One solution for simulating conditional access in DTV Software on PC platform

Milan Petrović, Đorđe Glišić, Uroš Jokić and Marija Jovanović

***Abstract*— Digital television (DTV) software runs on various hardware platforms, from low-cost low-performance devices to high-end devices that could compare with modern smartphones and PC configurations. The development quality depends on the tools available for the target platform. A new approach was taken to improve development by moving to the PC platform to avoid this dependency. The benefits are apparent, but it comes with some constraints. Typical examples are components available for target platforms but not PC platforms for security and legal reasons. One such component is the conditional access system (CAS) and digital rights management (DRM) components. This paper will present one solution to simulate conditional access (CA) in software without vendor CA libraries and support in hardware. The aim is to get the ability to test and verify various parts of DTV software that depend on CA functionalities.**

***Index Terms*— digital television, simulation, conditional access, DTV stack test environment.**

I. Introduction

A device that can reproduce digital television needs to comply with some DTV standards (DVB, ATSC, ISDB, etc.). Often it needs to support some content protection mechanism (encryption, signing, etc.). Additionally, the device needs to have a certain number of standard features and a few unique features dictated by the operator that will be available to the user.

In developing DTV software, specific components are delivered from third parties, like a software development kit (SDK) for the target platform or CA libraries for content protection. Content protection certification is an essential step in the development life-cycle, and DTV software is adopted according to the specification documents and APIs delivered. Upon development completion, the application is verified using several test suites that prove it behaves in the required way. This process repeats for every new target platform.

The DTV software development is tightly coupled with the target platform. Depending on the platform and its supporting packages, it may be impractical to develop a more complex project using them as a development platform. Instead, one way to overcome those difficulties is to develop on more suitable platforms. That platform should support at least logging mechanisms, the ability to re-write persistent memory, access to hardware debuggers, and good enough software packages to use those features. In practice, this is not the case, and almost always, given components are missing, and software packages are always behind the state-of-the-art counterpart packages available for PC. Selecting a more applicable platform instead of the target one for development is not applicable if the target CA library has different requirements (hardware or software) compared to its counterpart on a development platform.

For DTV software to be as robust as possible, there was a need to implement support for different CAS vendors. They shared core concepts for content access rights, content protection, operator box management, operator messaging to users, and other customized product and feature protections.

The CAS vendors' APIs significantly differ, although concepts are very similar. The differences between versions from the same vendor may not be compatible. Older libraries tend to have fewer restrictions, while newer versions have more demands and APIs to support, as new scrambling algorithms are added, and more security protocols are employed. It is necessary to have a level of abstraction in DTV middleware to adopt those changes and differences.

As a result, the first DTV simulator was developed on a PC platform [1]. It aimed to support the development of a graphical user interface. It became clear it could be used for implementing DTV middleware features as well. Those features were related to the DTV standard. To support it, the middleware test environment (MTE) [2] was created to test and verify different parts of the software on a PC platform using white box testing [3] [4]. This approach could not cover the code developed for a CA subsystem and the application code that was connected and dependent on that CA subsystem.

This paper aims to discuss paths that could be taken to overcome those obstacles. It gives one solution that is implemented and tested to prove the concepts. We could not find any relevant work on this topic. Closes to the work are discussions on testing approaches made in [5], [6], and [7].

Section two details the challenge and introduces the DTV system's architecture. Section three provides more information on the implementation and final solution. Section four explains verification and test results. Section five concludes the work.

Milan Petrović – RT-RK Institute for Computer Based Systems, Novi Sad, Srbija, (e-mail: Milan.Petrovic@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.

 ISBN 978-86-7466-930-3

II. PROBLEM STATEMENT

The CA vendor dictates two primary CA integration approaches depending on the target platform and selected operating system. If the target platform runs an operating system (OS) that does not support processes, only threads (tasks), the architecture looks as in Fig. 1. Here DTV software consists of OS, software development kit (SDK, drivers), hardware abstraction layer (HAL), middleware, and application layer. The application depends on middleware, and middleware depends on the abstraction layer (HAL) API that abstracts OS and SDK APIs [8].

The middleware and application layers contain all the business logic, whereas the remaining layers, like HAL, are porting layers designed to be very thin. Application is oriented toward user interface and feature logic, whereas middleware is oriented toward controlling hardware, supporting DTV standards, and interacting with CA subsystems.

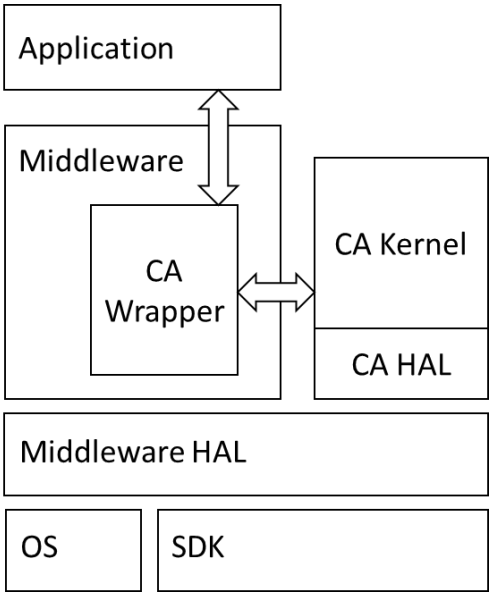

Fig. 1. Typical architecture of DTV software in case of OS where processes are not supported.

The module depicted as CA Wrapper is the actual module seen by the DTV middleware. It interacts with the middleware and application-level modules. It behaves like a proxy between the DTV stack and the existing vendor-specific CA subsystem. The conditional access subsystem consists of the kernel part where the logic is implemented and the hardware abstraction part (CA HAL) used as a glue layer between the CA kernel and underlying OS and SDK APIs. In this architecture, middleware HAL acts like a resource manager and has the information about allocated resources and tasks running in the system. This allows better resource management compared to the second approach.

The second approach is required with the OS supporting processes, like Linux and Android. As depicted in Fig. 2, the CA kernel and CA HAL depend directly on the underlying OS and SDK. They are running in a separate process. If the DTV stack is unstable or crashes, it does not affect the CA kernel. This approach ensures that the rest of the system never compromises CA. Still, the CA wrapper serves as a proxy between the CA kernel and DTV stack. It is up to the SDK vendor to ensure that multiple clients can access the same hardware peripherals. If not provided, some features like PVR may need to be carefully designed to ensure that components do not overlap in responsibilities.

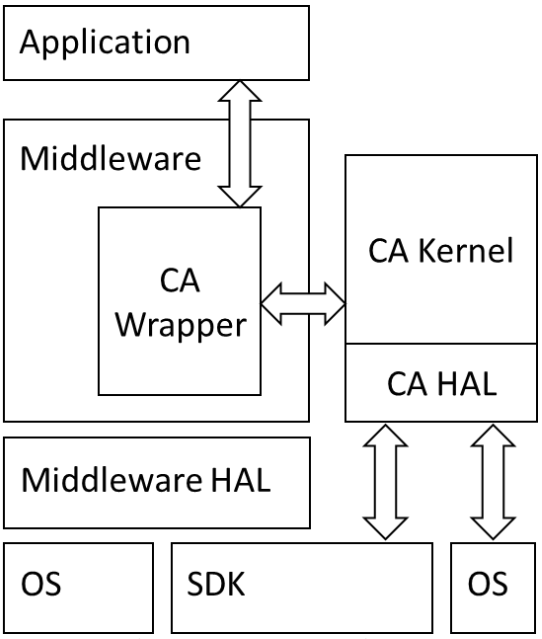

Fig. 2. Typical DTV software architecture in platforms with OS supporting processes.

We need to add CA subsystem support on a PC platform to test CA-related features. There are two possible paths:

1. Implement CA wrapper replacement module
2. Implement CA kernel replacement module (supporting CAS API)

The first solution gives us the ability to have a general CAS subsystem, irrespective of the actual CAS vendor. However, it puts aside CA wrapper code that interacts with the existing CA subsystem. Changes in the requirements of the CAS do not directly affect this solution.

The second approach is to develop the CA kernel module and the CA HAL module. It will preserve the CA wrapper module and allow it to be appropriately tested. However, this approach is considerably more time-consuming and has open questions related to all behaviors implemented in CA kernel API.

Our aim is not to implement content protection as software or hardware encryption. That is transparent to the middleware. Middleware only knows that content is protected and that the CA subsystem must start. CA subsystem is entirely responsible for the content decryption.

Encrypted content is never used in testing on PC because it has a complicated decryption procedure requiring specialized hardware protected by patents and legal documents. Only unencrypted content is used. This type of content can be generated using open-source tools like TS-duck [11] and video content available.

 ISBN 978-86-7466-930-3

III. IMPLEMENTATION

We have decided to take a hybrid approach given the above pros and cons. We implemented CA wrapper API on the DTV stack side as it already exists, allowing the remaining parts of the system to be unaware of the difference. CA kernel is partially shifted to the Middleware Test Environment (MTE). It is a framework for testing the DTV stack on PC.

The DTV software is running as a standalone executable. It has a middleware hardware abstraction layer (HAL) adjusted for the PC platform. Hardware devices are simulated in HAL using SDL [9] and FFmpeg [10] open-source libraries. The test environment is written in Python and communicates with the PC simulator using interprocess communication, particularly sockets. The test environment supported remote control, logging, and execution of automated tests. It can fully control the PC simulator, user input, and DTV stream input. Automated tests are supported by different APIs that are implemented in MTE. More about it can be found in [1] and [2] papers.

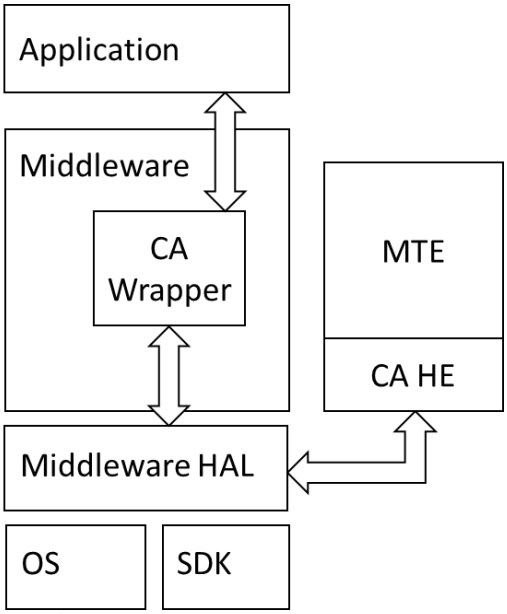

Fig. 3. DTV software architecture with middleware test environment (MTE) supporting conditional access head-end (CA HE)

As given in Fig. 3. the middleware test environment communicates with the PC simulator through the HAL layer that implements interprocess communication. A module CA wrapper uses send/receive routines from HAL. This is to mimic actual data flow, where CA information comes from a demultiplexer connected to the data stream. It will parse received commands and act accordingly. One typical example is the zapping procedure, where service is changed from one to another. In that case, middleware notifies the CA wrapper who needs to check access rights for that service in the database, sharing the data about the service being connected to and additional information about tracks to be descrambled. The module checks access rights in the database and responds to middleware. In our work, descrambling is not implemented, as it does not add any test value since all the descrambling is done in hardware, and none of that logic is done in the DTV stack.

Module CA wrapper is responsible for maintaining the CA kernel database. It exchanges data with the remaining parts of the DTV system. The middleware test environment can get the CA kernel database and modify it by sending appropriate commands. It communicates with a PC simulator using conditional access head-end (CA HE).

Following features (commands) we implemented in the CA HE subsystem and CA wrapper:

1. Device activation in a network
    a. Smart card
    b. Virtual smart card
2. Product access rights
    a. Checking rights
    b. Adding rights
    c. Removing rights
3. Service access rights
    a. Checking rights
    b. Adding rights
    c. Removing rights
4. Content protection
    a. Covered fingerprint
    b. Periodic fingerprint
    c. Permanent fingerprint
5. Mails
6. Changing service bouquet
7. Forced software update
8. CA notification messages
    a. Periodic messages
    b. Permanent messages
    c. User acknowledges messages

In Fig. 4, a window containing the setup for generating CA messages is presented. This CA HE submodule of MTE supports creating test cases. The test case is a message with all the parameters for that particular command. This way, the QA tester or developer does not have to enter test commands each time manually. Instead, he can select test cases saved as an XML file.

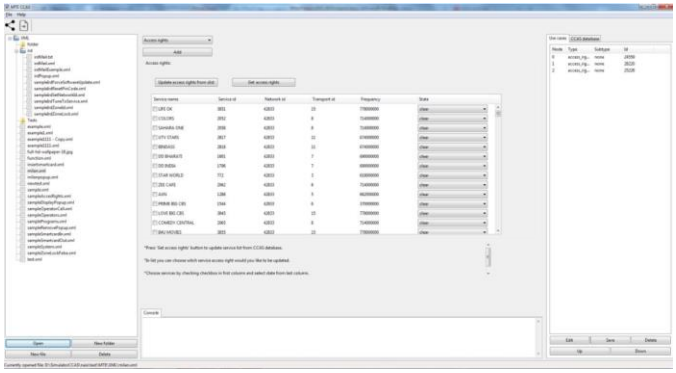

Fig. 4. CA HE main window consists of three parts, the user can re-use existing test cases or make new CA commands and send them in bulk.

CA HE main window consists of three parts, the left part reserved for displaying a tree of saved test cases, the middle

 ISBN 978-86-7466-930-3

part consists of a panel for generating CA commands, and the right part for listing generated commands ready for sending.

The middle panel for adjusting access rights for particular services is depicted in Fig. 5. There can be a list of services available on the box and the access right for that service.

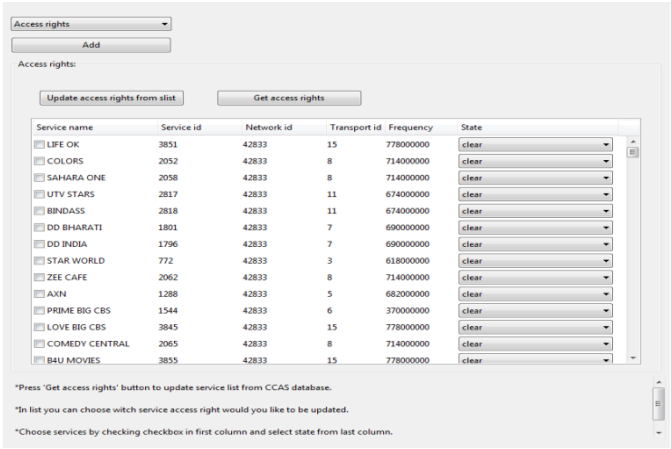

Fig. 5. The middle panel of the CA HE main window, where users select options to modify the service database's access rights.

The service access rights database is saved on the simulator side at runtime. The database can be exchanged between the simulator and MTE upon request sent from MTE. Once they are synced, MTE can send commands related to services access rights to the simulator. When the CA wrapper module receives a command, it processes the message, updates the database and saves it in an XML document. The CA state simulated in the CA wrapper can be restored from the XML file upon simulator restart. With this approach, the simulator is a standalone application, and MTE can communicate with it, but there is no dependency on MTE.

IV. Verification and Results

To test prepared CA subsystems, we have created a set suite that covers all supported types of messages that could be sent to the CA or received from the CA module by the DTV middleware [4]. We have observed that the code is executing correctly and that middleware behaves in the same manner as it is expected in the production environment.

In the case of sending chains of commands, we have observed new failure cases that were not covered by the DTV middleware and application. Those cases involve low probability cases like at the same time receiving a fingerprint message and a CA message. Those cases uncovered several combinations that could not be adequately tested on the development side, the operator's production live network or the lab network. They are not simple to prepare as a test case in those environments.

Scenarios that combine user interaction, CA signaling, and DTV signaling can reveal hidden bugs. Those bugs could be reported as software malfunction in a production. Yet those issues are impossible to reproduce manually unless the exact preconditions are known, which is rarely the case. Troubled combinations may be of low probability, but in networks with many end users, the chances that the failure will be seen and reported are very high. Still, the ability to troubleshoot it efficiently is very poor.

V. Conclusion

This paper focused on expanding capabilities for testing DTV software on a PC platform. It allowed more complex test cases to be executed that would be very hard or impossible to replicate in a network with real hardware. A further way of improving the solution is making a CA API on the MTE side. That could allow the creation of automated tests for testing application behavior as a response to CA events and user interaction.

Work could be extended toward implementing specific CA vendors' API allowing the whole DTV stack to be tested for required functionalities. It will increase the coverage of testable code to almost 100%. But gains versus cost ratio for doing this may not prove as an appropriate step. Another improvement can be made towards implementing some descrambling capabilities.

References

[1] A. Šuka, Đ. Glišić, M. Jovanović, "One solution of DTV simulator for PC platform", TELFOR, 2019
[2] M. Petrović, Đ. Glišić, M. Jovanović, "One solution for testing embedded DTV software on the PC platform", ETRAN 2022,
[3] S. Nidhra1, J. Dondeti, "BLACK BOX AND WHITE BOX TESTING TECHNIQUES – A LITERATURE REVIEW", IJESA, Vol.2, No.2, June 2012
[4] I. Jovanovic, "Software Testing Methods and Techniques", IPSI TIR, 2009
[5] 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
[6] Cabot Communications, "Automated testing of digital television devices", accessed 2022, http://www.cabot.co.uk/solutions/robotester-white-paper/at_download/CB.pdf
[7] M. Kovacevic, B. Kovacevic, D. Stefanovic, V. Pekovic "System for automatic testing of Android based digital TV receivers ", INDEL 2014, Banja Luka
[8] G. Miljkovic, "DTV Linux Device Abstraction for Embedded Systems", ISCE, ISBN:978-1-4244-6673-3, 2010
[9] Simple DirectMedia Layer, https://www.libsdl.org/, accessed May 2022
[10] FFMPEG library, https://ffmpeg.org/, accessed May 2022
[11] TSduck, https://tsduck.io/, accessed May 2022

 ISBN 978-86-7466-930-3