being made manually. Due to the big amount of variables to be tested, for example video codecs, manual testing provides an incomplete product coverage.

This research work goal is to create an automated test framework, that will test all these combinations of variables, providing a complete coverage.

The framework was conceived in a modular way, so that modules can be easily added or removed in case of new requirements and functions to be tested. The several modules were responsible for generating the tests, organizing them in a specific order, and performing them using the products SOAP interface. There were also modules responsible for setting up the hardware and software, like the signal generator and license manager modules.

The framework was fully integrated with a test modelling tool, testLink, that can create test results reports. This way, it was easier to analyse the results and make decisions concerning the product release.

Overall the goals were achieved successfully. The framework was used not only in complete product testing, but also in other tasks, like specific feature and bug testing. In these cases, the plans only contained the necessary test cases to fix the bug or to validate the feature.

The testing of several products proved the framework efficiency, and it is currently being used to fully test all the company's products.

Resumo

A área de testes é uma parte fundamental no processo de desenvolvimento de *software*. Permite aos programadores melhorar continuamente a qualidade do seu software e certificarem-se que o utilizador final tem uma experiência perfeita ao usá-lo.

Esta dissertação foi feita em parceria com a MOG, uma empresa de pós-produção e *broadcast*. Os produtos da MOG realizam várias tarefas relacionadas com o processamento de vídeo, como *file-based ingest* e captura SDI. Devido à complexidade dos produtos, é necessário testá-los de forma exaustiva, de forma a validá-los para o utilizador final.

O processo de testes na empresa é feito de forma manual. Devido à grande quantidade de variáveis a serem testadas, por exemplo *codecs* de vídeo, a utilização de testes manuais resultam numa cobertura de testes para os produtos incompleta.

O objectivo principal desta dissertação é a criação de uma *framework* de testes automáticos que irá testar todas estas combinações de variáveis, resultando numa cobertura de testes completa.

A framework foi concebida de uma forma modular, de forma a que os módulos possam ser facilmente adicionados ou removidos no caso de surgirem novos requisitos ou funções a testar. Os módulos são responsáveis pela geração de testes, pela sua organização numa ordem especifica e pela sua execução usando a interface SOAP dos produtos. Também há módulos responsáveis pela configuração do *hardware* e *software*, como os módulos do gerador de sinais e gestão de licenças.

A *framework* foi integrada com uma ferramenta de gestão de testes, *testLink*, que consegue criar relatórios com os resultados dos testes. Desta forma ficou mais fácil analisar os resultados e tomar decisões acerca do lançamento dos produtos.

Em geral os objectivos foram atingidos com sucesso. A *framework* foi usada não só no teste completo dos produtos, mas também em outras tarefas como testes de *features* ou *bugs* específicos. Nestes casos, os planos de testes só continham os casos necessários para corrigir o *bug* ou validar a *feature*.

O teste completo de vários produtos provou a eficiência da *framework*, que está actualmente a ser usada para validar todos os produtos da empresa.

Acknowledgements

This dissertation would not be possible without the help and support of several people.

I would like give my appreciation to my supervisor, Ana Paiva, for her guidance throughout the work, and to Augusto Sousa, for his help in several moments during last year and for his constant dedication to his students.

In a thesis made in a company, working in a good team is fundamental. I would like to thank all my co-workers at MOG for their precious help, in particular to Eduardo Espinheira for his constant support and faith in me.

During all these years, my friends were an essential part of my life. I thank them all for their friendship. In particular, I would like to thank Rubén and Vanessa for being my best friends and André, Sofia and Soraia for putting up with me during this semester.

I am only here today because of my family's efforts. During all my life, they have fully supported my decisions. For them goes my greatest appreciation and respect.

Last but not least, a special thanks to Marta for her love, patience, and for always believing in me.

Miguel Nabuco

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

B2B	*Business to Business*
BVA	*Boundary value analysis*
ECP	*Equivalence class partitioning*
GUI	*Graphical user interface*
HD	*High definition*
RC	*Release candidate*
RPC	*Remote procedure call*
SDI	*Serial digital interface*
SIFT	*Scale-invariant feature transform*
SOAP	*Simple Object Access Protocol*
TDD	*Test Driven Development*
TL	*testLink*
V&V	*Verification and Validation*
VCR	*Video Cassette Recording*
XML	*Extensible Markup Language*

Chapter 1

Introduction

In software engineering, it is normal for the programmer to fail, to commit errors leading to the existence of bugs. Sometimes they can be found while the code is being written. Most of the time bugs can only be discovered through software testing.
The cost to correct bugs can be low or high depending on the moment they occur. A research shows [Boe07] that the cost to fix bugs increases throughout a product conception phases, as seen in figure 1.1.

If the software is far into development, when a bug surges more modules or classes have to be changed. In this cases the bug covers a broader part of the program. On the other hand, if a bug is found in the requirements phase, little to no harm is done to the overall project.
It can be seen now that software testing is a fundamental piece of this puzzle and that testing must be done as soon and thorough as possible. Ideally, the testing phase should be running paralleled with all the development phases, starting with the software design.
Testing is not an easy task though; human and time resources needed to achieve 100% test coverage of a product are too high.
This problem cannot be fully solved, but there are some ways to maximize the test coverage or efficiency:

- **Requirements and test priority** - By defining which product requirements are features are more important to the costumer, the tests that cover those feature will be prioritized. In this way, although it doesn't necessarily provide a higher test coverage, it guarantees that the requested features work. The product is then ready for the costumer's workflow.

- **Test automation** - Test Automation saves a lot of time and human effort, while maintaining a high test coverage. By assigning the test execution to a machine,the machine can work uninterrupted to provide the test results. Human resources can be re-allocated to other tasks.

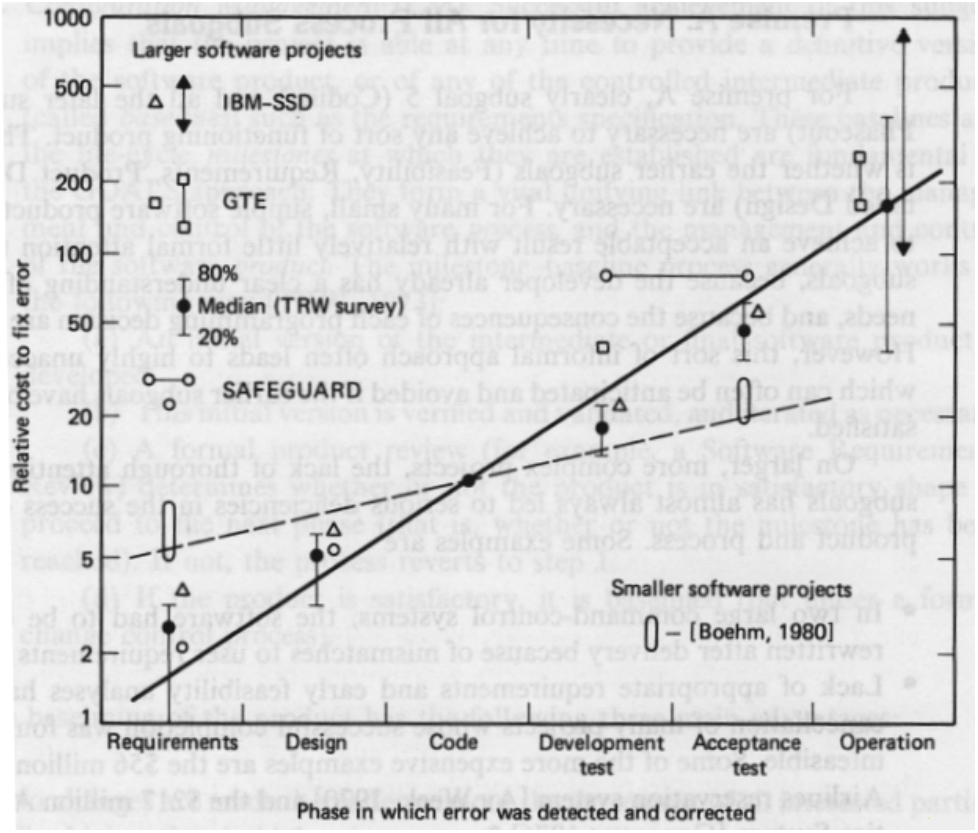


Figure 1.1: Bug fixing cost

1.1 Context

This research work will be made in a business environment meaning that a quick introduction to the company has to be made. This will provide a better understanding of the purpose and the problem the research work proposes to solve.
MOG, Media Objects and Gadgets, is a company that focuses in software & hardware for post-production and broadcast. Its products are sold to professional television networks and service providers. This makes MOG a B2B company. Giving the high cost of the products and the costumers conditions (a television recording/transmitting system has to work 24/7), they have zero tolerance to bugs or errors.
MOG's costumers invest a big amount of money in the company's products and high prices/investments means low costumer tolerance to errors. According to the *Nine Laws of Price Sensitivity & Customer Psychology* [NH02], the **Price-Quality Effect** law explains that buyers are less sensitive to price the more that higher prices signal higher quality. So MOG has to assure their products are of a high quality so that costumers are willing to pay for them.

MOG's products are divided into two main categories:

- **SDI capture** - Used to record, edit and export video in several formats. It receives as an

input a digital signal, from a professional video camera. Then the user interface allows the user to record a clip from the signal, edit the clip information and insert metadata. It also allows basic editing operations, such as cutting and splitting. At the end, the user can export the processed clip to several formats and storage paths simultaneously.

- **File-based capture** - Allow the easy manipulation of video files between removable media and video editing software. This type of products workflow is in some ways similar to the SDI one. It starts with the ingestion of video files from broadcast devices. During this ingestion, the user can edit the video files, as well as preview them, pre-selecting, trim or merging them. After this series of operations is completed, the video files can be exported or published in different formats to storage and web servers.

As it can be seen, video conversion is a fundamental part of the products workflow. There is an enormous amount of possible different formats video files can be converted to in these products. Considering all the possible factors (like framerate, resolution or codec) and the different product series, the amounts of tests to be performed is substantial.

1.2 Problem Description

The high number of tests to be made poses a serious issue to the company. There are only two people available for product validation. With this resources shortage it is impossible to manually test the combinations covering all the cases. In the current testing process, few test cases are made. Usually 4 or 5 different test cases are made to test a feature. To test the correction of a bug, one or two test cases are made.

Due to the lack of time and the fact that the tests are made manually (which means they also take a considerate amount of time), the test coverage is very limited. Considering the previously explained demands of this type of market, it is fundamental that this value increases.

Ideally, the solution would be to increase the test coverage without needing additional resources, both human and temporal. As the tests being considered are many in number, but with few changes between them and always following the same pattern, it is possible to automate them. This can be done by making a computer (or several) control their execution and report their results.

1.3 Goals

Considering the problem described and the proposed solution, the main goal for this research work is the creation of an automated test framework. Being a framework it has to allow the easy adaptation to new and different situations to be tested,and be as generic as possible.
Also, to help the test analysis and report, the framework will have to integrate itself with a test

modelling and management tool.

Since the creation of an automated test framework is a very big and complex objective, it was divided into smaller tasks:

- Study of the environment and the products.
- Study of the test modelling tools and test frameworks
- Definition of a set of rules and good practices for the tests.
- Framework development.
- Implementation of the solution in the company workflow.

The time planning can be seen in the Gantt chart in the picture 1.2.

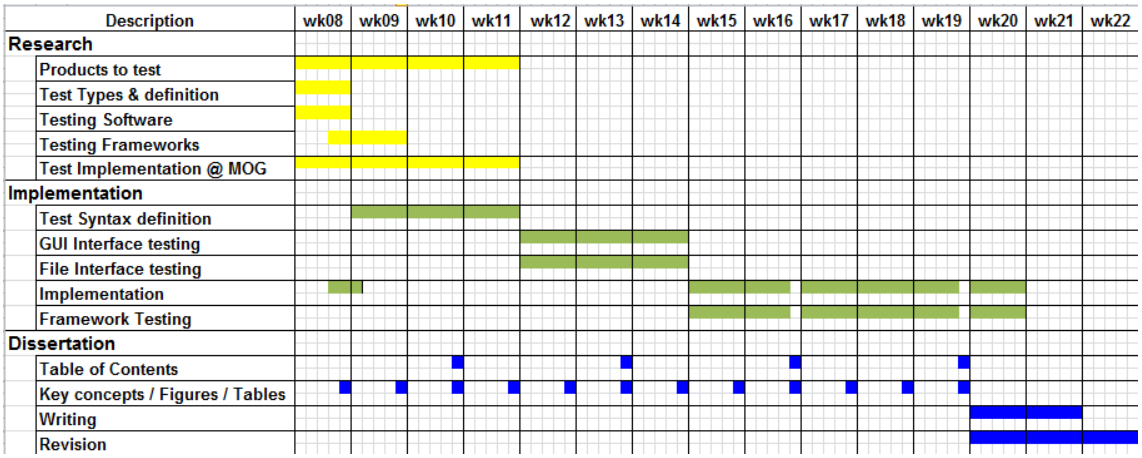


Figure 1.2: Initial Gantt chart

1.4 Motivation

Since this research work is made in a company, the framework to be developed will make an impact in the company's processes, and will test products to be used by real customers.
The area that the company inserts itself, post-production and broadcast, is an area of great interest. It is an area in which the technology is making a fast progress and innovation is constantly present. The possibility of changing the company's test process, making a significant improvement in the validation process, is very tempting. With automatic tests, the time spent testing and the human resources used can be reallocated to other tasks.
This will be a great opportunity to see how the skills, both soft and hard, learned in university can be applied in a professional context.

1.5 Document Structure

This document is divided into five chapters. Chapter 1 makes a quick introduction to the project, presenting the problem to be solved, the context in which the work takes place, the goals to be accomplished and the personal motivation of the student.

Chapter 2 presents the state of art in testing methodologies, risk analysis, as well as research specifically related to this research work, such as test frameworks and test modelling tools. In this chapter it is also explained what kind of testing is made in the company and what methods previously referred can be applied.

In Chapter 3, the implementation of the framework will be described. The chapter is split in small steps, each one describing a specific module or component. After the framework implemented, the results can be seen in Chapter 4. In this chapter, the use of the framework in the company testing workflow will be described.

Finally, in Chapter 5, the conclusions taken from the research work and the implementation are shown. The work to be developed in the future is also described in this chapter.

Chapter 2

State of the Art

In this chapter, some notions about testing will be explained. This area is quite vast so only the most relevant areas will be focused. Besides this explanation, test frameworks and test modelling tools comparisons will be made, to help choosing the most appropriate tools for use in the software development.

2.1 Testing

2.1.1 Definition

The basic purpose of testing in software engineering is to improve a software's quality by finding its bugs. It is critical for any system to have rigorous testing of the features it should have implemented. Also, it is necessary to verify that the implementation is consistent with the specifications [Tra99].

There is no such thing as flawless software, every program contains errors [Mir98]. Tests can help identify most of these errors and indirectly reduce significantly the number of errors in software [TSWI]. It's important to test frequently, so the errors can be found in the initial process of development, reducing the fixing cost.

Each test should only cover a small feature or task [Luc11]. In this way, in case a test fails, it is much easier to identify the source of the error. If a single test case is to cover a full feature or a full set of functions, in the case of failure it is much more difficult to debug. This also makes very difficult to find the precise moment where the program breaks. In software testing it is not enough to test boundary values [Bei95]. Although boundaries or extreme values are areas more prone to errors, it is still necessary to test some regular values. It may happen that the programmer had predicted the use of the boundary values and worked specially around them while neglecting

what can be considered as the "normal" usage of the software.

Test Driven Development is a development technique where the developer first writes a test that fails before he writes new functional code [Amb]. In this method, each feature begins by writing a test for it. To write this test, which will fail in the beginning, the developer can use a scheme of use cases and user stories to better understand the feature's requirements [Bec03]. After the test is completed, the whole battery of tests is run. Then the developer will write the necessary code to pass that test and the battery of tests is executed again. This cycle will be made until all the tests pass. When a new feature is proposed, a new test will be created and this method will repeat itself.

This technique is often linked with **unit testing**. In unit testing, small, simple tests are created to test individual functions or procedures, although they can be used to test entire modules. These tests are usually independent from each others [SMA05]. Most of the programming languages offer tools to design unit tests, such as JUnit, in Java, or PyUnit, in Python.

In the testing process, unit test is only the first phase. After testing these small fragments of code, **integration testing** joins several of these pieces, groups them and tests them as linked modules. The final result of this phase should be a functional system that will be examined for all its final purposes. In this phase, **system testing**, testers will try to find abnormalities and bugs in the system complete workflow [BL01]. At the end, the finished product needs the customers approval. This **user acceptance testing** gives the end users the confidence that the application being delivered to them meets their requirements.

2.1.2 Test Procedures

The process of determining whether the requirements for a system or component are complete and correct and the final system complies with specified requirements is called **verification and validation** [ioeee90].

Verification - Are you building it right? [Boe81]

Verification is intended to check if the software matches the specifications. It is the process of evaluating a system to determine if the features of a development phase satisfy the conditions imposed at the start of that phase [ioeee90]. It can consist of procedures that test the individual parts of a products or even the product in its entirely. In the post-development phase, the tests can be repeated regularly to ensure that the software still meets with the initial requirements and features specification as time progresses and new versions are released.

Validation - Are you building the right thing? [Boe81]

Validation is the process of evaluating a system during or at the end of the development process to determine it satisfies specified requirements [ioeee90]. It ensures that the product meets

the users needs and it performs as the users want.

There are two main test design strategies: **black-box** and **white-box** testing. The two mainly differ in the knowledge that the tester has of the system he wants to test.

Black-box testing

Black-box testing, also known as functional testing, is a software testing technique where the tester does not have the knowledge of the software inner methods or structures [Ltd11]. The tester only knows the inputs and what the expected outputs should be [Bur03]. He never examines the code and does not need further knowledge of the program other than its specifications.

Some of black-box testing advantages are the following:

- **Unbiased tests** - the developer and the tester usually are different members of the team.
- **User point of view** - the test is done from the user point of view, not the developer.
- **Quicker test case development** - testers do not need to spend time identifying the internal paths associated in a specific process, they only concern with the several paths a final user can take in the user interface. Also, test cases can be designed as soon as the specifications are complete.

Some disadvantages can also be found with this method:

- **Incomplete coverage** - testing every possible input path may be an impossible task, or take an inconsiderate amount of time. Due to this, many program paths will remain untested.
- **Script maintenance** - if the user interface is constantly changing, the input may also be changing, forcing the tester to be rewriting the tests several times.
- **Lack of introspection** - since the tester doesn't have knowledge of the system inner workings, it is quite difficult and even impossible to fully test the system.

Equivalence class partition, *ECP*, is a test case design strategy in black-box testing. In this method the input domain data is divided into different equivalence data classes [Hel08]. It is used to reduce the total number of test cases maintaining the effectiveness and achieving the same coverage. It is an attempt to find the most errors with the smallest number of test cases. All test cases are divided into classes and only one test needs to be picked from each class.
For example, testing an input text box that accepts 3 to 8 characters, like a password input box, 3 classes can be created: two with invalid values (smaller than 3 and greater than 8) and one with valid values, between 3 and 8. Any test that you choose from these classes is enough to cover the requirements.

Another test case design strategy is **Boundary value analysis**. There is a greater chance of causing an application error if a user gives as input extreme values [Mat08]. This method can be

combined with *ECP* to select tests at the edges of the equivalence classes.
In a software that receives a number in a certain range as an input, boundary values would be the minimum value, minimum minus one and plus one, maximum value and maximum plus one and minus one.

White-box testing

White-box testing, also known as structural testing, is a software testing technique where explicit knowledge of the internal workings is needed to select the test data [Wil06]. It does not test functionality, but the functions and methods of the program. The test is only accurate if the tester knows what the program is supposed to do. Like black-box testing, this method has some advantages:

- **Introspection** - since testers can identify all the software functions, they can cover a broader range of possible cases in the tests.
- **Thoroughness** - white-box testing allows testers to fully explore every possible internal path and subsequent interactions [Ost02].

And also some drawbacks:

- **Complexity** - in order to test the application, the tester needs detailed knowledge of how the application works.
- **Integration** - opposite to black-box testing, which focus on testing the GUI, testing the internal code, mostly done through unit testing, requires external tools and software. That can raise problems such as compatibility, because some testing tools do not support more than one platform or operating system.

2.1.3 Test Types

Since an application can be composed by several modules and interfaces, it has to be assured that each one is tested correctly. However, as different interfaces have distinct parameters and requirements, there is not one single test model to cover all these features.

Code-driven is a testing type that, as the name suggests, tests code classes and functions with different arguments. The tester, before executing the unit tests, provides the expected results. In the end, to determine if it has passed or not, the actual results are compared with the expected ones.
This type of testing is run constantly during development phase and it guarantees a good code coverage and safer code refactoring [Too].

GUI testing tests a software's graphical interface, therefore providing a more concrete and user-centred way to validate software than code-driven testing. In this method, the user or testing

application simulates mouse clicks, keystrokes, text inputs and other possible user input events in the software to be tested [MSP01]. Menubars, toolbars, buttons and other components that react to user input are verified if they are performing or not in the desired manner.
This method is usually less efficient in terms of time, because the user or the application needs to simulate the test workflow, waiting at each moment for the software to react. For example, if a user clicks on an option he has to wait for the software to react, opening a menu or visually displaying a result.

In the research work context, there are two specific testing types that should also be mentioning, related to image testing, both static and moving (video).

Static image testing focuses in image conversion, in the process of checking if a converted image, to other format or quality, maintains the same base characteristics and looks similar to the original one. This can be done in two ways: manual, visual testing and automatic testing, using image comparison algorithms.
In the manual testing, the tester analyses the two images side by side and measures several points of interest. Depending on the image, these can be characteristics like color accuracy, sharpness or brightness. Due to the lack of precision of this method, image comparison algorithms were developed:

- **Keypoint Matching** - This algorithm detects points in one image with more information than the others. SIFT, scale-invariant feature transform, is used to scan the image and detect interesting points in each object [Low04]. This scan is performed several times over the image to improve the object detection. An evolution of these scans can be seen in figure 2.1. This algorithm applies the same process both to the original and to the converted images and compares the results.
  The image conversion is divided into steps: in each step a new image is created with few changes and stored in database. In the end, the algorithm chooses the image with the closest match to the original. This is a very slow process; the running time is O(n^{2m}), where **n** is the number of keypoints in the image and **m** is the number of different images in database.

Figure 2.1: Sift Keypoints Filtering

- **Histogram** - Histogram is a discrete approximation of stochastic variable distribution. It represents a simple statistic description of an object, an image in this case [Cor01]. This methods consists in, after creating several converted images, building feature histograms for each image, including for the original one.
  After the histograms creation, the absolute value of the difference between each converted image histogram and original image histogram is calculated. The smaller the value is, the better the match, so the images are more similar.
  Different histograms can be used: color histograms, texture, direction, scale. All these different types can be used in the same comparison, creating a much more accurate result. Although this method is faster than keypoint matching, it is unviable if the converted image is scaled, rotated, or has different color based. A histogram can be found in figure 2.3, created from sample figure 2.2.

Figure 2.2: Sample image to histogram

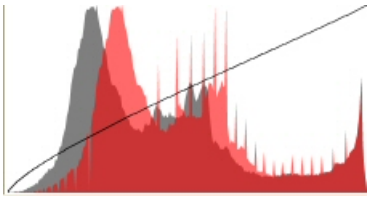


Figure 2.3: Histogram

- **Pixel by Pixel** - This is a very basic and simple method. Given two images, the algorithm compares the color of the pixels with the same coordinates, as seen in figure 2.4. If they are the same, the two images can be considered identical. It is possible to define values of tolerance for pixels location and color range. This means that, even with the conversion altering small image characteristics, the two images may still be considered identical.

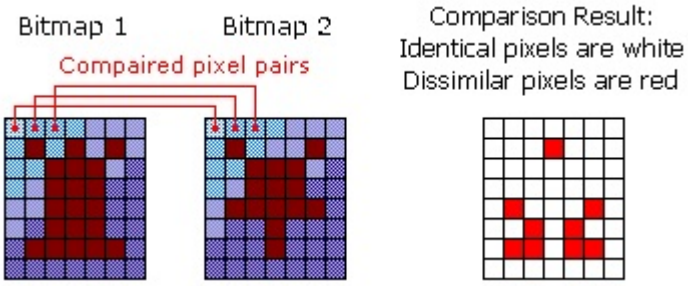


Figure 2.4: Pixel-by-pixel algorithm

Just as the static image testing algorithms, video testing is also focused in video conversion. However, automated video testing is not as evolved as it could be and so most of the video testing

is still being done manually, with the help of expert viewers [Rec].
Video testing consists in testing codecs, video conversion and broadcasting. The results of the tests can be seen either by checking the file size and other parameters such as metadata and video bitrate or by visually verifying the video file to see if the result is the one expected [Vea11]. This second method needs human verification as there is no reliable and efficient method to automatically compare the content of two video files.

To measure the codecs efficiency, several sample video clips are converted to various formats. The use of different sample video clips is important, as some codecs are more appropriate and produce better results for different video types. For example, a codec encoding a video with a lot of movement (a sports event) will have a different efficiency than the same codec encoding an animation movie.

There are two main ways to automatically verify the result of a video conversion:

- The first is checking if the converted video files has errors, is broken or has bad frames. This can be done through command utilities, like *ffmpeg*. This utility can be used for converting video files, as shown in figure 2.5, as well as verifying the integrity of one. It can be seen in figure 2.6 that the video file being analysed has some packages damaged.

```
$ ffmpeg -v 5 -i episode01.avi -f null -
Input #0, avi, from 'episode01.avi':
  Duration: 00:23:52.1, bitrate: 1188 kb/s
  Stream #0.0: Video: mpeg4, 640x480, 29.97 fps
  Stream #0.1: Audio: mp3, 48000 Hz, stereo, 128 kb/s
Output #0, null, to 'pipe:':
  Stream #0.0: Video: rawvideo, yuv420p, 640x480, 29.97 fps, q=2-31, 200 kb/s
  Stream #0.1: Audio: pcm_s16le, 48000 Hz, stereo, 1536 kb/s
Stream mapping:
  Stream #0.0 -> #0.0
  Stream #0.1 -> #0.1
Press [q] to stop encoding
frame=   63 q=0.0 size=       0kB time=2.1 bitrate=   0.0kbits/s
```

Figure 2.5: ffmpeg usage example

```
[msmpeg4 @ 0x68a3c0]ac-tex damaged at 15 10
[msmpeg4 @ 0x68a3c0]
error while decoding block: 15 x 10 (3)
[msmpeg4 @ 0x68a3c0]Error at MB: 425
[msmpeg4 @ 0x68a3c0]concealing errors
```

Figure 2.6: Output of a video error

- An XML or text file can be extracted from a video file to see if its characteristics match to what was expected from the conversion. This method follows the same principle as the unit testing, as the tester provides the expected results and performs the conversion. The results are then parsed from the text file and compared to the expected ones. There are several tools that can fetch that information, like mplayer and hachoir, which usage is shown in figure 2.7.

```
$ hachoir-metadata file.avi
- Duration: 51 min 40 sec 141 ms
- Image width: 624 pixels
- Image height: 352 pixels
- Frame rate: 24.0 fps
- Bit rate: 928.1 Kbit/sec
- Producer: Nandub v1.0rc2
- Comment: Has audio/video index (3.1 MB)
- MIME type: video/x-msvideo
- Endian: Little endian
Video stream:
- Duration: 51 min 40 sec 141 ms
- Image width: 624 pixels
- Image height: 352 pixels
- Bits/pixel: 12
- Compression: XviD MPEG-4 (fourcc:"XviD")
- Frame rate: 24.0 fps
Audio stream:
- Duration: 51 min 40 sec 80 ms
- Channel: stereo
- Sample rate: 48.0 kHz
- Compression rate: 12.6x
- Compression: MPEG Layer 3
- Bit rate: 121.6 Kbit/sec
```

Figure 2.7: Video metadata

2.1.4 Manual/Automatic Tests

In **Manual testing**, a tester is required to play the role of an end user. Test plans are created to simulate a complete workflow. The tester follows that test plan, performing all of its test cases. As manual testing usually cannot be as intensive and as broad as automated testing, it is crucial to select a set of workflows. These workflows must represent the users most frequent and important actions.
The usual methodology can be described as the following set of steps:

- Choosing an appropriate test plan according to resources such as personnel, machines and time.
- Writing detailed test cases, dividing them into clear, short steps and expected results.
- Assigning the test cases to testers. The testers will then perform them manually and record the results.

- Creating a test report based on the results reported by the testers. The test report will be used by programmers to fix the problems found and by product managers to determine if the software release is valid or not.

In **Automated testing**, software is used to control the execution of tests, comparison of results and setting-up the pre-conditions [KH07]. Automated testing has several advantages over manual testing. Since the test execution is controlled by computers the performance will be much faster, even though the test design takes more time in the beginning. This also provides a more efficient and repeatable test environment [DRP99]. Since more tests can be done in the same amount of time, the test coverage will be higher. One tester can only simulate one user at a time, but thousand of users can be simulated and controlled using different threads or execution processes. This is specially useful in stress testing, where the systems are put under heavy load to test their stability and robustness.
However, implementing automated tests is not effortless, and there are some misconceptions about the use of this method:

- **The effort to develop the tests is insignificant/Immediate test effort reduction [DRP99]** - Even though efficient test automation brings clear benefits in long term, a lot of effort and time is needed in the beginning to design the tests. Opposite to the manual tests, automatic ones need to follow a clear and consistent syntax to be parsed and executed by the testing machine.

- **Immediate Reduction in Schedule [Dus01]** - Directly related to the previous misconception, new test tools and entirely new testing processes have to be learned, developed and implemented. In addition to this, it takes time for the testing team to get familiar and to use efficiently this new automated test process.

- **Everything will be automated, no human intervention needed** - Particularly in analysing which tests can be automated, analysing test results and creating reports, human intervention is still fundamental in the testing process.

2.1.5 Risk Analysis

Risk is the probability that a specific problem or issue can occur [Rot01]. The purpose of risk analysis is to identify high-risk applications or features [Per92]. The error-prone components have to be identified and tested more thoroughly.
The objective of risk analysis is to find the most important defects as early as possible at the lowest price [Sch04]. The result of this analysis is used to determine the testing objectives and focus.
Several parameters can be determinant to identify the risk dimensions:

- **Project structure** - the more structured a project is the less risk it contains.

- **Project size** - the larger the project in terms of cost, staff and time, the greater the risk.

- **Experience with technology** - the more experienced the development team is with the technologies being used, less likely it is for the team to commit mistakes. Therefore the risk associated is smaller.

To determine the risk, an analysis has to be made to the project. There are analysis processes with different degrees of formality and complexity.

Judgement and instinct is the most common risk analysis method, while also being the most informal one. It uses the project manager's knowledge and experience with previous projects to estimate the amount of testing required (as well as which areas should receive special attention) to the current project.
This is a repeatable, fast approach that can also contain a good level of accuracy if the manager is experienced enough. However, it is not formally written or explained for others to use, since the manager's experience is not transmittable.

Dollar estimation is a quantification method that calculates the project risk by using dollars as a measure unit. Being a method that is based in discrete values only, it requires a great deal of precision and therefore can be very hard to implement correctly. The measures are calculated based on estimates of frequency of occurrence of a certain issue and loss per occurrence.

Identifying and weighting risk attributes consists in identifying the attributes that cause a risk to occur. Each of these attributes should have an importance associated. The manager uses weighted numerical scores to classify the attributes and, using the average of the attributes general importance and the scores of each attribute in every risk, calculates a final score. This score determines what areas are at most risk and which components should be tested more deeply.

The previous methods can be defined as manual ones. An automated approach to risk analysis can also be used. **Software risk assessment** uses computer software to identify and evaluate controls to manage and reduce risk [Des]. Several types of software can be used for risk assessment: spreadsheets to analyse and compute the weighted scores of the risk data, database software to collect several statistics, financial software to determine the costs. With all the data collected, the system will create an analysis to determine the areas with more risk and will allocate most of the test resources to these high-risk areas.

2.2 Test Frameworks

A **test automation framework** is a set of assumptions, concepts and practices that provide support for automated software testing [Kel03]. It allows the communication with test management tools and the execution of automatic test case batches.

The framework is responsible for:

- Receiving the pre-conditions, steps and expected results from the test management tool.
- Parsing the data and creating a structure readable by the test execution application.
- Calling test application/script to perform the tests.
- Receiving the results.
- Returning the results to the test management tool, where they are processed and grouped, allowing the creation of reports.

There are several framework types, that will now be explained. Some of them do not contemplate the use of test management tools and integrate their functions into the framework:

- **The test script modularity framework** - This type of framework requires the creation of small and independent scripts that represent modules, sections and functions of the application. The scripts are encapsulated in bigger packages that test larger parts of the application. So, each test plan can be divided into layers, and testing starts with the one with lower complexity.

  To better understand the use of this framework an example will be explained. Considering a video recorder, tests will the made to its most basic operations, like playing a clip, or recording one for 10 seconds. For each operation, a script is made and ran. When all the simple tests are completed, bigger wrappers are created, testing more advanced features, while maintaining the simple tests. To test the VCR scheduler feature, that allows the hardware to start a recording at a given point in time, a wrapper is created that contains the scheduler testing function. But to test the scheduler, the recording function also needs to be tested so it is also included in the wrapper. In this way, it is also checked if, during the implementation of more advanced features, some basic feature implementation was changed.
  In this method, if the code that controls some operation is changed, the tester only needs to change that specific operation test. All of the wrappers that use that basic operation will use the updated version.

- **The test library architecture framework** - Very similar to the test script modularity framework, with the same advantages. Instead of dividing the application into scripts, it divides into functions. Each function, or set of functions, is made into a library, like a .dll file. The library files are then called by the test case scripts.

- **The keyword-driven testing framework** - It follows the same principle as manual tests. It requires data tables and keywords to be created manually, representing step-by-step instructions to run the tests. Giving the same example as before, the table 2.1 contains a test case for a video recorder. The table contains, the actions, their arguments and the results these actions are expected to produce.

Table 2.1: Keyword-driven framework data table

Actions	**Arguments**	**Expected Results**
ClickButton	ID=login	
TextEnter	Field user=”user”	
TextEnter	Field pass=”pass”	
ClickButton	Id=OK	Msg:”Welcome user”
ClickButton	Id=storage	
SelectOption	Id= AvidUnityISIS	
ClickButton	Id=Ok	
ClickButton	Id=StartCapture	Msg: “Capture Started”
Wait	10	
ClickButton	Id=Stop	Msg: “Capture Stopped”
CheckValidFile	Filename	True

The developer has to create a parser to interpret the tables and create test cases. One table corresponds to only one test case. As it can be seen, all the navigation data is presented, so the testing application knows which button click to simulate, which options to select. Due to this amount of information presented in the table, the parser can be built in a direct, straight forward way. Less code is written, providing easier test reviews. On the other hand, test design will take up a lot more time, since every little action has to be detailed in the test steps.

- **The data-driven testing framework** - In this type of framework test preconditions, steps and expected results are read from data sources and loaded into variables in manually coded scripts. It is similar to table-driven testing in the way that the test case is also a data file and is not presented directly in the script.
  However, in this method the instructions to be processed (steps to be taken) are coded in the script. Only test data (and not navigation data) is in the data files. Test data can easily be reviewed and used across different scripts. Using the same example, in the table 2.2 it can be seen that the data is presented as functions (which as defined in the test scripts) and their variables, and not detailed instructions to every test.

Table 2.2: Data-driven framework data file

Actions	**Arguments**	**Expected Results**
Login	User=user, Pass=pass	Msg: “Welcome user”
ConfigStorage	AvidUnityISIS	
ConfigRes	1440x1080i	
ConfigFrameRate	60	
Capture	10	Msg: “Capture Started”
Wait	10	
Stop	None	Msg: “Capture Stopped”
CheckValidFile	Filename	True

Contrary to the keyword-driven framework, the test automation environment setup and maintenance will take up more time than the test design. [GG]

2.3 Test Management Tools

When a tester has to create multiple automatic tests, and the changes are minimal between them, he has to have a tool that allows him to create several test copies and allow small changes between them. When he wants to make changes to a test, that tool should allow him to perform those changes without altering the testing script or software.

Test management tool is software used to structure test cases, in a manual or automated form. Since it includes several phases of the testing process, such as describing pre-conditions, determine expected results and generate test reports, it allows the developers to track the tests execution and review the status of the whole project.
Most of these tools integrate directly with various bug, project and requirements tracking systems, allowing a complete software verification and validation workflow.

Using test management tools in software testing allows the project manager to:

- Simplify and speed up the process of testing.
- Organize the work efficiently, allowing an effective distribution of the tests by the developers.
- Keep all the data in one place, allowing to retrieve it easily and compare test results between each version and iteration of the product.
- Check where the developers are working, the status of their work and the resources being used.

Most of the test management tools available are web-based applications that need to be installed in a server, becoming available to intranet users with specific permissions. There is a large number of tools available and to be object of further analysis the most popular options were chosen.
For this project, the tool to be chosen ha to meet some specific criteria: it had to be either free or a low-cost solution and contain support for automated tests, either through built-in support or external plugins.

The tools analysed can be seen in table 2.3.

Table 2.3: Test management tools analysed

Paid	**Free**
Inflectra – SpiraTest	XQual – XStudio
Zephyr – Zephyr	TeamFest – TestLink
QMetry – Qmetry	Mozilla – Testopia

Except for XStudio, which is a desktop application, all the tools are web applications, as mentioned before. A comparison of most of the common features in this type of software can be found in table 2.4.

Table 2.4: Comparison between test management tools

Name	**Req**	**Rep**	**Bug**	**Auto**	**Price**	**Ext**	**Plan**
SpiraTest	X	X	-	-	$500(1)	-	-
Zephyr	X	X	X	X	$800/month	X	X
QMetry	X	X	X	-	$450/month	-	X
Testopia	-	X	X	-	Free/open-source	X	X
TestLink	X	X	-	X	Free/open-source	X	X
XStudio	X	X	X	X	Free	X	X

(1)-support for five concurrent users

Req-requirements manager
Rep-generate reports
Bug-bug tracking system built-in
Auto-support for automated tests
Ext-possibility to create extensions
Plan-Testplans generation

2.4 Tests in the company

In a broadcast environment, due to different customers workflows, there are a lot of different products. Each order contains a specific request, with different features specification. This means that there is not one single product to be tested, and the testing phase is made focusing on each customer more typical and frequent workflows.
For the S Series, the SDI capture products which will be the focus of this research work, a feature consists in a specific recording process: having one input type of signal and a series of operations (record, insertion of metadata, defining storage) creating an output file with the appropriate format and specifications.
To test each feature automatically, tests of different types were defined:

- **Login tests**

- **Storage tests** - test the configuration of a server where the recorded files will be stored.
- **Profile configuration tests** - in this tests, the complete set of options for the capture is defined. This includes basic definitions, like resolution, framerate, codec or number of audio channels, as well as advanced and optional data, such as metadata information, and clipnaming profile definition.
- **Capture tests** - test the recording of a video file with the configuration defined in the correspondent profile. With the capture tests, a basic workflow is finished.
- **Extra tests** - test the additional features of the product, like scheduler capture (start automatically capturing at a certain time) and gang control (controlling several products at the same time).

2.4.1 Equivalent Class Partition

Considering the types of tests explained before, the login, storage and capture tests have no need for class definition. Login test is a single, simple test, storage tests consist in the creation of six different storages each one different, therefore all the six tests need to be different and the capture tests only have as a parameter the profile.
Classes are needed in choosing which profiles to test, and defining how many different combinations are to be tested. There are five parameters that globally define a profile, and each combination of these five values origins a different profile. The parameters are:

- **Codec** - the video codec which will encode the video.
- **Wrapper** - the container for the video file, like .mov, for quicktime files.
- **Resolution** - the video resolution, PAL for Europe, NTSC for Asia/America and several others that can be high or low definition.
- **Framerate** - number of frames per second.
- **Audio channels** - number of audio channels that the video will contain.

Not every combination of these values is valid. A NTSC video cannot have a framerate value of 25. Combining all the possible values, and considering one value for the number of audio channels, the number of possible profiles is 1408.
Since the goal is to fully validate a series of products, all the different class values have the same importance. The only variable value is the number of different audio channel options. Since there are some boards that only support a maximum of 8 audio channels, a minimum of two different options have to be chosen.
To include also the most used values by the customers and the most error prone ones (one or zero audio channels), a total of six different options were chosen: 0,1,2,4,8,12 and 16. This brings the total profile configuration tests number to 9856. For each profile test, there is a capture test that

uses the profile to record a video file.
The total tests to fully validate the complete product with all the codec licenses installed is 19,719.

2.4.2 Risk analysis

Risk analysis is not fully implemented in the company's testing process. The method used to determine the areas with more risk is **Judgement and Instinct**.
The product quality process owner determines the amount of tests to be performed and the importance each one of them has. The features with highest priority to test are the ones most requested by the customers and the ones more prone to bugs/errors. No written formal documentation is used to determine these features.

2.5 Conclusions

In this section, some conclusions will be taken from the state of the art chapter. The choices made based on the information researched will be shown here.

The testing processes in focus are system testing and user acceptance testing.
Since the testers have no knowledge of the products source code or inner structures, the testing technique used is black-box testing.

2.5.1 Test Framework

As it is being talked about frameworks and not final software, the solutions can be both combined or adapted. To choose the best option to use in this research work, the real context and environment must be considered.
Since the tests are being created and described in a test management tool, the framework to be developed should not contain any test data. Taking this into consideration, the **test script modularity** and **test library architecture** framework are automatically discarded because they require that all the inputs, navigation data and outputs to be coded in either scripts or libraries.

As for the **keyword-driven testing framework**, all the three types of data are provided (input, navigation data and output) in a detailed manner. This can be very useful for GUI tests, where it is needed to explicitly explain which buttons to press or what to put in each text field. It not useful though for command-line or SOAP interface tests, because they are based in web commands or packages that call functions from the SOAP servers embedded in the products.
In the **data-driven testing framework** however, there is no need to have the steps as detailed as in the keyword-driven, making it more appropriate to the SOAP tests, but not sufficient for the GUI tests.

It can be concluded that the framework will use these two last kinds, depending on the type of test. Each test case will have a tag in its description informing if it is a GUI or SOAP test. Depending on this tag, the framework will redirect the test to a different internal module to be processed.

2.5.2 Test Management Tools

Every solution presented the same basic features, such as requirements management, report generator and users permission control (except Testopia). The differences found between them were gimmicks, extra features non essential to the project, such as dashboard widgets and team management utilities.

As expected, the paid solutions have more features and are more efficient. However, the difference is not substantial and the price asked for these solutions does not pay up. Therefore, in this comparison only the free alternatives remained as possible choices.

Testopia is a very basic piece of software that does not have one of the main required features, support for test automation. Between testLink and XStudio, the choice fell upon testLink. Even though Xstudio is a more complete solution, it has some drawbacks: it is closed-source and its interface is not as intuitive as testLink's, which has all the necessary features, and it is designed in a simple, clean manner. Also, the company is already using testLink for manual tests and knows how it works. Migration to XStudio would not be compensatory.

2.5.3 TestLink

To start creating and manipulating tests in testLink, an analysis has to be made to understand how the software works, how are projects, test plans and test cases connected and what operations can a user perform with them.

testLink data is represented in the same way as a graph. The structure represented in figure 2.8 shows that the head nodes correspond to the projects and its children nodes to the test suites. The test cases and steps for each test follow the same base structure. Each mode has an unique ID and a parent ID that links to the parent node, along with other attributes specific to the node type.

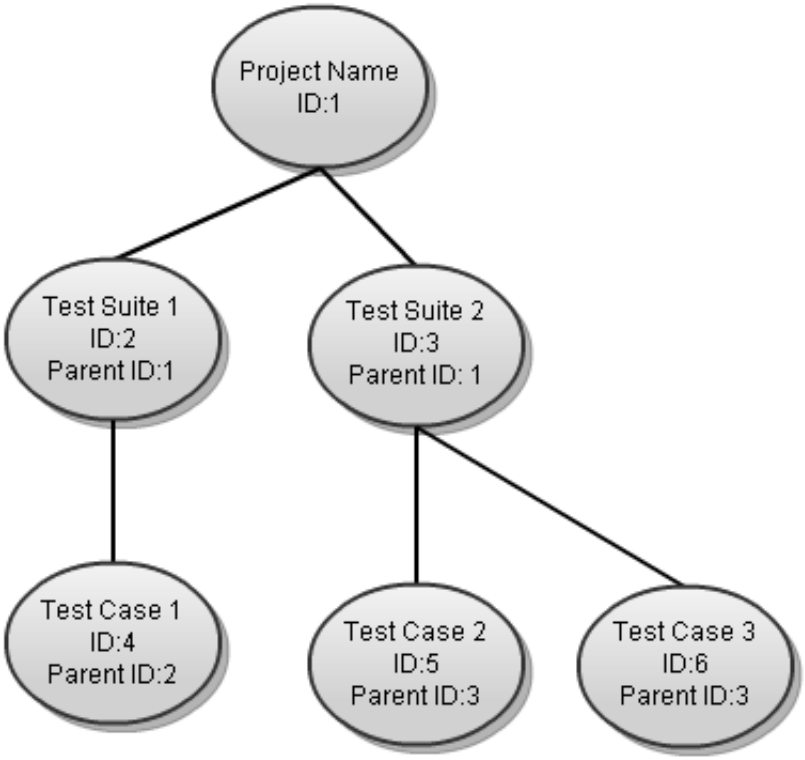


Figure 2.8: TestLink node structure

A sample of a typical workflow in a testing process using testLink can be seen in figure 2.9.

When a test case is created, it cannot be immediately executed and its results reported. Since a test case can be executed several times in different products and configurations, the results of one execution cannot be properties of the test itself. If it would be like that, it would be very hard for testers to see which configuration corresponded to the latest execution and test results.

To bypass this issue, the test case has to be assigned to a test plan to be executed. A test plan is not more than a set of tests to be performed. The results from the test execution are not properties of the test alone, but shared with the test plan and stored in the database as *executions*. Figure 2.10 contains a fragment of the database that stores this information.

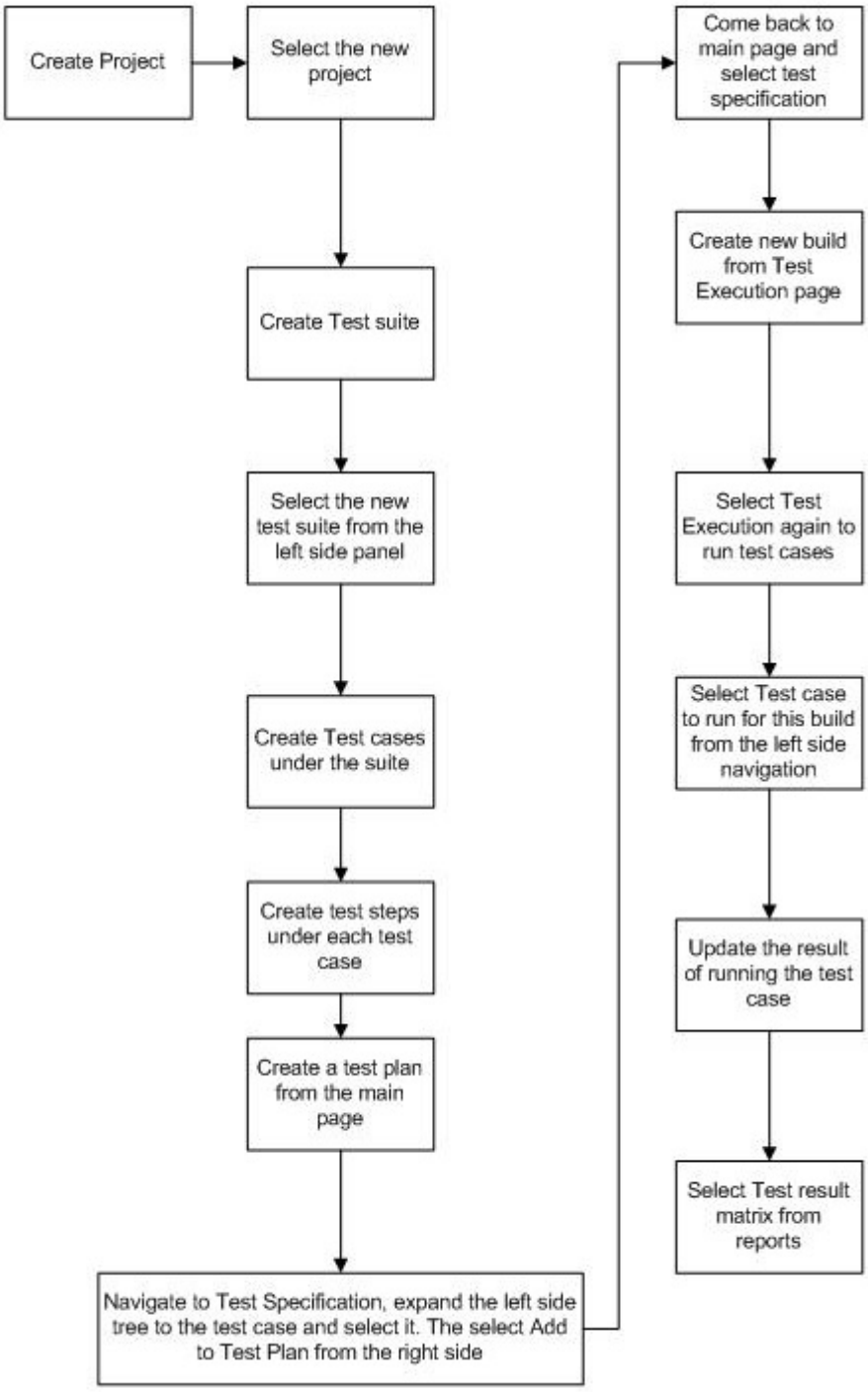


Figure 2.9: TestLink workflow sample (from testLink documentation)

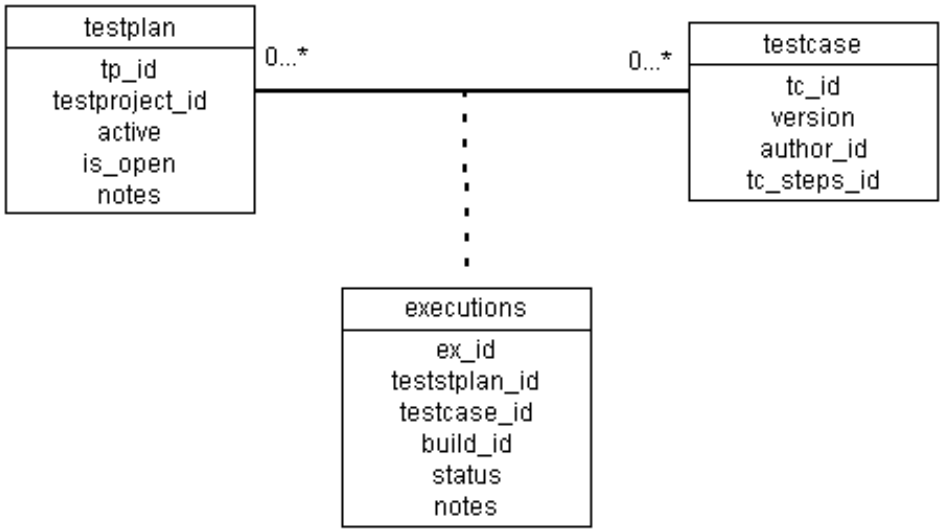


Figure 2.10: TestLink testplan-testcase relation

Chapter 3

Implementation

3.1 Testing process workflow

3.1.1 Initial Workflow

The company testing process can be divided into two main areas: verification and validation tests. Verification tests are made by the developers themselves. This research work will cover a part of the validation tests, that have as main concern the costumer satisfaction with the product and its features.
The current validation testing workflow is represented in the sequence diagram 3.1.

In this workflow, the product validation process owner can also act as a tester. This begins with him receiving the product release candidate, RC, analysing the features and assigning them to all the testers. The testers then go through all the testing process: they design their tests, perform them and update the results and report bugs. The product validation process owner will then gather all this information from the test management tool and the issue tracker. Based on the data analysed, he will decide if the release is viable or not. In this workflow each tester design his own tests, which can cause some incoherence, specially with similar tests assigned to different testers.
With the implementation of framework, the workflow will suffer some changes, as shown in the sequence diagram 3.2.

3.1.2 Workflow to implement

The beginning of this workflow cycle begins one week earlier than the previous one. It doesn't start with the RC release but with the RC planning meeting, where the features are announced. In this way, the tests are being designed while the RC is being developed and not after its conclusion. Based on the features proposed, the product validation process owner inserts the data into the test generator. The test generator will automatically generate the necessary tests and insert them into testLink (TL). The product validation process owner will then assign in TL the automatic tests to

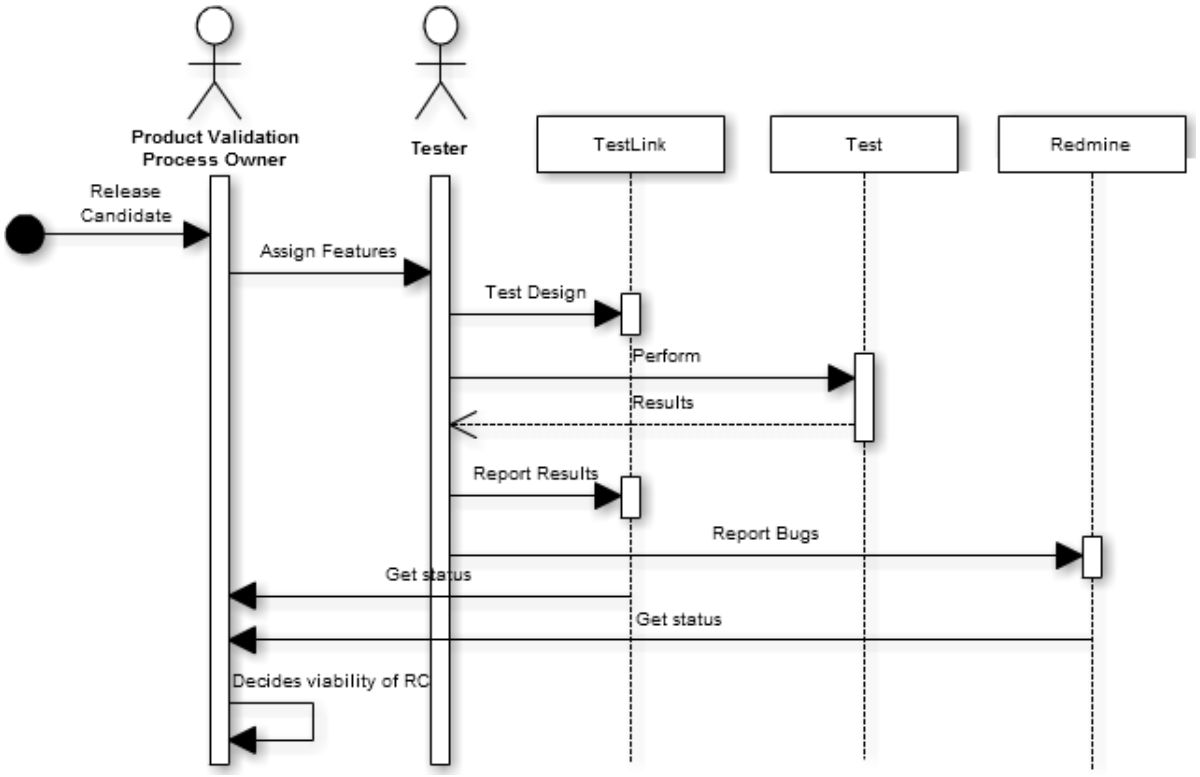


Figure 3.1: Testing Workflow sequence diagram

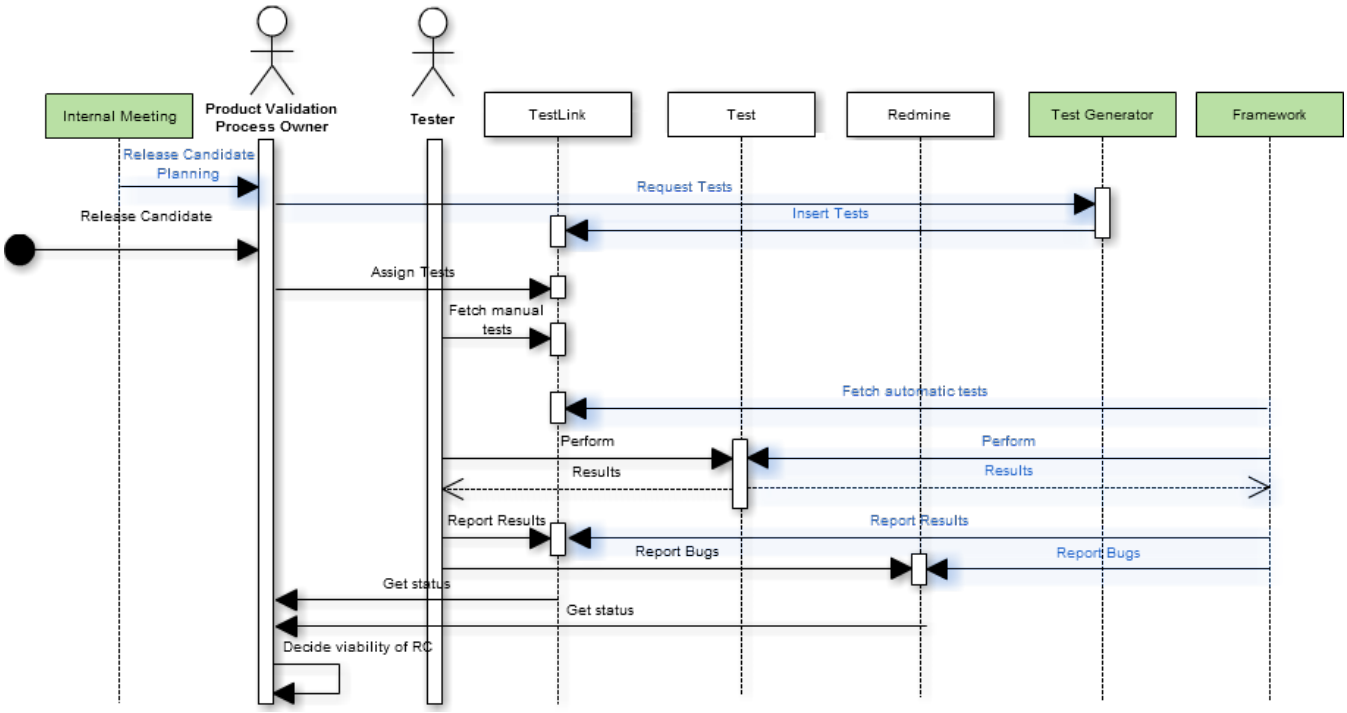


Figure 3.2: Testing Workflow to be implemented

the framework and the manual tests to different testers.
From this step, the workflow is very similar to the previous one, except for the inclusion of the framework that will now fetch, perform and report the results of the automatic tests.

In this process, to fetch the information from testLink, TL, the framework will use the XML-RPC library already existent in TL's code. The library will communicate with the client implemented in Python. The use of XML-RPC library allows the transmission of data between TL and an external program coded in another language. The use of these technologies and their relation is graphically represented in figure 3.3.

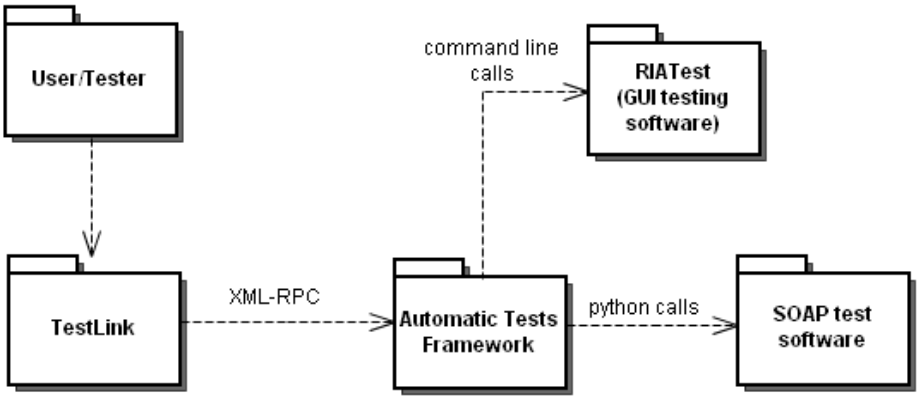


Figure 3.3: Package diagram of the new testing process

The planned inner workflow of the framework can be seen in the activity diagram from figure 3.4.

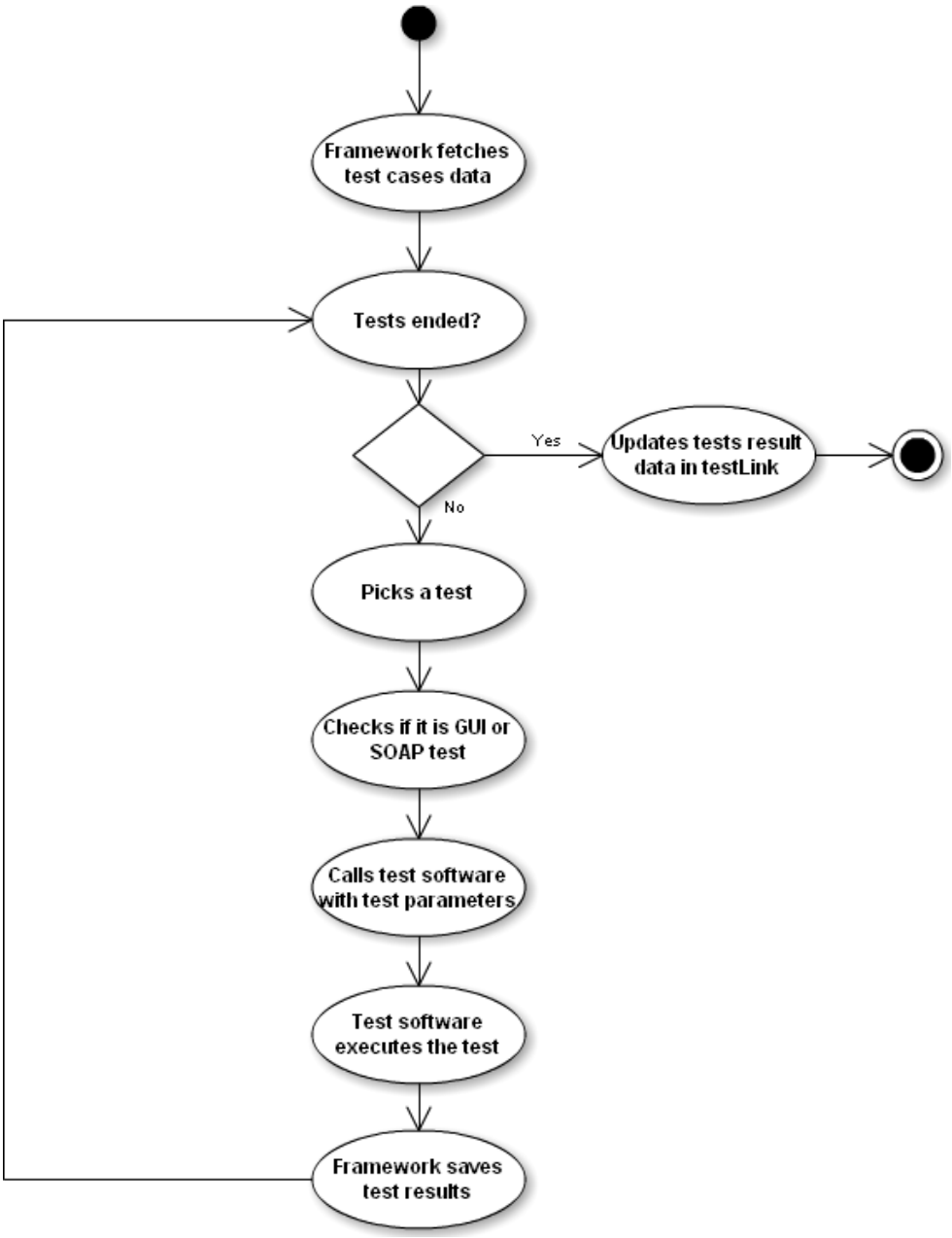


Figure 3.4: Activity diagram of the new testing process

The test structure has to contain some parameter that distinguishes GUI from SOAP tests, so that the respective module can be called.

The graphical interface of the products to be tested is written in Adobe FLEX, using ActionScript programming language. The most used and practical tool to tests FLEX applications is RIATest.
RIATest is a test automation tool that only tests Adobe FLEX interfaces. It can automatically inspect the application GUI elements and simulate mouse clicks in the interface and other types of

user interaction, like typing text into a text box [RIA]. When RIATest finishes the tests execution, it exports the results to a XML file. This file will be inspected by the framework to process these results.
As seen in the package diagram from the figure 3.3, the framework will call RIATest through the command line, passing as parameters the test names and respective parameters.

SOAP interface tests will consist in sending SOAP requests from a network library in Python to the products. The products will receive the requests and reply using the same protocol. The framework will then parse the data received and form test execution results.

All the modules from the framework and testLink, represented in figure 3.5, will be fully integrated so that minimal to no human intervention will be necessary to perform the test plans and validate the products.

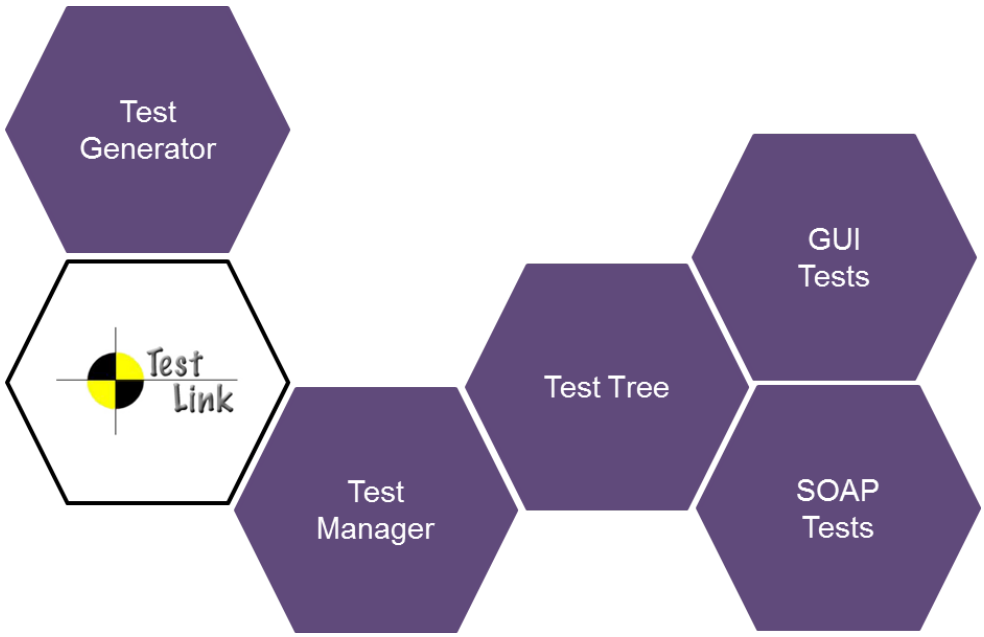


Figure 3.5: Framework modules

3.2 Test Design

3.2.1 Test generator module

Problem

The company has a tool that generates all the video capture profile combinations to insert into the products. To generate a profile, several lists of parameters have to be combined. These parameters include resolution, codec, storage server and others. Each combination can correspond to one or more test cases, depending if there is a need to test not only the capture but extra functions

like insertion of metadata.
All these profiles have to be tested and it would be a massive amount of work manually inserting all the test cases into TL.

Solution

Based on the lists combinations, created using single lined/column matrix multiplication, test cases can be automatically generated. Some combinations are not valid: for example, resolution NTSC cannot have codec IMX30. These combinations are discarded.
An extra script was created in Python to transform these combinations in a format recognized as a test in TL.

3.2.2 TestLink importing module

Problem

With the tests in TL specific format, including test steps, expected results and pre-conditions, it is necessary to create an effective way of importing the data into TL and create a structure of test suites and test cases.
There are three possible ways of import the data: XML files, Python and XML-RPC and database injection.

- **XML** - The script could generate XML files containing the testcases. With this method, the files have to be manually imported into TL. It is a relatively fast method with few test cases (takes 5 seconds to load a file with 1500 basic testcases) and it doesn't require a constant connection to TL for much time. However, TL has some issues with the size of the files and amount of data to be imported. Issues have been discovered when trying to import files with over 1.4 MB, so in most cases the test cases have to be split for many files. Since import of multiple files at a time is not supported and there is a big amount of test cases to be inserted, many XML files have to be manually imported, making it a particularly tedious task.

- **Python and XML-RPC** - The second option is to use TL XML-RPC functions to create test cases and test suites. The script that generates the tests would send them directly using these procedures. Although this is a fully automatic process, without any need for human intervention, it requires a constant connection between the TL server and the python client. During tests with this method, a connection drop-out happened several times, forcing the user to start the whole process again. This method is also very time consuming: sending a project with 9828 test cases divided into several hundred test suites took 1921, more than 32 minutes.

- **Database injection** - The last method is to do database injection to TL's server. This method was quickly discarded because sending SQL commands with the tests to the database is dangerous due to the considerable complexity of the database. There are many confusing

links and connections between the data tables and no proper documentation that explains them.

Solution

To guarantee a relatively fast and secure transmission of data, the most valid method turned out to be the creation of XML files. Although it includes a manual step in a thought fully automatic process, it is the only method that guarantees the complete transmission of data (no data loss). An example of a capture test in XML format can be seen in figure 3.6. Figure 3.7 shows a profile configuration test already imported in TL.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="SOAP S1000 EC">
<testsuite name="Avid OPAtom">
<testsuite name="IMX 30">
<testsuite name="PAL">
<testsuite name="25">
<testsuite name="0">

<testcase name="TCAP:0:PAL 25-AvidGenericStorage-IMX 30-0">
        <summary>
            <![CDATA[<p>[SOAP]</p><p>PAL 25-AvidGenericStorage-IMX 30</p>]]>
        </summary>
        <preconditions>
            <![CDATA[
            <p>[TCFG:0]</p>]]>
        </preconditions>
        <steps><step><step_number><![CDATA[1]]></step_number>
                        <actions><![CDATA[Select profile PAL 25-AvidGenericStorage-IMX 30-0]]></actions>
                        <expectedresults><![CDATA[Status=ready]]></expectedresults>
                        </step><step><step_number><![CDATA[2]]></step_number>
                        <actions><![CDATA[Start Capture]]></actions>
                        <expectedresults></expectedresults>
                        </step><step><step_number><![CDATA[3]]></step_number>
                        <actions><![CDATA[Wait 10 seconds]]></actions>
                        <expectedresults></expectedresults>
                        </step><step><step_number><![CDATA[4]]></step_number>
                        <actions><![CDATA[Stop capture]]></actions>
                        <expectedresults><![CDATA[pass]]></expectedresults>
                        </step></steps>
```

Figure 3.6: TestLink test case in XML format

Version 1

Created on 09/04/2012 11:01:31 by mnabuco

Summary

[SOAP]

Config PAL 25-AvidGenericStorage-IMX 30

Preconditions

[Test:TSTR:AvidGenericStorage]

#	Step actions	Expected Results
1	[Resolution:PAL:0]	
2	[Framerate:25]	
3	[AudioChannels:0]	
4	[Codec:IMX 30:300]	
5	[Wrapper:Avid OPAtom]	
6	[Storage:AvidGenericStorage:avid_standalone:\media\AUTO\StandAlone]	[Status:ready]

Figure 3.7: Test case as presented in testLink

3.2.3 Test Structure and Hierarchy

Problem

As explained before, the tests in focus in this research work consist in testing features. To perform them, a number of steps have to be made: login, configuration of storage, configuration of capture profile and the video capture.
Due to the large number of tests, there is the need to organize them. When a tester needs to create a test plan, searching through a large number of tests is extremely hard if they are not divided by some criteria.

Solution(First attempt)

It was decided to divide tests into folders. Explaining the folder structure as a tree, the resolution and framerate correspond to the top level and storage to the second one, as seen in figure 3.8. Each test would contain login, storage configuration, profile configuration and recording as steps and its name would be the sum of its features, resolution-framerate-storage-codec. In this way, the user knows what is he going to test just by seeing the name.

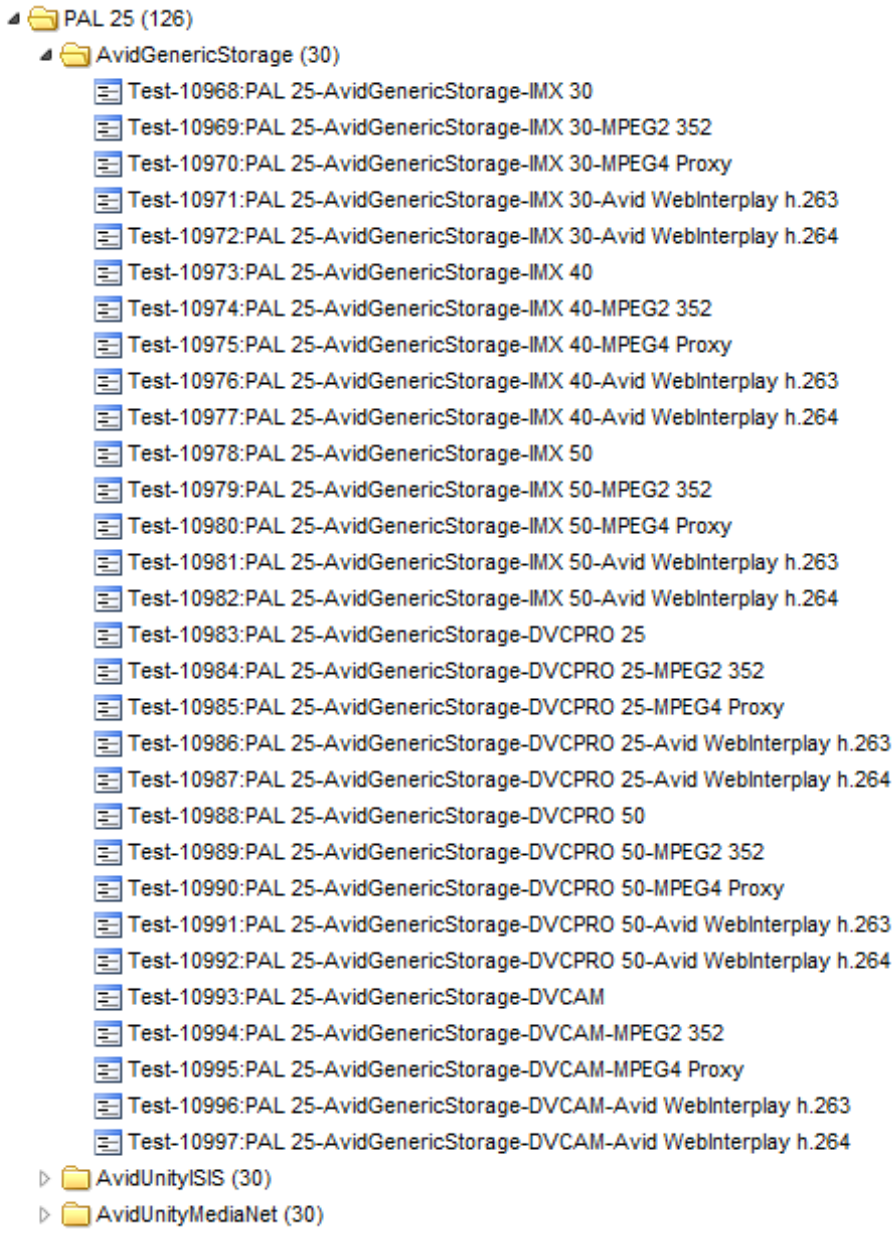


Figure 3.8: Test folder structure

This solution was soon to be found complex and confusing. All variables were thrown in the pre-conditions. This way, since one test case tested a lot of features, it was hard to debug a test and to see where it failed.
A second solution was then designed.

Solution(Second attempt)

A more complex folder structure, seen in figure 3.9, was created to easily separate and identify the tests. Each major step of the precious tests was divided into a separated, single test. Instead of one test containing everything, there was a test for the login procedure, another for the storage,

for the profile configuration and for the recording. The test structure was more atomic, since there was now 4 tests where before there was only one.

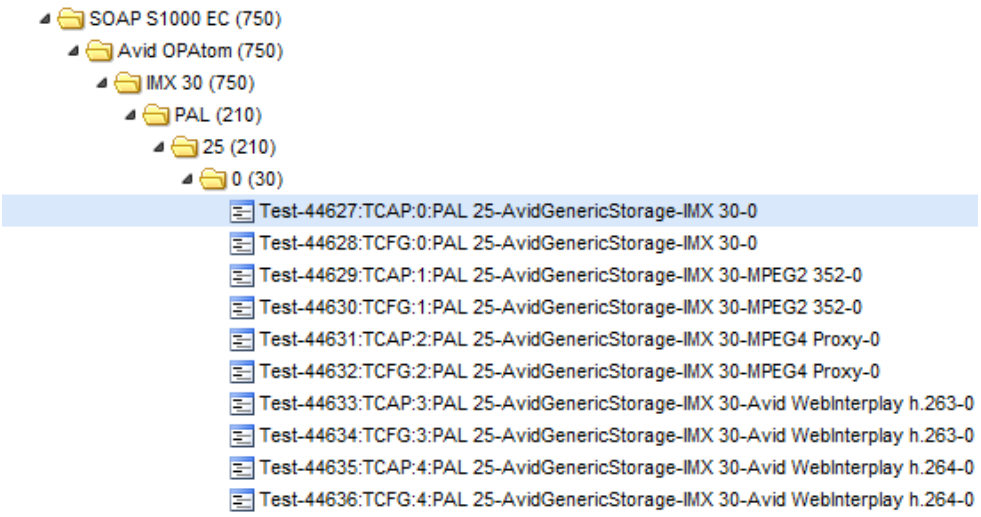


Figure 3.9: New test folder structure

To connect these different tests, since ones had to be performed before the others, the tests to be performed first started being used as pre-conditions for the subsequent ones. To perform a capture test, the profile configuration test specified in the capture test precondition had to be performed first.
Due to the different characteristics of each type of test, each type had its different syntax.

Since the test structure was updated, changes had to be made to the test generator module to accommodate this new structure. A set of filters were applied to divide the tests suites as explained before.

3.2.4 Requirements definition

Problem:

Following the atomic tests model, a product requirement comprises more than one test. However, there is no entity that functions as link between test cases (and their results) and product requirements. This entity will allow an easier test coverage calculation. Besides having statistics relative to test cases, statistics for the product requirements will allow a more detailed and concrete evaluation of the product status.

Solution

The creation of product requirements was then planned: a requirement, in this case, consists of a feature to be tested. It should comprise the tests that fully cover that feature. Since the creation

of tests was automatic, it was logical that the generation of requirements would also be automatic. The connection between both entities was made in the test generator module: when generating the tests, it would also generate the correspondent requirements, in TL syntax, and link these two entities. The user, when creating the test plan, would only have to add the requirements.
Not every product requirement has the same significance for the validation process. There are some requirements that must be checked in order to successfully validate the product, while others are less important. These, in a time shortage situation, will not be tested. To separate these different types, it was decided to set a priority for each requirement. The priority is set based on the customers order and can vary between three values: high, medium or low.

- **High priority requirements** should cover the functionalities used in the clients workflow. This requires a profound knowledge of the clients line of work and the features they will use. For a product to be validated, a complete coverage of this type of requirements must be performed.
- **Medium priority requirements** cover the features that can be used by the client but not as frequent or likely as the high priority ones.
- **Low priority requirements** cover all the remaining features, not commonly used in the client's workflow.

Ideally, the goal is to cover 100% of all the requirements. Due to time constraints that is not always possible. So, depending on the time available, different requirement coverage plans can be made. For example, the goal can be to cover 100% of the high priority requirements, 50% of the medium and 20% of the low ones.

3.3 SOAP Tests Execution

3.3.1 Test plan manager module

Problem: Since the test structure was now designed and specified, the next step was to read plans from testLink and perform the test cases defined there. It was decided to start by the SOAP tests, since they are faster than the GUI testing. This will allow the framework to test more products in a shorter period of time.
The tests needed to have a specific syntax so that their parameters could be parsed and sent through SOAP functions. The module to be implemented has to read data from testLink and let the user choose what tests he wants to perform.

Solution: The test cases needed to be rewritten so that the parsing would be easier. A standard syntax was defined for the several types of tests.

More parameters can be added easily as extra steps, putting between square brackets the parameter name and its value. The framework allows an easy configuration of extra parameters and extra types of tests.

```
1
2  Capture Tests:
3  Name: TCAP:XX:test_name
4  Summary: [SOAP]
5  Preconditions:[Test:TCFG:XX]
6  Steps:
7  [StartCapture]
8  [Duration:YY]
9  [StopCapture]
10
11 XX-id (has to be the same as the configuration test)
12 YY-number of seconds to capture
13
14 Profile Configuration Tests:
15 Name:TCFG:XX:test_name
16 Summary:[SOAP]
17 Preconditions:[Test:TSTR:storage_name]
18 Steps:
19 [Resolution:resolution_name:resolution_ID]
20 [Framerate:number]
21 [AudioChannels:number_of_audio_channels]
22 [Codec:codec_name:codec_ID]
23 [Wrapper:wrapper_name]
24 [Storage:nameofstorage:storagecode:storagepath]
25 (optional)
26 [ProxyCodec:codec_name:codec_ID]
27 [AudioMode:mono_or_stereo]
28 Expected Results:
29 [Status:ready]
30
31 Storage Configuration Tests:
32 Name:TSTR:storage_name
33 Summary:[SOAP]
34 Preconditions:[Test:TLogin]
35 Steps:
36 [Name:storage_name]
37 [Kind:kind_of_storage]
38 [Server:name_of_server]
39 [User:username]
40 [Pass:password]
41 (optional)
42 [FCServer:name]
43
44 Login Test:
45 Name:TLogin
46 Summary:[SOAP]
47 Preconditions:
48 Steps:
49 [User:username]
50 [Pass:password]
51 Expected Results:
52 [Result:!=:None]
```

Listing 3.1: Tests syntax rules

```
'id': '139305',
'author_last_name': 'Nabuco',
'tc_external_id': '23283',
'version': '1',
'testsuite_id': '139301',
'testcase_id': '139304',
'author_first_name': 'Miguel',
'importance': '2', '
'preconditions': '<p>[Test:TCFG:8594]</p>',
'creation_ts': '2012-05-08 10:38:21',
'name': 'TCAP:8594:1440x1080p 25-XDCAMGenericStorage-
MPEG2 LGOP  18-8',
'summary': '<p>[SOAP]</p>',
'steps':
{ 'step_number': '1',
  'actions': 'Start Capture',
  'expected_results': ''},
{ 'step_number': '2',
  'actions': '[Duration:10]',
  'expected_results': ''},
{ 'step_number': '3',
  'actions': 'Stop capture',
  'expected_results': 'pass'}
```

Listing 3.2: Test information

To fetch the necessary information, test cases and test plans, from TL, a XML-RPC client was implemented in Python to communicate with the XML-RPC server in TL. This client has functions that allow the retrieval of test plans and test cases as well as the report of each test result.

3.3.2 Test execution

Problem: To perform the tests in the products, a module had to be implemented to create a bridge between the test plan manager module and the SOAP server implemented in the products.

Solution: This module can be divided in two parts: parsing of the test information and test execution itself. The test comes from TL in a Python dictionary. A sample from a test case can be found in the code above, in Listing 3.2.
It is fairly easy to parse the necessary information from this structure. The framework will check if it is a SOAP or GUI test (this is presented in the summary) and redirect the test to the appropriate module. After this first check, a second one follows to determine the type of the test. As it is seen in the tests name, *TCAP* means that it is a capture test.
Depending on the type of test, a different function will be called that will parse the information presented on the *Steps* and call SOAP functions to perform the action required (in this case, start and stop the recording).

3.3.3 Test results manager module

Problem:

After the tests execution, the results have to be reported to TL so that the users can see how the product performed. The tests that failed must also contain the necessary information to debug the test, which errors occurred.

Solution:

In a similar way to the test plan manager module, to send the results it is also used XML-RPC functions. To determine if a test has passed or not, the framework keeps track of every error that may appear in the product log. In case of errors, the test case is immediately marked as failed and the errors are appended as notes to the test case. Users can then see where the problem occurred.

When this module was completed, a full cycle could be made. The very first proof of concept was made with the simplest test, the login. This experimental proof of concept consisted in:

- Creating automatically the test cases and import them to TL.
- Creating a test plan and only adding the login test to that plan.
- Performing the test plan through the framework and wait for results to update the test status in TL.

3.3.4 Test tree

Problem:

After successful completion of the initial proof of concept, it is time to run a test plan with multiple tests. However, there were some issues: since there was no order in the test execution, sometimes the tests that were preconditions of another tests (meaning that they should run first), were being performed after. Other times, if these tests ran first but failed, the ones that had them as pre-conditions would obviously fail as well, when they should not even run.

Solution:

The solution found was to implement an iterable structure to store the tests and organize them to be performed in a logical order. A tree was found to be adequate for the purpose; the root node is the first test to be executed and it is the one that has no pre-conditions. The children nodes of any node are the tests that contain the parent node test case as their pre-conditions.
The table 3.1 contains an example of this structure.

Table 3.1: Test/pre-conditions relation example

Test	Pre-conditions
A-Login	-
B-Storage	A-Login
C-Storage	A-Login
D-Config	B-Storage
E-Config	B-Storage
F-Config	C-Storage

From the table of tests a tree can be created, as seen in figure 3.10.

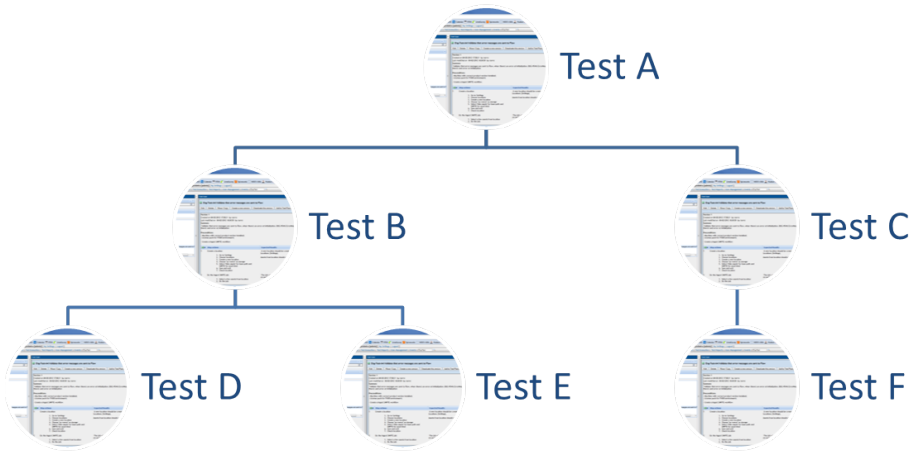


Figure 3.10: Test Tree

If no test fails, the test order is A-B-D-E-C-F. When one test fails, it is marked as failed and all its children nodes are marked as blocked.
A node is composed by an ID, the information from the test case and a list of children nodes. The tree structure only points to the root node and iterates from there.
When executing a test plan, the method can be divided into steps:

- **Step 1:** Find the root test case, the one with no pre-conditions, and tag it as the main node.
- **Step 2:** Iterate through the test cases list to find tests that have the main node as precondi-tions.
- **Step 3:** Add the selected tests to the main node children nodes list.
- **Step 4:** For each child node, tag it as the main node and repeat from **Step 2**.

3.4 Implementation details

3.4.1 Time control

After implementing the test tree, it was decided to find out how effective and fast the solution implemented was. To do this, time counters were placed in several points to measure the efficiency of each process.
The results were saved in a text file for analysis. Several test plans were tested, with 4, 11, 23 and 113 tests. The results can be found in table 3.2 and in figures 3.11, 3.12 and 3.13. The values for time are represented in seconds.

Table 3.2: Time performance

Number of tests	TestLink fetching	Tree creation
4	0.51	3.66
11	0.97	22.02
23	1.01	96.89
113	-	>1800

```
Number of tests: 4
TCFG:145:NTSC 30-AvidGenericStorage-IMX 30-MPEG2 352-2
TSTR:AvidGenericStorage
TLogin
TCAP:145:NTSC 30-AvidGenericStorage-IMX 30-MPEG2 352-2
Test fetching from testlink took 0.513000011444 seconds
Tree Creation took 3.65600013733 seconds
Login test took 11.0399999619 seconds
Storage test took 10.7139999866 seconds
Config test took 10.7590000629 seconds
```

Figure 3.11: Time performance results with 4 tests

```
Number of tests: 11
TSTR:AvidUnityISIS
TCAP:0:PAL 25-AvidGenericStorage-IMX 30-0
TCFG:11:PAL 25-AvidUnityMediaNet-IMX 30-MPEG2 352-0
TSTR:AvidGenericStorage
TSTR:XDCAMStructureStorage
TCFG:0:PAL 25-AvidGenericStorage-IMX 30-0
TSTR:QuickTimeStorage
TCAP:11:PAL 25-AvidUnityMediaNet-IMX 30-MPEG2 352-0
TSTR:XDCAMGenericStorage
TSTR:AvidUnityMediaNet
TLogin
Test fetching from testlink took 0.97000002861 seconds
Tree Creation took 22.0199999809 seconds
```

Figure 3.12: Time performance results with 11 tests

```
Number of tests: 23
TCFG:137:NTSC 30-AvidUnityISIS-IMX 30-Avid WebInterplay h.264-1
TCFG:144:NTSC 30-AvidGenericStorage-IMX 30-2
TCAP:144:NTSC 30-AvidGenericStorage-IMX 30-2
TCFG:158:NTSC 30-AvidUnityMediaNet-IMX 30-Avid WebInterplay h.264-2
TCAP:334:PAL16:9 25-AvidUnityMediaNet-IMX 30-Avid WebInterplay h.264-16
TCFG:334:PAL16:9 25-AvidUnityMediaNet-IMX 30-Avid WebInterplay h.264-16
TCAP:158:NTSC 30-AvidUnityMediaNet-IMX 30-Avid WebInterplay h.264-2
TSTR:AvidUnityISIS
TCFG:11:PAL 25-AvidUnityMediaNet-IMX 30-MPEG2 352-0
TSTR:AvidGenericStorage
TCFG:0:PAL 25-AvidGenericStorage-IMX 30-0
TSTR:XDCAMGenericStorage
TCFG:130:NTSC 30-AvidGenericStorage-IMX 30-MPEG4 Proxy-1
TCAP:130:NTSC 30-AvidGenericStorage-IMX 30-MPEG4 Proxy-1
TCAP:137:NTSC 30-AvidUnityISIS-IMX 30-Avid WebInterplay h.264-1
TSTR:AvidUnityMediaNet
TLogin
TCFG:138:NTSC 30-AvidUnityMediaNet-IMX 30-1
TCAP:0:PAL 25-AvidGenericStorage-IMX 30-0
TCAP:138:NTSC 30-AvidUnityMediaNet-IMX 30-1
TSTR:XDCAMStructureStorage
TSTR:QuickTimeStorage
TCAP:11:PAL 25-AvidUnityMediaNet-IMX 30-MPEG2 352-0
Test fetching from testlink took 1.00600004196 seconds
Tree Creation took 96.8870000839 seconds
```

Figure 3.13: Time performance results with 23 tests

Each test took approximately 11 seconds to perform. With the 113 tests test plan, measure is not completely correct because the framework created a corrupted file. However, manual measurement proves that it took over 1800 seconds, more than half an hour.

These results were disappointing and showed that the solution implemented was ineffective. The test tree implementation had to be improved, because it was the bottleneck, the process that took the most time. In the previous tree implementation, each time it wanted to retrieve information from a test, it would connect to testLink to retrieve it. Meaning that, each time a test was accessed, it took some time to establish the connection. Also, when organising the tree, each node contained a lot of unnecessary information that came from TL, such as test author, last modification date and others.

A new test tree was then implemented to eliminate these defects. Before building the tree, the program now fetched the information from every test and created the nodes, storing them in a temporary list. That extra information from the nodes was removed and now they only contained essential data:

- Name
- Summary
- ID
- Preconditions
- Steps (including expected results)

The tree is created the same way as before, except that now it only iterates through this temporary list. Less information means smaller data to work with.
The time measurements were repeated to verify the efficiency of the new solution and the results can be seen in 3.3 and in figure 3.14.

Table 3.3: Time performance with the new tree implementation

Test tree	Number of tests	Total time
Previous	113	>1800
New	157	36.75

```
Number of tests: 157
Test fetching from testlink took 36.753000021 seconds
Tree Creation took 0.336999893188 seconds
```

Figure 3.14: New tree implementation time results

The removal of all the unnecessary data and the reduction of TL servers accesses greatly improved the efficiency of the test tree.

3.4.2 Workflow test

Now that a first functional version of the framework was working, it was time for a workflow test. This test has the purpose of verifying the consistency between all the modules.

Before starting the test, the environment had to be prepared. One S1000 product, where the tests would be ran, was fully prepared and connected to a SDI signal generator. This workflow test was divided into the following steps:

- Generate the tests
- Import them into testLink
- Create a test plan with some tests of different types
- Run the tests with the SOAP module
- Return the results to testLink
- Observe the results

The test plan to be performed, represented in figure 3.15, contained the login test, all the storage server configuration tests, three profile configuration tests and three video capture tests.

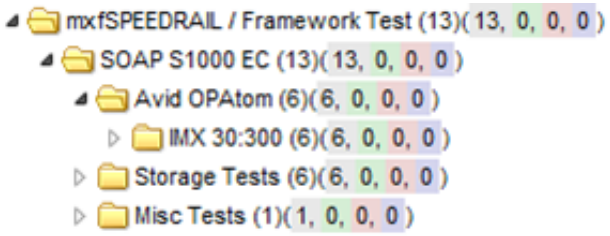


Figure 3.15: Conceptual proof test plan definition

The figure 3.16 shows the user interface. The user can choose the test project and test plan he wants to perform. Before the actual execution, the tests contained in the selected test plan are listed (figure 3.17) and the user gets a final prompt. The tests are then performed through the SOAP interface.

```
Menu
    1) List Projects
    2) Search Test Case By External ID
    3) Update Test Case Results
    4) Execute Test Plan
    5) Test Parser
    6) Try RIATest "login test"
    7) Exit
Make a selection> 4

Name: mxfSPEEDRAIL ID: 1
Name: mxfSPEEDRAIL_Eng ID: 254

Select the ID of the project:
1
Name: Framework Test ID: 338164
Name: O1000 ID: 31
Name: P1000 ID: 32
Name: S1000 ID: 30

Select the ID of the test plan:
338164
```

Figure 3.16: Selecting test plan through user interface

```
Select the ID of the test plan:
338164

Tests:

TSTR:XDCAMStructureStorage
TCFG:13:PAL 25-AvidUnityMediaNet-IMX 30-Avid WebInterplay h.263-0
TCFG:1:PAL 25-AvidGenericStorage-IMX 30-MPEG2 352-0
TCFG:0:PAL 25-AvidGenericStorage-IMX 30-0
TSTR:XDCAMGenericStorage
TCAP:0:PAL 25-AvidGenericStorage-IMX 30-0
TSTR:AvidUnityMediaNet
TCAP:1:PAL 25-AvidGenericStorage-IMX 30-MPEG2 352-0
TSTR:QuickTimeStorage
TCAP:13:PAL 25-AvidUnityMediaNet-IMX 30-Avid WebInterplay h.263-0
TLogin
TSTR:AvidGenericStorage
TSTR:AvidUnityISIS

Execute test plan? (y/n)
y
```

Figure 3.17: User interface showing tests in selected test plan

The results are represented in figure 3.18. The login and storage tests passed (in green), all

the configuration tests failed (in red) which caused the capture tests (in blue) to block, following the tree structure previously explained.

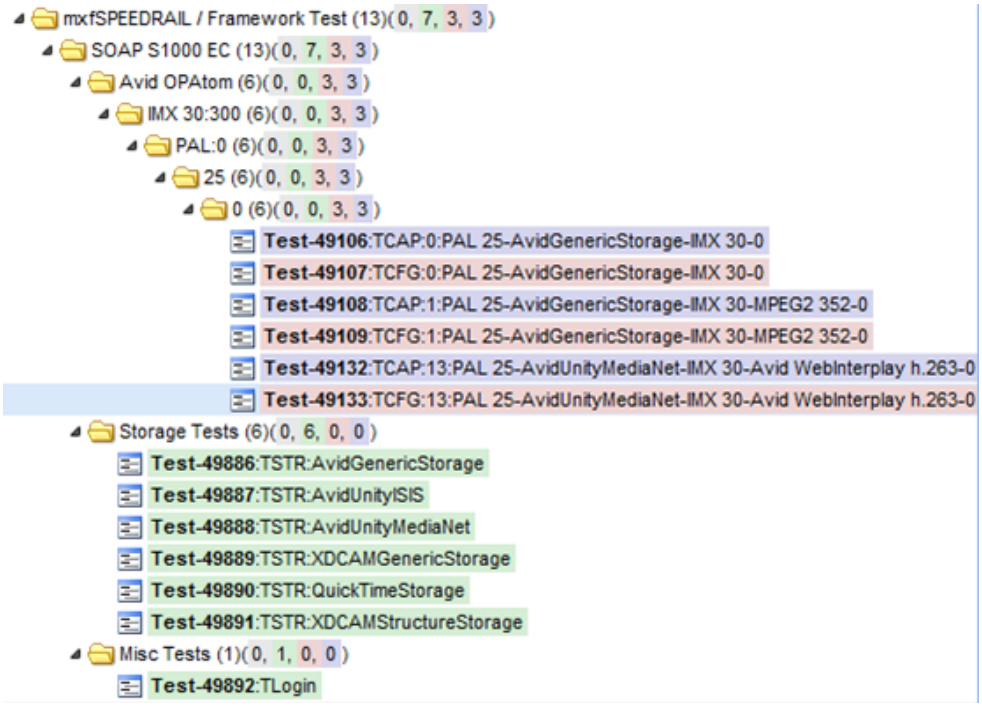


Figure 3.18: Execution results in testLink

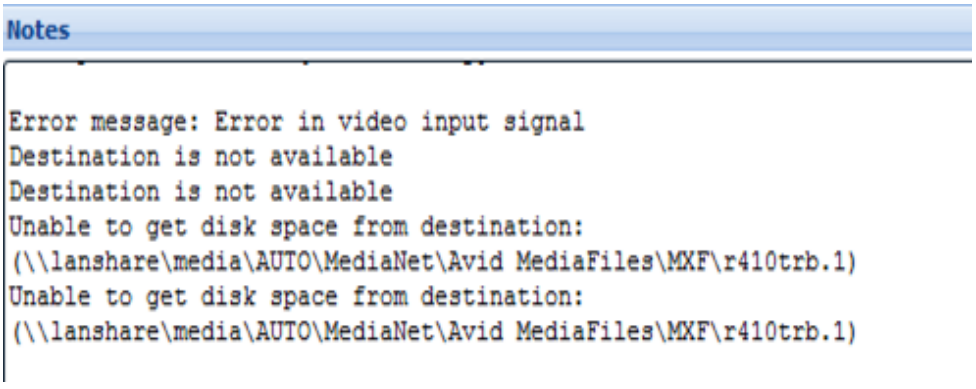


Figure 3.19: Configuration test error message

As it can be seen from figure 3.19, the errors in all the tests were from storage configuration (the path was not valid) and also from the video input.

3.4.3 Signal generator module

Figure 3.20: Signal generator

Problem

The tests mainly failed due to the video signal generator. The signal generator, in figure 3.20, controls the resolution and framerate that serves as input to the product. Since the video input is changing from test to test, there needs to be a way to control the signal generator.
Normally this control was made manually. Due to the large number of tests, an automated module to control the signal generator had to be created.

Solution

Following the instructions in the signal generator's communication protocol manual, a tool in C# was created that created that received the framerate and resolution from the test case and generated the necessary instructions. These instructions were sent to the framework machine serial port, where the signal generator was connected.
After this module implementation, the workflow test was repeated. The test plan created had both NTSC and PAL tests, to prove the new module reliability.

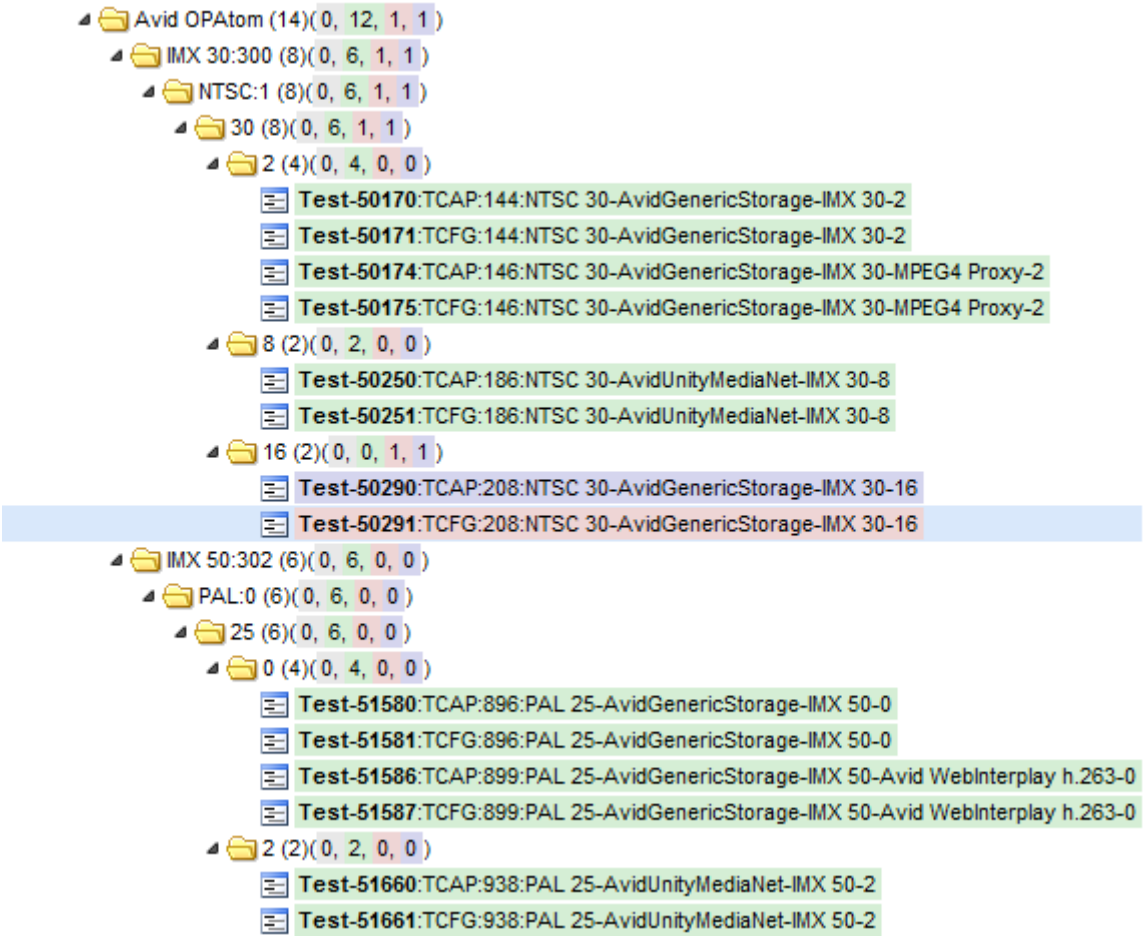


Figure 3.21: Execution results with different resolutions

As it can be seen from the figure 3.21, only one configuration test failed now because the board didn't support 16 audio channels.

3.4.4 Stress tests

With the positive results of the workflow test, it was time to test the framework and TL to their limits. The previous test plan had very few test cases. It was decided to load the framework with all the test cases needed to fully validate the most complete product. With all these tests, over 20 000, TL had problems processing all the data. The figure 3.22 shows that the CPU usage to load a web page with the information from 804 tests was very high.

To load that same page took 2 minutes in a machine with 4 cores and 4 Gb or RAM. It spent 2 other minutes to load them into a test plan. Trying to load a test plan with 13000 tests to the framework, there was a connection error due to the amount of data being sent.
This situation was unbearable and it limited the fundamental framework purpose, to fully validate the company's products.

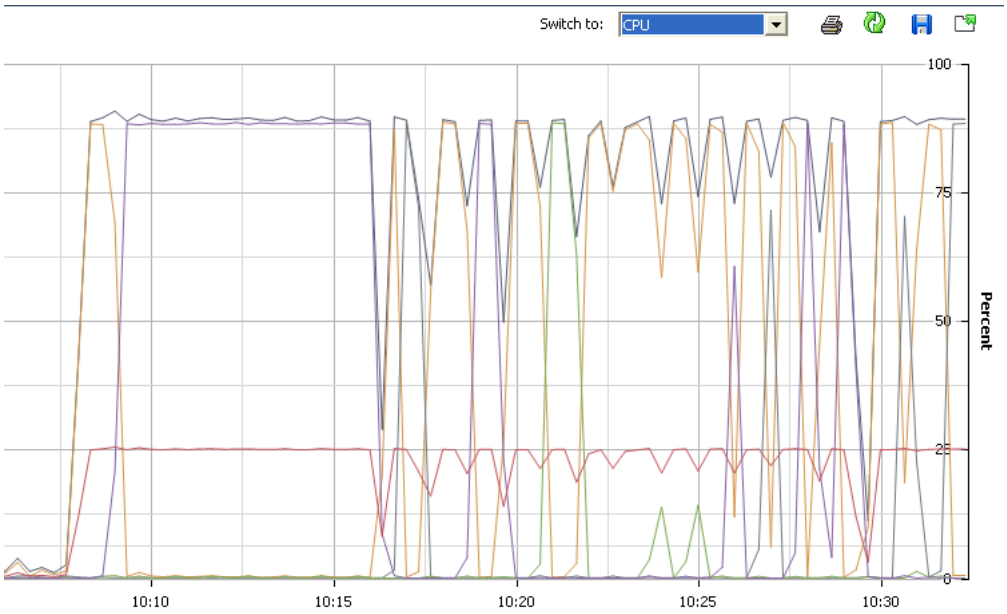


Figure 3.22: testLink CPU usage for loading a web page with 804 tests

To improve TL's performance, TL server was reallocated to a dedicated machine and a dedicated database, with more resources. All the possible PHP and SQL variables that could be limiting TL performance were maxed out.
Although the performance didn't improved exponentially, the gain in time spent loading data (over 60%) made TL's usage acceptable.

However, the problem with huge data transfer between TL and the framework remained. To measure the number of tests that could be put in a test plan, several test plans were made with an increasing number of test cases.

Table 3.4: testLink stress test

Number of tests	Time to start execution	Time to fetch the tests	Time to create the tree
847	115	105	10
1687	223	208	15
2527	350	324	26
3367	451	406	45
3871	537	478	59
3996	-	-	-

In table 3.4, the test plans and the times (in seconds) needed to process them are shown. With the test plan containing 3996 test cases, the testLink could not send the data to the framework. It was concluded that the test plans should contain a maximum of 3500 test cases each to avoid data loss.

3.4.5 License Manager

The goal of the framework was to validate the company's products. Inside a specific series there are different products, determined by the licenses they have. A license is a set of video codecs and formats that the product can process and record to. For example a S1111 XD.SD can only record in AVID format using IMX codecs.

To make sure a specific product is correctly tested, it has to have installed only the necessary licenses. The company had two tools regarding licenses: one to delete all the licenses in a product and another to generate specific licenses. There was no way to automatically generate the files that contain the licenses. However, since these files are only needed to be generated once, there was no need to automate this process.

Each time a product (or a set of products) was to be tested, the folder containing the licenses would be put in the hardware operating system. When executing a test plan that contained the product name, a script was run to delete the existing licenses and install only the necessary ones. The communication between the framework and the machine to be tested was made using telnet. The framework uses telnet commands to login to the remote machine and execute the license registry file.

Since a product can only process certain video codecs, not every test case needs to be generated because most of them will fail. Therefore, changes to the test generator module had to be made. Filters were introduced so that the user can choose which product he wants to test, as shown in figure 3.23. The test generator will only create the tests for that product.

```
    Select Product to generate tests:
Name: S1000 ID: 1000
Name: S1100 ID: 1100
Name: S1010 ID: 1010
Name: S1011 ID: 1011
Name: S1012 ID: 1012
Name: S1013 ID: 1013
Name: S1110 ID: 1110
Name: S1111 ID: 1111
Name: S1112 ID: 1112
Name: S1113 ID: 1113
Choose the ID> 1111
Select Product Type:
Type: XD_SD ID: 7001
Type: DV_SD ID: 7002
Type: XD_HD ID: 7003
Type: DV_HD ID: 7004
Type: DN_HD ID: 7005
Type: AV_HD ID: 7006
Type: DN_HD220 ID: 7010
Choose the ID> 7001
{'license': 'XD_SD', 'max_audio_channels': 8, 'resolution': 'SDHD', 'name': 'S1111', 'wrapper': 'OP1a'}
done
30.010999918 seconds
240 combinations
480 tests
```

Figure 3.23: Test generator creating the S1111 XD.SD tests

Chapter 4

Case Study/Results

In this section, it will be explained how the framework is being used in the company's testing worklow, as well as some test coverage results.

4.1 Feature testing

Each new product version comes with a set of features that need to be tested. So, besides using the framework to test entire products, it can also be used to validate these features. This means that, each time a feature is launched, there is no need to run the whole product test plan. A new test plan is created with only the necessary tests.

A practical example will be demonstrated with feature 5955 that consisted in allowing the generation of MXF OP1a files with XDCAM with more than 8 audio channels.

After creating the test plan(figure 4.1) and verifying that all the software and hardware requirements (figure 4.2) are met, the framework is run.

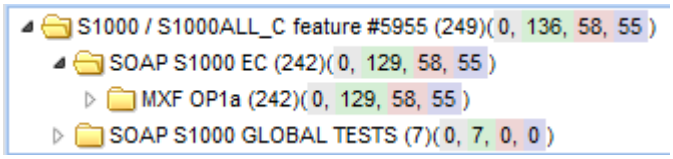


Figure 4.1: Test plan to validate feature 5955

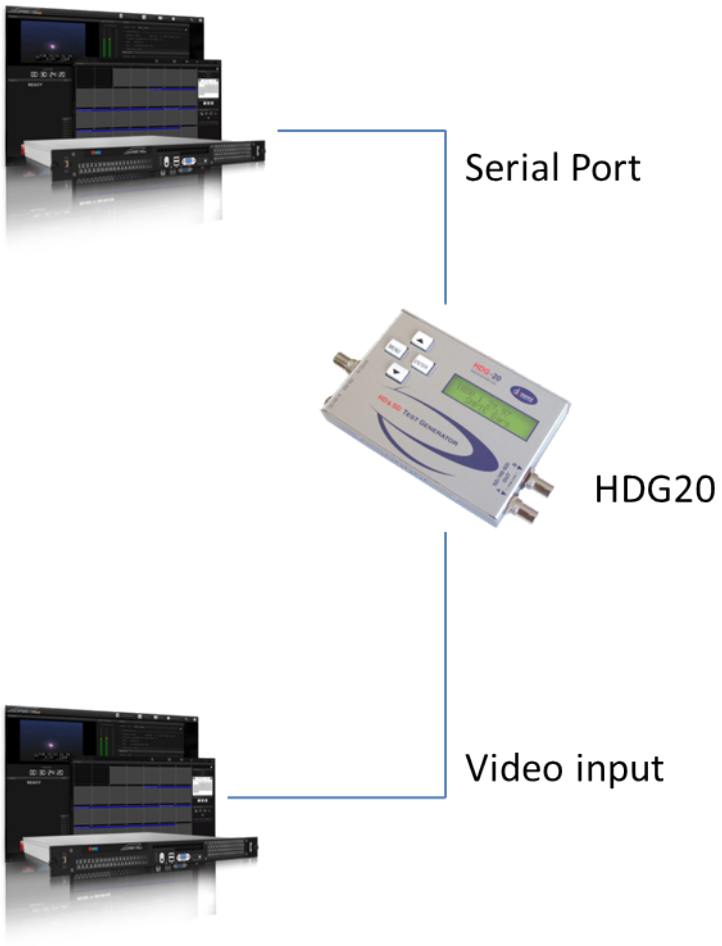


Figure 4.2: Hardware requirements

The test plan results can be seen in figure 4.3. As it can be seen, half of the tests failed or were blocked so further analysis had to be made to find a common pattern on the tests that failed. This analysis has to be made manually, otherwise the framework would create dozens or hundreds of bugs for the same issue. In other situations it could happen that the test failed due to a wrong preparation of the environment, therefore not being a software bug. The figure 4.4 shows that this feature has problems in HD progressive resolutions (1440x1080p and 1920x1080p).

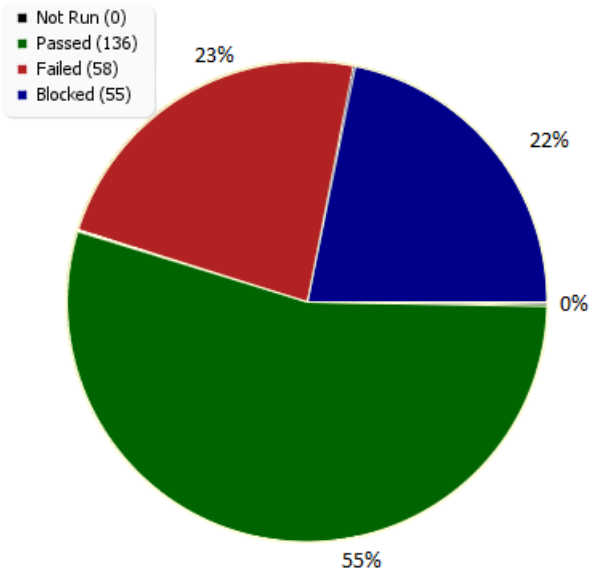


Figure 4.3: Test result chart

SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 8	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 8	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 8	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 12	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 12	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 12	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 16	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 16	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 16	S-	Medium	Failed [v1]
SOAP S1000 EC / MXF OP1a / MPEG2 LGOP 50 422:420 / 1920x1080p:10 / 25 / 16	S-	Medium	Failed [v1]

Figure 4.4: Failed tests sample

Since this problem with HD progressive resolutions was also present in another formats and already reported, there was no need to open a new bug.

4.2 Product Validation

The product series S1000 is divided in several products. Since the machine used for testing had a maximum of 8 audio channels only the S11xx products could be validated. S10xx products have a different board that has support for 12 audio channels.

Inside S11xx, there are more variants, according to wrappers, as shown in table 4.1.

Table 4.1: Products by wrapper

Product	**Wrapper**
S1110	Avid OPAtom
S1111	OP1a
S1112	All
S1113	QuickTime

The first two variants, S1110 and S1111, were successfully validated and the results can be seen in table 4.2 (for better legibility, the tests that failed and were blocked are grouped). Each variant also has its products, which differ from the type of codecs installed.

Table 4.2: Validation status

Variant	**Product**	**Tests made**	**Passed**	**Failed**	**Ratio**
S1110	XD:SD	1810	1794	16	99.1%
S1110	DV:SD	1810	1789	21	98.8%
S1110	XD:HD	2894	2851	43	98.5%
S1110	DV:HD	848	785	63	92.3%
S1110	DN:HD	1992	1939	52	97.3%
S1110	AV:HD	1090	-	-	-
S1111	XD:SD	488	437	51	89.5%
S1111	DV:SD	248	248	0	100%
S1111	XD:HD	2894	2851	43	98.5%
S1111	DV:HD	128	126	2	98.4%
S1111	DN:HD	348	327	21	94%
S1111	AV:HD	1810	-	-	-

Every product has a different number of tests due to the amount of codecs and proxies it is able to process. The AV:HD products could not be tested because its codecs, AVCIntra 50 and 100, require a more powerful machine than the one being used for testing.

The results were very good. Every product got a test success rate above 90%, with the exception of S1111 XD:SD which came quite close (89.5%).

The total test results are shown in figure 4.5.

Based on further analysis on all the tests, several patterns were found. This means that many tests failed because of the same reason. Several bugs were then reported.

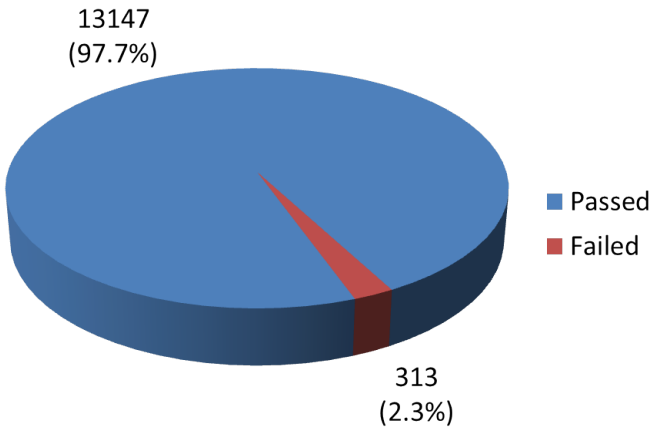


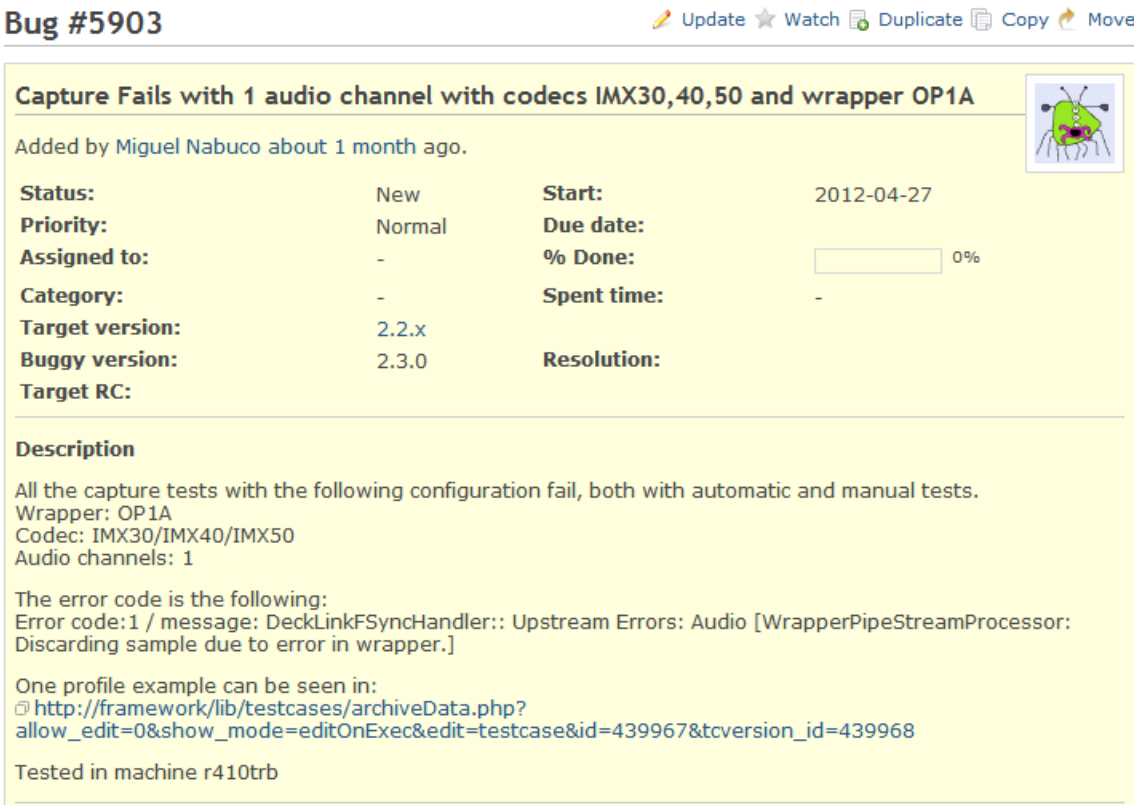
Figure 4.5: Total test results

4.3 Bug reporting

During framework tests, it was found a bug that was not yet discovered and reported. In this cases, the tester needs to open a bug in the bug tracking system Redmine.

To open a bug, several details have to be provided such as subject, description and buggy version. Files can also be appended to the bug report, in case they are needed to replicate it. In the description, there is a link to the test cases that failed due to the bug. The figure 4.6 shows the bug report.

When the bug is created, a mail is sent to the engineering team. The project leader will then assign the bug to a developer. The developer will then update the bug status, such as time spent working on it or percentage of the work done to fix it. After the engineering team fix the bug, it needs to be tested again, to ensure the issue was properly solved.



Bug #5903

Update Watch Duplicate Copy Move

Capture Fails with 1 audio channel with codecs IMX30,40,50 and wrapper OP1A

Added by Miguel Nabuco about 1 month ago.

Status:	New	**Start:**	2012-04-27
Priority:	Normal	**Due date:**	
Assigned to:	-	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	2.2.x		
Buggy version:	2.3.0	**Resolution:**	
Target RC:			

Description

All the capture tests with the following configuration fail, both with automatic and manual tests.
Wrapper: OP1A
Codec: IMX30/IMX40/IMX50
Audio channels: 1

The error code is the following:
Error code:1 / message: DeckLinkFSyncHandler:: Upstream Errors: Audio [WrapperPipeStreamProcessor: Discarding sample due to error in wrapper.]

One profile example can be seen in:
http://framework/lib/testcases/archiveData.php?allow_edit=0&show_mode=editOnExec&edit=testcase&id=439967&tcversion_id=439968

Tested in machine r410trb

Figure 4.6: Bug report created

Chapter 5

Conclusions and Future Work

System testing is a fundamental part of software development. It allows the testers to simulate the final customers workflow. Using a black box methodology, the automatic tests were designed without any knowledge of the system inner workings.

To efficiently design tests, the tester must analyse the system and choose the correct approach. In this case, atomic, simple tests were found to be more accurate in bug finding and reporting. If one of these tests failed, it was very simple to verify where the problem occurred. This would not be possible if the tests were more complex and contained a lot of different actions.

The simple tests structure needed a specific execution order. To ensure this, some tests were preconditions of another tests, meaning that if the precondition of a test failed, that test would not be run. To perform the tests in this specific order, a tree structure was implemented.

5.1 Goal Satisfaction

During the framework implementation, many obstacles appeared. Signal generator issues and problems with testLink, the test management tool, were some of them. These obstacles were successfully being defeated as the implementation was being concluded.

Comparing to the initial goals, most of them were successfully achieved. The framework was completed and it is currently testing the company's products. It was decided during the implementation that SOAP tests should be of critical importance. By being much faster than GUI tests, they allow a greater coverage in shorter time.

Since it was decided to focus on the SOAP tests, the GUI tests module was not developed. It was not possible to fully test all product series. However, the validation of several products was

sufficient to prove the framework's efficiency.

Manual tests are still made to cover the features that the framework cannot test. The implementation of the framework managed to remove some of the testers work load. To compare the tests covered in the framework, the manual way to create the test, insert the data into testLink, perform it and report the results would take more than 10 minutes, since it also involves several sub-tasks like configuring the signal generator. The automatic way will take no more than one minute in the whole process.

Another significant advantage is that the framework can run 24 hours a day, this way being able to run thousands of test cases over one night, something that would be impossible if made in a manual way.

5.2 Further Work

The framework was developed for validation testing. However, with little effort, it can be easily implemented in the verification testing process. Since the framework is modular, it is easy to add the necessary features.

To further improve the test coverage and the efficiency of the result analysis, a re-design of testLink's database is planned. This would result in the creation of a new database where only the necessary data from the testLink instances (manual and automatic tests) would be put. Further experiments with testLink will also be made to improve its efficiency and reliability.

It is also being considered the implementation of new modules for better and more complete file validation.

References

[Amb] Scott W. Ambler. Introduction to test driven development. http://www.agiledata.org/essays/tdd.html.

[Bec03] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, May 1995.

[BL01] Lionel Briand and Yvan Labiche. A uml-based approach to system testing. In *«UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001.

[Boe81] Barry Boehm. Software engineering economics. 1981.

[Boe07] Barry Boehm. Revisiting software engineering economics. March 2007. http://www.cs.stevens.edu/~lbernste/cs552spr07/Lectures/CS%20552%20SER.pdf.

[Bur03] Ilene Burnstein. *Practical software testing*. Springer, 2003.

[Cor01] Intel Corporation. *Open Source Computer Vision Library – Reference Manual*. 2001. http://itee.uq.edu.au/~iris/CVsource/OpenCVreferencemanual.pdf.

[Des] Blue Claw Database Design. Software risk management, risk analysis, control and assessment. http://www.blueclaw-db.com/software_risk_assessment.htm.

[DRP99] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management and performance*. Addison-Wesley Professional, 1999.

[Dus01] Elfriede Dustin. The automated testing lifecycle methodology. May 2001. http://www.informit.com/articles/article.aspx?p=21468&seqNum=3.

[GG] Anand Gopalakrishnan and Keith Gallagher. Conquest: Interface for test automation design. http://www.qualitycow.com/Docs/ConquestInterface.pdf.

[Hel08] Software Testing Help. What is boundary value analysis and equivalence partitioning?, 2008. http://www.softwaretestinghelp.com/what-is-boundary-value-analysis-and-equivalence-partitioning/.

[ioeee90] The institute of electrical and electronics engineers. *IEEE Standard Glossary of Software Engineering Terminology*. May 1990. http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf.

REFERENCES

[Kel03] Michael Kelly. Choosing a test automation framework. November 2003. http://www.ibm.com/developerworks/rational/library/591.html.

[KH07] Adam Kolawa and Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society, 2007.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. January 2004.

[Ltd11] TestPlant Ltd. Black-box vs white-box testing: Choosing the right approach to deliver quality applications, 2011. http://www.testplant.com/wp-content/uploads/downloads/2011/06/BB_vs_WB_Testing-1.pdf.

[Luc11] Jeff Lucas. Automate small, test big, 2011. http://www.softwaretestingclub.com/profiles/blogs/automate-small-test-big.

[Mat08] A.P. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[Mir98] Eduardo Miranda. The use of reliability growth models in project management. November 1998.

[MSP01] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for gui testing. September 2001.

[NH02] Thomas Nagle and Reed Holden. *The Strategy and Tactics of Pricing*. Pretince Hall, 2002.

[Ost02] Thomas Ostrand. *White-Box Testing*. John Wiley and Sons, Inc., 2002.

[Per92] William E. Perry. *A Standard For Testing Application Software*. Auerbach Publishers, 1992.

[Rec] Bill Reckwerdt. White paper: Video testing for broadcasters. http://www.videoclarity.com/PDF/WPBroadcastTesting.pdf.

[RIA] RIATest. Riatest automation for adobe flex features. http://www.riatest.com/products/features.html.

[Rot01] Johanna Rothman. Risk analysis basics, January 2001. http://www.stickyminds.com/sitewide.asp?knav=colarchive&ObjectId=3061&ObjectType=COL&Function=edetail.

[Sch04] Hans Schaefer. Risk based testing – how to choose what to test more and less, 2004. http://www.cs.tut.fi/tapahtumat/testaus04/schaefer.pdf.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, September 2005.

[Too] QA Testing Tools. X-unit – code driven testing. http://www.qatestingtools.com/xunit_code_driven_testing.

[Tra99] Eusshiuan Tran. Verification/validation/certification. Technical report, Carnegie Mellon University, Spring 1999. http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html.

REFERENCES

[TSWI] Anas Tawileh, SteveMcIntosh, Brent Work, and Wendy Ivins. The dynamics of software testing. http://www.tawileh.net/anas//files/downloads/papers/Software-Testing-Dynamics.pdf?download.

[Vea11] Dmitriy Vatolin and Dmitriy Kulikov et al. Mpeg4 avc/h.264 video codecs comparison. Technical report, CS MSU Graphics and Media Lab, Video Group, May 2011. http://compression.ru/video/codec_comparison/h264_2011/mpeg-4_avc_h264_video_codecs_comparison.pdf.

[Wil06] Laurie Williams. White-box testing. *White Box Testing*, 2006. //agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf.