Keyword Base Framework for DLNA Acceptance Testing

Bruno Filipe Guia de Sousa
bruno.sousa@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2015

Abstract

A successful acceptance represents the minimum operational functionality required by the customer to accept a software delivery, along with validation by use case studies. So far, acceptance frameworks focused on automating this testing type. We propose an acceptance testing framework based on a keyword model whereby each device is mapped into the network through a state machine and acceptance testing scenarios mapping are translated into a unique code, replacing the manual testing for an automatic approach. Results showed that the proposed framework can reduce the demand for manual testing in the acceptance process, hence eliminating part of the limitations associated with human operations, so far required in testing on a regular basis. However, the framework itself does not produce a fail or pass result. That result evaluation derives from the level of functioning achieved in the final user implementation test.

Keywords: Testing framework, Acceptance testing, DLNA, keyword driven model

1. Introduction

Testing is an expensive activity, counting up to 55% of a project cost [5]. In fact, failure to detect and correct defects in early states of a project can increase considerably its cost [3] and compromise the company reputation. Therefore, optimization of testing activities can lead to important savings on the overall project cost. Cost is reduced in two interlinked ways: by reducing the testing activities cost as a whole and by increasing success in early finding of any testing bugs, hence avoiding costly solutions for later issues. The core of the testing activities are test cases whereby a test case corresponds to a set of instructions and expected result. Combining several test cases originates initially a testing suite and ultimately a test plan which defines a set of test cases to be implemented. A frequent test plan required to be performed either several times per month or even per week, is a good candidate for optimization. Manual testing of the test plan can be tedious for the tester and time consuming for the company resulting in increased costs and increasing human error potential. Testing automation is a desirable solution as it overcomes manual testing limitations through operational optimization. Most of the work and research was conducted at ACCESS Europe GmbH1 offices in Oberhausen, Germany as part of my Software Tester position in the Quality Assurance department. The primary focus for the present work was to improve the acceptance testing procedure by developing a new automation framework, easily expandable and able to perform simple and complex workflow testing without overlapping with existing tools. Currently, most of internally testing performed by the quality assurance team is base on a called End-To-End (E2E) approach. In order to apply a distributed testing based on end user interactions, a manual setup of a test environment would be required. In a test environment and prior to testing, it is required to setup the network configuration, deploy the System Under Test (SUT) and Device Under Test (DUT) while ensuring that all configurations are correctly configured. Testing operations involve one or two testers that perform test cases as part of a test plan and report the results. Interactions between software sharing the DLNA norm and digital media is one of the main products. Currently, an automatic solution that controls all the different devices potentially involved and their interactions occurring at a human level and that simulates a wide range of end-user scenarios is unavailable. The present work aimed to develop a solution applicable not only to testing between network devices but also to additional scenarios and products.

1.1. Objectives

The main objectives underpinning this thesis was to develop an automatic testing framework solution that can save time and resources on the long run by

1http://eu.access-company.com/
creating templates, reusable codes and an easy to follow method for extending the solution. A successful testing framework would allow the release of resources from manual testing to others activities such as testing of other features not automated.

2. Background
Digital Living Network Alliance is a protocol for accessing and sharing multimedia content between devices developed by an organization with the same name. DLNA categorizes the products in Device classes, whereby each class has its own set of rules and characteristics. The main device classes are the following:

- DMS: correspond to the servers where the content including audio, video or images are stored and sent on request to another device class.
- DMR: are devices where the content from the server is displayed, like an TV.
- DMC: is a controller that enables the user to see a list of available servers and renders
- DMP: corresponds to a DMC with an embedded DMR
- DMPs: Is a printer using DLNA protocol and able to print images.

2.0.1 Use Cases
To ensure interoperability between devices, which could have differing manufacture origins, a certification is issued by DLNA after verifying each product through a program partially described in section 3. The certification is attributed only when there is proof of meeting the necessary requirements. DLNA provides a certification program, including tests against specific tools or testing with other already certified devices. Typical use cases of DLNA protocol can be summarized in three types:

1. Content playback from the server to a renderer:
2. Upload or download content from the server: Clients can upload and download content from DMS.
3. Printing content: Printers can be connected to the DLNA eco system and print content send by DMC or DMP.

2.1. Test Automation Techniques
According to various authors and studies [4], testing automation techniques can be divided in 5 large types: Capture-Replay Testing; Script Based Testing; Keyword Driven Testing; Model Base Testing. All of those which will be described in detailed in the next sub sections.

2.1.1 Script Based Testing
Script Based testing is supported on the idea of decomposing testing into small units and creating a small testing script for each unit. Once the scripts are implemented execution is done automatically. To create new tests, scripts can be combines among themselves and new tests can be created by changing the script execution order. The script base testing abstraction and quality depends in large part on the implementation used. Coverage is not easily extracted and needs to be manually compared to the requirements.

2.1.2 Keyword Driven Testing
Keyword base script are based on the idea of having keywords and additional info, and the keyword is mapped to scripts and library executing testing. Adaptation between the script and SUT can be achieved hence increasing the tool abstraction. Keyword-driven testing allows a high level of abstraction and can be automatically executed however coverage is difficult to measure.

2.1.3 Model Base Testing
Model base testing consists in the idea of representing the behavior or a sub set of this behavior of SUT into a written form. Further testing follows the model to ensure the correct behavior of the device. The model testing is generated by a test module generator that later is executed with the support of test scripts. Some authors indicate that the main model base testing advantage is the awareness of the system and coverage[1].

2.2. Continuous integration
Continuous integration is based on the idea of continuously getting the latest version of the source code and compilation to ensure proper working, on top of which tests can be executed. This practice ensures that the source quality is tested in early states and broken source code can be detected and fixed early in the project. According to several websites, Jenkins is currently the most used continuous integration tool, also the one used by ACCESS GmbH making it the tool we are required to have a good operational understanding.

3. Implementation
Proposed Testing Framework
The main result and outcome of the present thesis is a testing Framework called GenFramework. Due to the fact that this thesis is being conducted in a business environment, cost and time to market are both two critical criteria. If the solution development costs are higher than others available in the
market, the present research would have limited interest and application. Concerning time-to-market, given that shareholders (testers, programmers and managers) can be considered the main end-users of this framework, the shorter the time to deliver the tool results the better, otherwise this tool would only have internal use application. Negative feedback or delay in presenting results may result in project cancellation. GenFramework is required to provide easy integration and support of new hardware/operating system needs given that new operating systems and mobile devices may arise. Despite the fact that the SUT running hardware and software operate with minimum change and preferably in an invisible mode, the new element to be added should be integrated in the most transparent way possible. Once the setup and configuration is done, the development and maintenance of test cases should be executable by a tester without programming skills. This approach releases the programmers from this time consuming task. However, changes in the environment such as new platforms still require a programmer support. After support is added give, interaction between similar scripts should be applicable. In a distributed system such as DLNA operations are not always sequential. A common example is a server access, when different devices can perform operations in the server simultaneously. GenFramework should be able to control multiple devices simultaneously. When executing a test case all SUT present in the system need to be monitored and controlled. A failure may occur in any of the SUT and GenFramework needs to be able of detect individual SUT failures.

3.1. Solution overview
The idea behind the solution of GenFramework is using a script files as the one in listing 1. Each script file represents one test script that should be executed. The language used is high level and is not bound to any specific language. The technical requirements are the following:

- Web sockets modules : pip support many modules.
- Communication with Appium
- Communication with Selenium
- Support multi platform : Python is supported in Windows MacOS and Linux
- Integration with Jenkins
- Good community support (Subjective criteria)
- Support of reflection and dynamic properties

Using a keyword-driven framework approach testers only need to write scripts similar to example in 1. Lines 2, 3 and 4 represents initiation of devices, a DMC, DMR and DMS are created. In line 6 testing start by the created DMC playing a video from DMS in the the DMR. To make this possible the framework in background needs to perform a large amount of operations and more information is needed. DMS needs to have the content ; DMC needs to locate content and DMR needs to be on a ready state to get input. To improve testing quality the framework should have knowledge about current state of all SUT. This allow to better differing if some behavior or action is correct. State Chart extensible Markup Language (SCXML) is a XML language created by W3C [2] to represent event base state machines. DLNA device classes mainly only react to events, from other devices or users. SCML is presented as a as event base xml, this turns SCXML a good candidate to modeling DLNA components. The goal is only to use SCXML to help modeling the DUT, compatibility to SCXML will not be completed but only inspired in SCXML. This approach can hopefully help saving time by avoiding implemented instructions that are not needed in this content and not following SCXML convention completely. It is important to mention that we do not aim to fully support SCXML and/or generic implementations, but instead only use the concepts and operations from SCXML needed to better modeling DLNA systems. In the next sub sections some this tags and other SCXML concepts will be explained in more detail. An event can have multiple transitions, however only one can be selected. Every time a event match, condition will be evaluate. Only when the condition match the event is executed. When none of the events match the condition or name does not exist. Generic event called "*" can still be executed. Any event name can match with "*". However condition also needs to be verify. Transition are executed with a up down approach stopped in the first executable one. Non accessible transitions should not occur however they are possible if previous transition are always invoked. SCXML allow compound states which means that internally a state can have more states. Also allow parallel states which are
very similar to compound states, a group of state
tags have before a parallel tag and all children are
simultaneous activated. All onEntry and onExit
are executed. In the framework the implementa-
tion for simplification the path of a parallel state
need to be specify. In DLNA content, upload and
download are two example of functionality inside a
parallel state called streaming state.

DataModel is supported by SCXML supports in
his specification a data model embedded in the
SCXML machines. However we decided not to use
it. In alternative each SCXML will have an as-
soicate Data Model using the is own specification.
This option keeps the SCXML more clean and focus
on devices functionality only, testing specific inform-
ation are stored in different files and location.

3.1.1 New State Machine

When a generic line is read on the script a instance
needs to be created. A new process with all relevant
information is crated. this process should manage
the binary to deploy ( on request ). This info (lo-
cation of the delivery is passed to the process. The
binary is not running, needs to Wait for onStart()
instruction from GenFramework to start.

3.2. Solution Architecture and implementation

For a better understanding of the framework spe-
cific parts of the framework will be more detailed
explained. To achieve a framework as much generic
as possible some options had to be more complex
that expected. Allowing the framework to be more
generic as possible.

3.2.1 TestRunner

TestRunner is the center of the framework, is from
this module all framework is controlled. Once the
framework is invoked TestRunner receives the no-
tification with the information of the script under
test.

Every SUT will have is own separated process.
This allow the framework to control an unlimited
number of devices and control each one in a inde-
pendent way.

3.2.2 State Machines

This module create a state machine from SCXML
files. To keep this solution generic state machines
need to be construct on run time. To achieve this
state machine basic idea is to rely on list and dic-
tionaries to map the state machines. State ma-
cine modules is divided in two main blocks, the
generic block where a state machine is construct
and a concrete block were a state machine is ini-
ialized and use by other modules. To construct
the generic machine a folder defined in configura-
tion folder SCXML folder file should be there, for
example a DMS.scxml file will be the state machine
for a DMS machine.

3.3. Adapters

Adapters is the name of the components created in
the framework that will connect with the SUT in
order to interact with the SUT. The Adapters are
structured in the following way: Inside the folder
"adapters", device class folders should exist. A
device class name is obtain by the state machine
names. Inside the platform is expected more folder,
one for each implemented platform ( linux, win-
dows, and so on..) and a Generic folder. Inside
each folder a events.py file is expected. By default
the frameworks tries to access always first the plat-
form under test. If the function wanted does not
exist framework tries then the generic platform.

3.3.1 Scripts

Scripts are .txt files containing instructions about
how tests should be executed. Ideally instructions
are mapped from already existing manual test cases.
The two main type of instructions supported by
this script are generic and concrete instructions.
Generic instructions are the creation of a new ob-
ject, representing a new instance of a concrete ob-
ject under test. The syntax should be "DeviceClass
Name Param" where:

- DeviceClass: Correspond to a existing state
  machine and a concrete instance of it will be
  created.
- Name: Name from which a concrete instance
  of a state machine will be refer.
- Param: Extra parameters, each device class or
even external conditions dictates the parame-
ters passed. A unlimited number of parameters
are supported.

Once devices are created they can be invoked,
this is refer as concrete operations, to invoke them
the used name should be use. This option make the
script more readable and helps user to know with
device is interacting with. The syntax for concrete
operations is the following:

- Name: Name from which a concrete instance
  of a state machine will be refer.
- Operation: Correspond to a existing state ma-
  chine and a concrete instance of it will be cre-
  ated.
- Param: Extra parameters, each device and op-
eration have is own parameters, tool support a
unlimited number of parameters.
Generic and concrete instructions can mix during the script, however concrete instructions can only be perform after generic instructions or it will be no device to perform testing activities. A third set of instructions can also exist on script, call configuration options to device specific rules in case of abortion or what to do when some event occur.

When a line of the script fails, the execution is aborted and the exception and/or error message is returned indicating also the line of the script where the error occur.

3.3.2 DataModel

Scripts to run need to have information, for example instructions such as "DMS myDMS linux" per si does not provide all information needed, beside that is a DMS running in a Linux machine. Extra information is needed, in this case would be at least the ip of the machine. Information can be send directly in the scripts "DMS myDMS linux ip=192.168.12.3". To keep the scripts more generic and avoiding modifications at each test run or environment modification a Data Model Module was created. All added parameters either in Generic or Concrete modules without allocation, it means, a=b, DataModel will be consult and string replace by the correct and complete information.

3.4. Communication between process and GenFramework

Having each SUT controlled by a StateMachine running in its own process may create communication issues. Action in one SUT may have consequences in another SUT. Using DMC to play a video on DMR changes DMR status, communication is requested to update DMR state machine. In some cases like the previous example of a video playing in DMR, the state machine only have a passive behaviour, should only update state and if some verification function is implemented may check if DMR is actually playing content. Reporting ok or fail to GenFramework. Several features are added to solve the communication issue. The first one is a share memory. Taking advantage of python multiprocessor libraries, a manager from multiprocess module is created and instantiate with a dictionary. This dictionary is not iterable so when is accessible by someone needs to know what is searching for. Each concrete instance should add a entry to the dictionary with the instance name. The procedure is not strict but is expected that the new entry with the concrete name of a instance is himself a new dictionary. Returning to the example from previous phase, now DMC to play a video in the selected DMR, can search in this dictionary for the concrete name of the DMR and is expected to have all information needed to interact with the DMR.

3.5. Modification to ACCESS DLNA testing

Once a generic based framework was obtained it was important to adapt it so as to enable its compatibility with ACCESS use. The adaptation includes modification of the framework to be integrated with current infrastructure and also the creation of the links to interact with the SUT. In order to evaluate the effectiveness of the proposed framework for acceptance testing in DLNA context a wide range of scripts were created. The selected tests incorporate features relevant to acceptance testing and have highly defined test cases. When possible only mature features should be used and at this testing stage, only very defined specifications already manually tested should be considered. This will facilitate the evaluation before deciding if the tool is compatible with its application in new and more complex cases where no manual test was performed. From analysis of the existing and performed test cases and plans a list of candidates were selected. A key list of features together will their test plans was identified:

- **Parental Guidance**: Restrict visibility of content to certain devices.
- **Download/Upload**: Download and upload of content either to DMC or DMS.
- **Content Playback**: Multimedia operations (play, pause, stop..) on audio video or images.
- **Discovery Services**: Visibility of devices on the network

3.6. ACCESS DLNA Testing Architecture

GenFramework must be incorporated in current ACCESS testing architecture, not added as a new system running separately which would increase costs and resources to assure its functionality. Current architecture for testing products is based on a continuous integration using a Jenkins solution. As can be seen in Figure 1, the Jenkins setup is controlled at the highest level by a master Jenkins which coordinates all the slaves where all specific binaries - Windows, Linux, iOS and Android - are compiled.

![Figure 1: Jenkins setup and organization](image-url)
Source code is stored in GitHub and fetched by Jenkins to compile. Once the compilation process is completed the testing can start. Pytest\(^2\) is the testing framework selected by ACCESS. Pytest tests code is stored together with the source code fetched from GitHub. Once invoked by Jenkins, a specific folders is analyzed recursively and run all python files with test tag identifiable by Pytest.

For DMS testing a server with content was added to the infrastructure and to provide content to the servers on request. An API set was created to interact with all builds in order to decrease the individuals interactions times. GenFramework does not directly interact with the builds created by Jenkins but only through a proxy API. This abstraction level boosts GenFramework implementation in Jenkins, simplifying interactions and implementation details. This API enables devices creation and specifies what devices are to be created and interact with other devices using an API.

As an alternative to this framework, API to the back-end of the devices is also available with communication following a similar procedure to the one from DLNA protocols. This keeps communication simple by allowing a back-end displaying similar information of the information sent to the network. This internal API is called Application Helper Library (AHL).

When using Jenkins and py.test the internal procedure does not allow to interact by using DLNA specification events that DUT sends and receives. The main reason for this decision was that using DLNA specification events, either as listening or by sending commands, could interfere with testing in some unexpected way - for instance adding more noise to the network whereby more events had to be processed - could lead to issues not directly related to the features undergoing testing.

3.7. GenFramework Integration with Jenkins

To integrate our GenFramework with Jenkins testing an interaction was made with pytest. Pytest is a testing framework written in python that generates reports and if needed can be integrated with JIRA to automatically create bug reports. In order to make GenFramework compatible only a configuration removing report generation flag is required and invocation is done as in example 2.

Given that the test run is performed by pyTest, the report module was removed from the framework itself so as to clean the code. Each test is run individually by pyTest. Instances of testing are created in the beginning of each execution and deleted at the end. pyTest simply runs all the tests with the specific tags, i.e., all py test files with a def test inside. Hence, the tests in the repository require modification ensuring that only the right tests are run by pytest. This was achieved within Jenkins by creating specific jobs for each feature we wanted to be tested, ensuring that for the script would only run inside folders where tests were required.

3.8. Test Bed

GenFramework only ensures testbed partially through the integration module that obtains the testing deliveries and ensures the proper configuration. However, this module depends on the implementation specifics and may depend of external tools such as Github to obtain source code or Jenkins to compile and generate a testing executable. Within DLNA testing, if inadvertent devices are in the network GenFramework may be unable to fail to detect them, since this is out of its scope but these devices may interfere with testing by originating false failures. In the scenario were GenFrameworks runs from Jenkins no testbed is needed. Testbed is a task performed by Jenkins and pytest, leaving GenFramework free to execute.

Originally, the framework had a module called simulated. For DLNA testing this element would allow creating any device class to simulate with the devices under test. However, during implementation it was concluded that this behavior could be encapsulated in the interaction module. Nonconformity test result due to test bed error (Same test repeated multiple times getting different results) is mitigate by Jenkins and structure approach.

3.9. Implementation workflow

To achieve an valid implementation, a set of steps is required. First, DUT needs to be identified in

---

\(^2\)http://pytest.org

Listing 2: Modification to run from py.test

```python
import AutomationFramework.
MainClass as MainClass

def test_myiot():
    mc = MainClass.MainClass()
    mc.run('IOT_Script')
```
order to create the state machines. DMS, DMR, DMC and DMP were identified. DMP from Linux and mobile have different features and therefore the need of being different. In Linux the download and upload folder can be specified and the DMS for each folder needs to specify if an upload is accepted or not, mobile version use only the pre-define folders. However, during the early stage DMC was discontinued and only equivalents of DMP were incorporated in testing. Another relevant aspect is that mobile versions of DMP have more options than the equivalent DMP for Linux. To accommodate for this, the machines states of DMP were scripted in two, one for mobile and another for Report required for parental guidance. Besides the continuous integration, when the framework is integrated with pytest and runs directly from Jenkins server, it should also work as a stand-alone application to run tests in isolated networks where pytest and Jenkins are not available. Parental guidance testing may use a DMS installed in a router when connection to outside is not available. Binary is already installed since this test is performed in an ad hoc way whereby binary installation is kept outside of the tool due to the effort involved in creating a solution to keep the test isolation but connected to external in order to obtain the needed binaries or event content for the servers.

3.10. Modification on the framework
ACCESS has its own testing automation system implemented and as such, in order to apply GenFramework to it adaptations are required. Two key scenarios were identified, integration with the ACCESS testing framework and ad hoc testing.

3.11. Parallel operations
Current test setup, base on unit and integration tests ran by Jenkins or testing using tools provided by DLNA focuses on testing at system level whereby all functions are tested. My aim in this testing was too focus on parallel and interoperability operations, Multi-support of parallel operations offers an unique advantage worthwhile considering concurrent operations between devices and not simple one to one testing.

3.11.1 Middleware interactor for DLNA Devices
For testing purposes, DLNA products software is required to be deployed in a wide range of hardware and software. Mobile operated systems Android, iOS and Windows Phone have their own specification, software and rules about how to start and deploy software. The enabling framework, called middleware interactor for DLNA devices (MIDD) runs as a server. MIDD was applied outside of GenFramework so that it can be used by any tool. This type of operation is required not only by GenFramework but by any other framework. Implementation and architecture of this software is out of the scope for this work, which focuses on how GenFramework interacts with MIDD. Testing in mobile, iOS, Android and Windows Phone presents important challenges due to limitations and specifications of these operating systems. New instances of DMS, DMC, DMP or DMC cannot be simply launched. Only one instance can run per device. Interactions with so many different interfaces may be challenging due to the different codes and tools required. Also, the mobile interface of our applications evolves fast making it difficult to assure feasibility. This problem was solved by developing internally a new framework outside Genframework to unify the interactions with mobile devices. An error: No Hardware available will be displayed if not possible to launch the device and aborts the test execution. Some of the fields can be optional and hardware can also be optional if any device that can do the test is considered and not a specific hardware. Frequently connected to the test environment are other manufacturers hardware and software, TVs with DLNA DMRs, boxes with DMR and DMS devices. Regarding the external devices, the Middleware Framework allows obtaining the IP from the device and a limited set of requested information. The abstraction level introduced at MIDM optimizes the GenFramework development time by presenting simplified interfaces. Once testing is completed, a release instruction of hardware should be sent to MIDD, or otherwise the device is marked as in use and cannot be used for testing by either other instance of GenFramework or other frameworks.
3.12. Parental Guidance
Parental guidance is a feature restricting content visibility to individual devices. The restricting levels and names vary according to the region of the globe, and we will use as of now the American levels: G (General Audience) to NC-17 were under 17 are not allowed to watch.

Parental Guidance are enforced in 3 different ways:

1. By default content levels whereby each DMS defines default levels for added devices and for added content. All new content added to the server, independently from the type is given a default level.

2. By default device level is a new device in the network detected by DMS. The added device does not need to interact with DMS to be added to the list. DMR are not subject to Parental guidance levels because they do not navigate through content. The parental guidance from the DMR is therefore equal to the level of DMC.

3. By DMS modification of the device and/or content levels. Inside DMS, folder levels can be altered and changes are recursively propagated if a given level is less restrictive than a new folder level. For instance if a folder is changed to level R, all G, PG-13 levels inside this folder and sub folders will update to level R, but NC-17 will remain unchanged, because is more strict than the above-level folder, which is acceptable and an expected result.

3.13. Download/Upload
Besides streaming, which is the default option for broadcast content, the user can also opt for download or upload content. Through a DMC the user can download a file from DMS from its local device or upload a file to DMS. From the technical perspective this is only one flag difference on the DMS side. Flags used to identify the operation in the protocol are different and DMS could not even authorize it.

When testing in simulated home networks and outside Jenkins where the infrastructure manages the interactions, GenFramework can run in a specific device and this issue is not handled in the Jenkins environment. Testing is only applied when using web services and browsing navigation and when available in MIDM. Other type of communication is are allowed but should be avoided to decrease the changes or testing interference.

3.15. External Devices
External device corresponds to all of those not manufactured by ACCESS GmbH and/or without access to special API. Attempting to incorporate these devices into the GenFramework context brings new challenges to the interactions and validation of results. The external devices work purely in a black-box testing context. Typically these devices are already on the market so during the development of GenFramework it is reasonable to assume that any issue - until proved otherwise - results from our products under development. Inside Jenkins testing, external devices are obtained using MIMD, a request is done when MIMD replies with OK.

3.16. Negative testing
Negative testing consists in doing something that is expected to fail in order to pass the test. This allows testing limit situations or non-expected input to analyze DUT behavior. Notice the follow sequence of actions: 1. DMC1 play audio DMR1 DMS1; 2. DMC1 stop DMR1 3 ; DMC1 pause DMR1 In a normal test, DMC1 would be considered a failed test since DMR1 reported failure to change state. However, behavior was the expected result since DMR1 was in a state which a pause cannot be an issue as no content is available. By knowing DMR1 state machine status, it would be possible for the framework to identify that in the current state of DMR1 a pause cannot be emitted. But this approach may not always guarantee an expected result. Due to implementation of parallel operations explained in chapter ?? it is not possible to ensure which operation will be executed first and the expected result. Testers constructing the test cases should avoid this ambiguity on the test cases. A negative tag is added to the second parameter and a keyword negative becomes a reserved word in the framework and should not be used to define anything in the scripts, otherwise potentially
triggering an unstable behavior.

3.17. Setting up the Framework and Adapters
The framework development and configuration processes were very time consuming tasks. The framework development was the most time consuming task with 12 weeks. The development stage also included the necessary adapters configuration for the testing framework code required to accommodate new features. The framework development time line was also affected by external factors such as: features requirements and changes, or difficulties in the API communication with the GenFramework and sometimes not defined since until middle or end of the project.

4. Results
During the development of acceptance testing for specific features, time was tracked for further analysis. In order to evaluate the efficacy of the proposed framework for acceptance testing in the DLNA context, a wide range of scripts was created. The selected tests should have features relevant to acceptance testing and very well defined test cases. If possible, features used should be mature and at this testing stage, should suitably reflect well defined specifications already manually tested. This procedure will facilitate the tool evaluation, before deciding its further application in new and more complex cases where no manual test was previously performed.

4.1. Testing
To verify the usability of the tool by testers without programming skills the tool was briefly explained and then testers were required to create test cases. Deployment attempts were made to deploy (as explained previously) however, configuration changes were too heavy to apply in every test run. During the development of acceptance testing for specific features, time was tracked for further analysis. The sub sector bellow contains the analyses of the results by category. Due to time restrictions, GenFramework was only applied in the Jenkins environment at full scale, all the other implementations in isolated network were just experiences not carry forward.

4.2. Discussion
Parental Guidance results shown the testing number increases when executed automatically since the majority of the tests were not possible to be executed manually (simultaneous download/upload and operations while download or upload). Number of bugs were smaller when manually executed. Difference is due to bugs be related not with functionality but with interface imperfections reported by tested while testing this functionality. Manual testing is finding different different bugs in this features. To catch the same bugs as manual testing interface needed to be tested as well and in the present that is not done.

Download and upload shown an increase of bugs findings when comparing to manual testing. Main reason to this fact is that download and upload can be done in parallel. Manual testing are by default executed by only one tester and parallel testing is hard when not impossible to execute. This also provoked the higher number of test cases when compared to manual testing, the combinations are now much higher. Verification of a successfully download or upload can be independently verify by analyzing the file after the operation concludes.

Discovery services testings in GenFramework are different from the manual testing executing. Automatic testing focus on detection of devices after connect and disconnect without notification. Manual testing beside this also focus in Ethernet connection on and off, router disconnects and other operations not yet implemented by the framework. Doing in a automatic way allow a combination of testing, such as verify if all devices are listed or not and lunch stop a large amount of devices.

Content playback when automated is one of the hardest features to testing due to validation if is actually working without human intervention to check. Current validation is done by back end API calls, meaning GenFramework software asking for the current state and trusting in the response quality.

4.2.1 ACCESS Testing Overview
Content playback is the most difficult element to test automatically because is difficult to know when it works. MCVT, the official DLNA development tool has the same issue, solving the issue with a request for manual import or requesting a predefined server port for the output. The API works, however interface issues where not identified if the internal state changes from Play to Pause but video continues to play. As seen in table ?? the range of testing increases significantly in this situation. We suggest that automatic executions run and manual testing should be made occasionally to implement a quality control that verifies the match between expected and real results. During testing, parental guidance rules had to be changed, forcing a framework readjustment at the PG levels. The number of combinations and complexity involved were exponential and almost impossible to do manually, however, automatic enforcement worked quite well. Discovery Services when doing automatically have more quantity, but manual testing of discovery is more focus on quality of testing since testing the same with small differences and Manual testing simply focus
in different areas of discovery services not yet focus by Content playback testing still requires improvements in the result validation. Current API approach does not offer enough accuracy and can lead to false negative or false positive results mining the trust in the GenFramework. Across testing of different functionalities was perceptible that interface itself needs to be tested. Either interface testing as a category of is own or improving current test to also test interfaces.

5. Conclusions
It is important to keep in mind that the goal inherent to the solution was not to replace manual testing completely as activities such as ensuring that look and feel are working as expected and reaction times, quality of items and free testing are all tasks potentially difficult to automatize. However, from a user point of view ensuring the functionality when an automatic test passes in a E2E test case does not mean that the manual user can identify issues not detected by the automatic test. However, automatization ensures in a regular basis that at least in some extent the test case still works and the solution is not damaged. The proposed goal of applying an automatic framework to map acceptance testing was accomplished, however, it does not replace manual testing completely.

The results from the use of GenFramework by testers without knowledge were discouraging since testers were confused and without knowing what to do. A better documentation of the framework is required, the tool principles and operational guidelines should be made very clear to the tester. Unfortunately, due the lack of resources testers could not use the framework for longer times and therefore tests scripts were developed together with the tool development.

6. Achievements
The overall number of bugs found was larger in automatic than in manual testing. However is important to register that the fact that some automatic tests categories reported less issues than the manual execution. However, this issue requires a deeper analysis. When considering the mobile testing, the automatic testing focuses on behavior only, and declaring issues when some functionality does not do the expected result. During the same process, the manual testing will report also glitches in the interface, sometimes even reporting non-related issues at all but for unidentified reasons, the tester executed further actions leading to a bug. When considering only the functionality feature and not the interface or secondary aspects of testing, manual test in general was equal or inferior to automatic testing.

6.1. Future Work
For some well-defined type of applications such as mobile (iOS, Android) or web pages the static analysis could be applied. From the state machine inferred information, data model and testing scripts would be generated automatically and run through GenFramework.

Some aspects of the framework can be improvement in the future, mainly extend SCXML supported languages, automatic generation of testcases and new interfaces. During the development of the code to interact with ACCESS solutions some impediments were identified that should be taken into consideration on a future work mainly the lack of graphical interface and IP configuration. Documentation still need to go one level up to improve the usability of the tool by testers without programming skills.

References