One solution for testing embedded DTV software on the PC platform

Branka Ševa, Đorđe Glišić, Uroš Jokić and Marija Jovanović

Abstract—**In an embedded device industry, applicable software is developed for a particular platform and device. Reusability, functional correctness, and quality control of the software are of great importance. The digital television industry is no different. Moreover, it requires compliance with device safety, security, and functionality standards. Compliance testing is often done with near-end products, as most functionalities require that all components be put together. Secondly, most development is done using target platforms that often lack tools and add significant delays in development. This paper gives one solution for testing the embedded DTV software on PC. The authors give a road map for developing testing environment to safeguard the product's quality. It allows early-stage testing by the development team and helping the QA team test the end product.**

***Index Terms*— automated testing, DTV, middleware test environment, python, OpenCV, tesseract.**

I. Introduction

In embedded devices, hardware capabilities vary in many areas. Available RAM, platform instruction set, supported peripherals, hardware accelerators, and dedicated specialized hardware blocks. On the other side, depending on the product or manufacturer, there are support variations, incomplete documentation, and very little support for the supporting development software packages.

On the other side, there is a problem with integrating third-party components. They may or may not come with the test suite or test application. In the case of open-source software, source code is available, but it was written for specific operating systems (OS), sometimes depending on unique OS features.

A common component for all devices is DTV middleware software. It grows with new requirements, new standards, etc. Testing is always pushed to the end product, verified against predefined sets of tests. The reason behind it is that many features depend on all components being put together, and it is tough to test partially completed software [1].

Additionally, suppose such a DTV stack is inherited from another source without a test suite. In that case, it is always commercially unjustifiable to spend engineering time preparing a test suite that will verify the DTV stack. Instead, it is pushed to develop the end product and confirm its functional compliance [2].

Commercially available solutions are focused on testing the end product. Depending on the solution, it may offer hardware compliance testing or functional testing. Some tools like Intent+ [3][4] offer automated and manual testing. Automated testing is accomplished using dedicated test suite applications. Suitest [5] offers visual preparation of tests. Other solutions provide general-purpose languages like Stb-tester [6]. Others provide APIs like black-box-testing (BBT) API from Intent+. They mainly focus on automating the remote controller, capturing the screen, recording audio, and processing it using a test suite.

As a result of described practices, software products' quality may be at a reasonable level, but the quality of the code may be poor. Reuse of already developed code is very inconvenient across projects. Feature development may slow down as maintaining code becomes more and more expensive. Products may suffer from bugs that have low repeatability rates and high severity. In such cases, black-box testing [7] is not suitable. It is necessary to implement white box testing [8] procedures.

Section two details a problem and describes the system's architecture. Section three explains the proposed solution and provides implementation details. Section four discusses the results. In section five, we conclude our work.

II. Problem statement

In the development stage, verifying a new feature is time-consuming. Platforms with limited hardware capabilities offer unique tools to write software to devices. It may need from 30 seconds up to 5 minutes to run the software. Often those platforms do not support hardware debuggers.

A typical application in DTV consists of the following components:

1. Application layer (APP)
2. Middleware layer (MW)
3. Hardware abstraction layer (HAL)
4. Platform-specific SDK (SDK)
5. Operating system (OS)

Platform-specific SDK is a set of libraries and APIs that provide access to platform hardware components and allows control over them. This layer and the OS layer are closed for the development team. Also, those layers are highly platform-specific, so they cannot be ported to other platforms without

Branka Ševa – RT-RK Institute for Computer Based Systems, Novi Sad, Srbija, (e-mail: Branka.Seva@rt-tk.com)
Đorđe Glišić – RT-RK Institute for Computer Based Systems, Novi Sad, Srbija, (e-mail: Djordje.Glisic@rt-tk.com)
Marija Jovanović – RT-RK Institute for Computer Based Systems, Novi Sad, Srbija, (e-mail: Marija Jovanovic@rt-tk.com
Uroš Jokić – RT-RK Institute for Computer Based Systems, Novi Sad, Srbija, (e-mail: Uros.Jokic@rt-tk.com.

considerable effort.

The hardware abstraction layer (HAL) provides a defined API [9] that exposes all necessary functions for upper layers (middleware and application) and abstracts platform devices and operating systems. It is implemented again with every new platform. It is common to have abstraction layers for every portable software and a test suite that verifies that the layer is ported correctly.

The middleware layer provides support for the DTV standard and is responsible for all functionalities in the DTV application. It consists of modules controlling hardware service change, acquiring information from DTV signal tables, maintaining program database, service lists, event information database, user interface engine, etc. Those modules are often interdependent. It is not simple to decouple one from the rest of the system and check their correctness using white box testing (e.g., unitary testing).

An essential component of the middleware layer is the conditional access system (CAS) or digital right management (DRM) system. It provides access to protected content. It is also a closed component that comes with the pre-defined test suite.

The application layer covers the graphical user interface and specific logic for the user interface. It is connected to the middleware layer and highly depends on it. Black-box testing mainly verifies this layer.

Architecturally higher-level components depend only on lower layer components. Key components that are developed are the application layer and middleware layer. Hardware abstraction layer API stays the same across different target platforms. We want to create a system that will test those two main components.

The goal is to prepare a software test environment that can support:

1. Functional tests as end-user
2. Scenario tests as end-user and operator
3. Monitoring and testing internal state
4. Code coverage

Functional tests cover black-box testing, where implemented features are verified [1]. Examples are video presence, audio presence, switching service, changing volume, displaying graphics, and event information presented. Besides core DTV tests, additional tests unique to the application have to be supported, like the position of some element on the screen, at the right time, for the correct period, etc.

Scenario tests verify DTV software in more complex cases. Those use-cases involve changing information in DTV tables signaling, new commands from the CAS/DRM system, or new data from other custom protocols that affect the device's state. Tests shell cover application responsiveness to the user interaction and user interface changes based on the system's internal state.

The monitoring system needs to monitor the execution and report critical situations. It should consist of a logging mechanism and software/hardware debuggers to automate the testing of internal states by inspecting calls to specific modules, APIs, and execution paths.

Code coverage gives insight into the test suite coverage of the existing code. If test coverage is low, it may mean that the test suite has to be expanded to cover some exceptional cases or that some source code is unnecessary occupying space (dead code). This work did not cover code coverage testing. Due to the complexity of this feature, implementation details are not covered in this paper.

The test environment defined would be capable of inspecting every module for its dependencies and behavior. Afterward, proper refactoring will allow white box testing (unitary testing, scenario testing).

III. Implementation

We decided to create a test environment to run and test DTV software on a PC. The reason behind it is to use current and future state-of-the-art tools. The first step was to port DTV software to the PC platform. It was done by porting the HAL layer. More details about it can be found in [10]. It supports working with actual transport stream data and makes DTV middleware fully operational. Compared to the commercial product, the only difference is that it does not support targeted CAS, as it is proprietary, and its libraries are only delivered for specific target platforms. Work is done to overcome this, using simulated CAS. Due to the complexity of this feature, implementation details are the subject of another paper and are not given here.

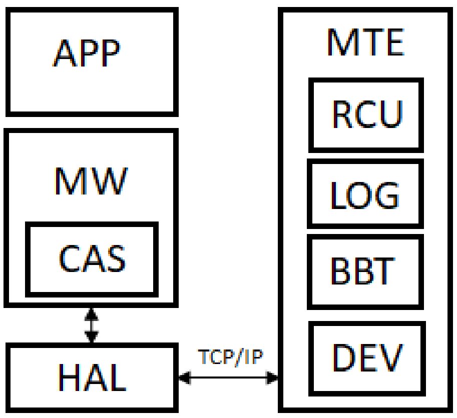


Fig. 1. Key components of DTV software running as part of the PC simulator are on the left side. On the right side are components of the MTE.

We decided to run separate processes for the test environment and DTV simulator. The DTV application runs stand-alone as it would be on the actual device. It allows us to have more options for the middleware test environment (MTE).

Communication with the PC simulator is done using TCP/IP. The communication protocol is designed to be minimalistic. The aim was not to disrupt the dynamics of the DTV middleware execution compared to its expected dynamics on the device. The protocol covers commands from MTE to PC simulator and data from PC simulator to MTE. Commands consisted of remote controller (RCU) events and requests for device state (screen capture, audio status, and similar).

We have decided to implement a test environment in Python language. We saw that this language is widely used in automation testing. Two STB automation test suites [4][5] already support Python scripting. It has extensive library support for user interface, computer vision, text recognition, communication protocols, etc. It is cross-platform, so we could design a tool to run on different platforms. It supports documenting code and a capable development environment (IDE).

We have selected the following frameworks to implement MTE:

1. wxWidgets - UI library (platform-independent, supports all major operating systems)
2. openCV - cross-platform library for computer vision, used for image manipulation and comparison
3. Tesseract - OCR engine for text recognition and extraction

The application was developed to support four different APIs:

1. Remote control API
2. Logger API
3. Black-box testing API
4. Development API

Remote control API covers control over RCU and sends commands to the PC simulator the same way a user would do using a remote control unit (RCU). To send commands, a TCP/IP protocol is used. On the side of the simulator, an existing module for receiving RCU input is adapted to receive TCP/IP commands from MTE.

Logger API is responsible for collecting log information from remote PC simulators using TCP/IP protocol. The existing logging module was improved to send log information over TCP/IP and the serial console on the simulator side. It supports filtering and searching for logging information.

Black-box testing API is a set of predefined APIs implemented on top of RCU API and an additional acquiring protocol for collecting screen output. It is aimed to be used for writing test cases. We selected to support the commercial black-box testing (BBT) API as part of Intent+. It was available to compare with the framework against an existing set of automated tests. Other solutions like Stb-tester API [5] are similar in API and exposed functionality.

Development API is created to support debugger integration in the MTE framework. It is implemented to support GNU GDB compatible debuggers. The framework can run the debugger and start the application or run the debugger and connect to the remote debugger server running the application (Fig. 2). This API makes it possible to start debugging software and send commands like setting breakpoints and watchpoints, printing values, etc. In this scenario, MTE spawns two processes, one for the GDB server that starts the PC Simulator and the second one for GDB used to control the remote PC simulator.

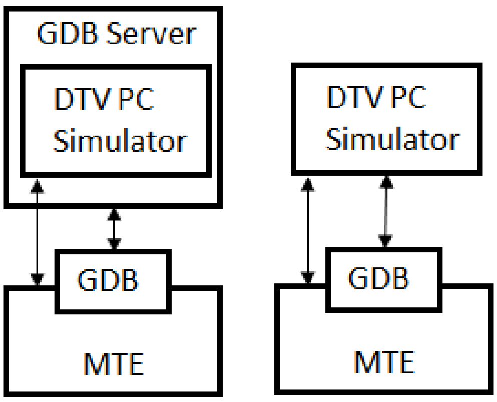


Fig. 2. Possible setups for running PC simulator using GDB debugging software with MTE.

Application consists of four parts similar to the APIs given above:

1. RCU controller
2. Stream controller
3. Logger
4. Test suite controller

Using an RCU controller, the user or developer can control the PC simulator using commands in the window that resemble the real RCU, as shown in Fig. 3.

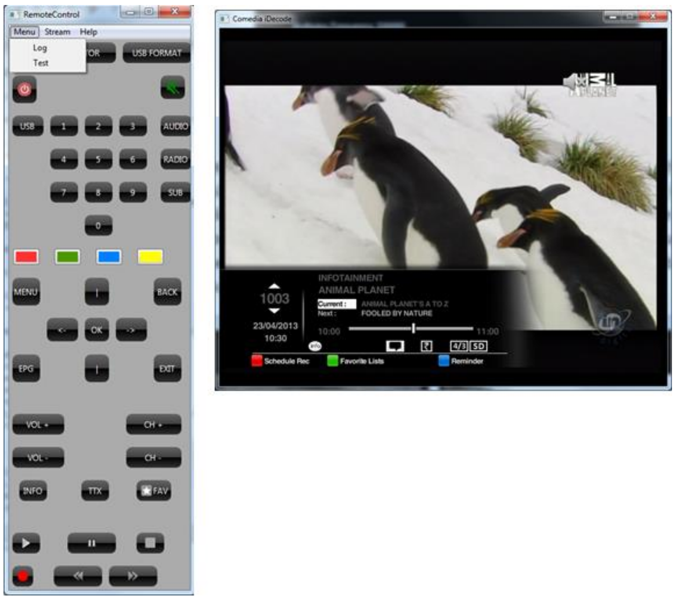
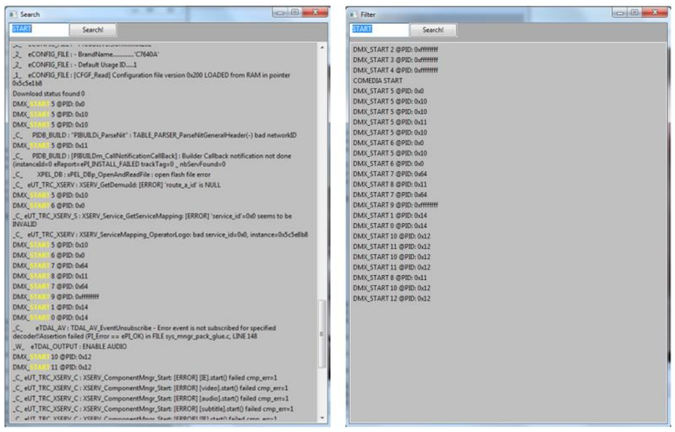


Fig. 3. Key components of DTV software running as part of the PC simulator are on the right side. On the left side are elements of the MTE.

The stream controller window is responsible for adjusting input DTV streams for the PC simulator. It allows setting stream files and broadcasting parameters.

Logger windows give information about logging data and allow users to filter and search for specific data in the log. The search pattern is highlighted in the log. In the filter window, only lines matching patterns are presented (Fig. 4).



```
_2_  eCONFIG_FILE : - BrandName..............'C7640A'
_2_  eCONFIG_FILE : - Default Usage ID.....1
_1_  eCONFIG_FILE : [CFGF_Read] Configuration file version 0x200 LOADED from RAM in pointer
0x5c5e1b8
Download status found 0
DMX_START 5 @PID: 0x0
DMX_START 5 @PID: 0x10
DMX_START 5 @PID: 0x10
_C_   PIDB_BUILD : "PIBUILDi_ParseNit" : TABLE_PARSER_ParseNitGeneralHeader(-) bad networkID
DMX_START 5 @PID: 0x11
_C_   PIDB_BUILD : [PIBUILDm_CallNotificationCallBack] : Builder Callback notification not done
(instanceId=0 eReport=ePI_INSTALL_FAILED trackTag=0 _ nbServFound=0
_C_     XPEL_DB : xPEL_DBp_OpenAndReadFile : open flash file error
_C_  eUT_TRC_XSERV : XSERV_GetDemuxId: [ERROR] 'route_a_id' is NULL
DMX_START 5 @PID: 0x10
DMX_START 6 @PID: 0x0
_C_ eUT_TRC_XSERV_S : XSERV_Service_GetServiceMapping: [ERROR] 'service_id'=0x0 seems to be
INVALID
_C_  eUT_TRC_XSERV : XSERV_ServiceMapping_OperatorLogo: bad service_id=0x0, instance=0x5c5e8b8
DMX_START 5 @PID: 0x10
DMX_START 6 @PID: 0x0
DMX_START 7 @PID: 0x64
DMX_START 8 @PID: 0x11
DMX_START 7 @PID: 0x64
DMX_START 9 @PID: 0xffffffff
DMX_START 1 @PID: 0x14
DMX_START 0 @PID: 0x14
_C_     eTDAL_AV : TDAL_AV_EventUnsubscribe - Error event is not subscribed for specified
decoder!!Assertion failed (PI_Error == ePI_OK) in FILE sys_mngr_pack_glue.c, LINE 148
_W_   eTDAL_OUTPUT : ENABLE AUDIO
DMX_START 10 @PID: 0x12
DMX_START 11 @PID: 0x12
_C_ eUT_TRC_XSERV_C : XSERV_ComponentMngr_Start: [ERROR] [IE].start() failed cmp_err=1
_C_ eUT_TRC_XSERV_C : XSERV_ComponentMngr_Start: [ERROR] [video].start() failed cmp_err=1
_C_ eUT_TRC_XSERV_C : XSERV_ComponentMngr_Start: [ERROR] [audio].start() failed cmp_err=1
_C_ eUT_TRC_XSERV_C : XSERV_ComponentMngr_Start: [ERROR] [subtitle].start() failed cmp_err=1
```

```
DMX_START 2 @PID: 0xffffffff
DMX_START 3 @PID: 0xffffffff
DMX_START 4 @PID: 0xffffffff
COMEDIA START
DMX_START 5 @PID: 0x0
DMX_START 5 @PID: 0x10
DMX_START 5 @PID: 0x10
DMX_START 5 @PID: 0x11
DMX_START 5 @PID: 0x10
DMX_START 6 @PID: 0x0
DMX_START 5 @PID: 0x10
DMX_START 6 @PID: 0x0
DMX_START 7 @PID: 0x64
DMX_START 8 @PID: 0x11
DMX_START 7 @PID: 0x64
DMX_START 9 @PID: 0xffffffff
DMX_START 1 @PID: 0x14
DMX_START 0 @PID: 0x14
DMX_START 10 @PID: 0x12
DMX_START 11 @PID: 0x12
DMX_START 10 @PID: 0x12
DMX_START 11 @PID: 0x12
DMX_START 8 @PID: 0x11
DMX_START 10 @PID: 0x12
DMX_START 12 @PID: 0x12
```

Fig. 4. Search and filter windows for logged information.

IV. Verification and Results

As a result of the following implementation, a test suite was created for commercial products using only BBT API (black-box testing). Test suites are grouped by the features they are testing. A list of all test groups and the number of tests are given in Table 1.

TABLE I
List of test suites prepared and results

Info channel	5 test cases	PASSED
EPG	6 test cases	PASSED
Genres	2 test cases	PASSED
Menus	10 test cases	PASSED
PVR	8 test cases	PASSED
Reminders	3 test cases	PASSED
Favorite lists	6 test cases	PASSED
Service lists	5 test cases	PASSED
Volume	6 test cases	PASSED
Service lists	5 test cases	PASSED
Zapping	5 test cases	PASSED

In the case of automated black-box testing, some graphical test cases may be challenging to create and prove reliable. User interface graphics blended with background video make it more difficult for AI-based engines to recognize certain visual elements' fonts, text, and shapes. Also, the comparison rate with expected images (shapes) may drop due to the background video. Our solution can compare video and graphical layers separately, resulting in higher recognition rates than blended image recognition using tools like OpenCV and tesseract. As a result, our MTE showed fewer errors than hardware running as part of the Intent+ solution.

Verification time was about 15 minutes, compared to manual testing, which will take 1-2h depending on tester skills. This allows developers to save considerable time when developing new features. Compared to automated hardware testing, execution time is around the same. It will enable continuous integration (CI) systems like Jenkins to repeat testing on selected changes.

V. Conclusion

With the proposed solution DTV application could be tested in the development phase by research and development teams or by dedicated QA teams. Automated tests written for MTE are usable for BBT devices in hardware testing, as they are written using the same API.

The essential contribution of this work is automated testing using software debuggers, where developers can inspect certain parts of the system multiple times and summarize information in reports. This type of testing can mimic unitary testing and complex scenario testing having internal systems state exposed for examination and reporting. It allows tightly coupled modules to be slowly refactored and isolated to introduce unitary testing and low-level verification.

Additionally, any other DTV software capable of porting to the PC platform could be tested using this MTE framework. It has to implement necessary features for that middleware and additional requirements to support communication protocol with MTE.

Further work could be done toward implementing support for CAS/DRM simulator or emulation. Also, it would be of great benefit to change DTV signaling from within the MTE application, as now it relies on signaling transported in DTV streams captured from live DTV networks. Another path for improvements is to add systems for code coverage and memory leak checks like Valgrind that could check applications in specific test scenarios as part of the automatic test.

References

[1] T. Tarkan, “User-driven Automatic Test-case Generation for DTV/STB Reliable Functional Verification”; IEEE Transaction on Consumer Electronics, vol.58, no.2, pp. 587-595, ISBN: ISSN:0098-3063, 2012
[2] Cabot Communications, “Automated testing of digital television devices“, accessed 2022, http://www.cabot.co.uk/solutions/robotester-white-paper/at_download/CB.pdf
[3] M. Kovacevic, B. Kovacevic, D. Stefanovic, V. Pekovic “System for automatic testing of Android based digital TV receivers “, INDEL 2014, Banja Luka
[4] Intent+, https://www.rt-rk.com/services/testing-centre, accessed May 2022
[5] STB Tester, https://stb-tester.com/, accessed May 2022
[6] Test suite, https://suite.st/, accessed May 2022
[7] S. Nidhra1, J. Dondeti, “BLACK BOX AND WHITE BOX TESTING TECHNIQUES – A LITERATURE REVIEW”, IJESA, Vol.2, No.2, June 2012
[8] I. Jovanovic, “Software Testing Methods and Techniques”, IPSI TIR, 2009
[9] G. Miljkovic, “DTV Linux Device Abstraction for Embedded Systems”, ISCE, ISBN:978-1-4244-6673-3, 2010
[10] A. Šuka, Đ. Glišić, M. Jovanović, “One solution of DTV simulator for PC platform“, TELFOR, 2019