https://doi.org/10.5755/j02.eie.33222 ELEKTRONIKA IR ELEKTROTECHNIKA, ISSN 1392-1215, VOL. 29, NO. 1, 2023

Smart Multi-Agent Framework for Automated Audio Testing

Jelena Kovacevic[1, *], Uros Radujko[2], Miodrag Djukic[1], Teodora Novkovic[2]
[1]Department of Computer Communications, Faculty of Technical Sciences, University of Novi Sad, Serbia
[2]RT-RK Computer Based Systems, Novi Sad, Serbia
[*]jelena.kovacevic@uns.ac.rs, uros.radujko@rt-rk.com, miodrag.djukic@uns.ac.rs, teodora.novkovic@rt-rk.com

***Abstract*—With the widespread use of embedded software in consumer electronics, automotive industry, medical devices, and industrial environments, embedded software testing is gaining significance as an indispensable part of development and deployment of embedded products. With more than 20 years of research, development, and testing of various consumer technologies and products based on digital signal processors (DSPs) and advanced reduced instruction set computers (ARMs), we obtained insight into typical embedded development process and testing, and the pros and cons of various testing approaches and environments. In this paper, we propose the Smart Multi-Agent Framework based on IoT and Jenkins agents, customised for audio technologies in the Home Audio domain. We evaluated our solution on several complex immersive audio technologies implemented on a multicore DSP. Our uniform, customised, fully automated approach proved to be time efficient, error resilient, easy to replicate and use across all development, certification, and deployment phases of the product life cycle.**

***Index Terms*—Home Audio; Testing; Framework; IoT; Jenkins; Automatization.**

I. Introduction

For the past 10 years, the new generation of immersive audio technologies extended its presence in consumer space and therefore significantly increased complexity of home theatre (HT) equipment. The first generation of audio technologies is related to the reproduction of stereophonic sound on two loudspeakers, with limited frontal sound field [1], [2]. The second generation added a surround experience [3], adding left, right, and back channels (5.1 or 7.1 loudspeaker configuration). Finally, immersive or 3D audio technologies added height sound dimension, with loudspeakers above and below the listener [4]–[6]. A richer sound experience also implies higher bit rates (192 kHz sampling frequency, 32 bits per sample, up to ~50 Mb/s compressed bit rate), now, for the first time, comparable to video bit rates. Typical audio/video receiver (AVR) or sound bar (SB) software stack contains entire audio portfolio with spatial sound technologies as their main business driver: channel-based decoders, object-based technologies, virtualisers, various post-processing technologies (such as bass managers, dynamic volume control, parametric equalizer, and the like), and all kinds of custom-made audio which is providing differentiation between manufacturers. A typical example of the audio decoding and post-processing chain (Fig. 1 and Fig. 2) shows the multiplicity of audio scenarios within audio video receivers (AVR) and sound-bars systems (SB).

From the hardware point of view, HT products are based on digital signal processor (DSP) and/or advanced reduced instruction set computer (ARM) platforms, and they provide various connectivity options and sources: multiple high-definition multimedia interfaces (HDMI), different wireless interfaces (for audio streaming), optical inputs, USB, headphone jacks, and multiple surround and height effect speakers layout. In our previous work, we discussed complexity of such devices that comes from inner-connectivity issues and latency. The root cause comes down to the flow of the video signal, which must go through AVR - an audio processing and traditionally central component of HT [7].

The diversity of audio technologies and the variety of audio concurrencies and scenarios of usage imply that testing of such technologies and equipment is becoming more complex as well.

The biggest part of the HT audio software stack is protected by intellectual property (IP) rights, meaning that prior to reaching the market, each technology/product must achieve official certifications, done by IP providers. The goal of certification from IP vendor perspective (such as Dolby, DTS, Fraunhofer, or Dirac) is to ensure intended audio quality, end-user experience, as well as correct branding and usage of logos and trademarks. There are typically two levels of certification: IC certification (chip level) and product system certification.

IC certification means that algorithm implementation for a particular embedded platform is tested to ensure that it is functionally correct, that it meets qualitative acceptance criteria, can perform in real time, and that it implements specified application interface (API) correctly. IC certified technologies are used in final system integration, most often by original equipment/design manufacturers (OEM or

Manuscript received 16 November, 2022; accepted 17 January, 2023.

ODM). Before reaching the market, the product (containing several IP protected technologies) needs to be certified and approved by IP providers, but this time with a focus on integration of the certified IC library into the system and its interaction with the rest of the products' system. The certified product goes to the market after additional intensive in-house testing, such as spot and stress testing. During product lifetime, testing is performed to replicate and resolve all potential bugs, features, and issues reported by users.

Before submitting audio technology to IC and Product system certification, it is required to execute in-house certification (on the developer's site, often called "ready to certificate" testing) and to prepare the certification package containing the software and hardware to be assessed. After successful in-house testing, the certification package is sent to IP provider, and that starts the official certification process.

Years of supporting audio technology certifications with different IP providers and OEMs/ODMs showed that it is a complicated and a time-consuming process. The testing ecosystem, development board and appropriate software package, as a part of certification package, needs to be properly setup in IP providers laboratory, prior to official certification testing starts. That process, initial hardware and software setup, is often lasting for weeks due to documentation flows or not thorough reading of provided documentation (which is very often happening), different hardware/software setup, different equipment, inexperienced testers who are executing tests, various problems with drivers and software versions, software installations, and wrong assumptions. Our experience shows that high percentage of reported test failures are related to a bad HW/SW setup and not the technology itself.

Testing methodologies, processes, tools, level of test automation, and test hardware varies even within teams working on the same product (for example, software developers and application engineers rarely use the same tools and test setups). Off-the-shelf testing tools are seldom used due to specific home audio test equipment, price, and inability to cover all test phases.

This poses several problems. How can we simplify and speed up the certification process? How to ensure the same software and hardware setup on different premises, across different teams and companies (IP provides laboratory, product developers laboratories, ODMs, OEMs)? How to exclude all human-related errors? How to enable robust remote testing, as working from home becomes standard after Covid? How can we use the same testing ecosystem throughout the lifetime of the product?

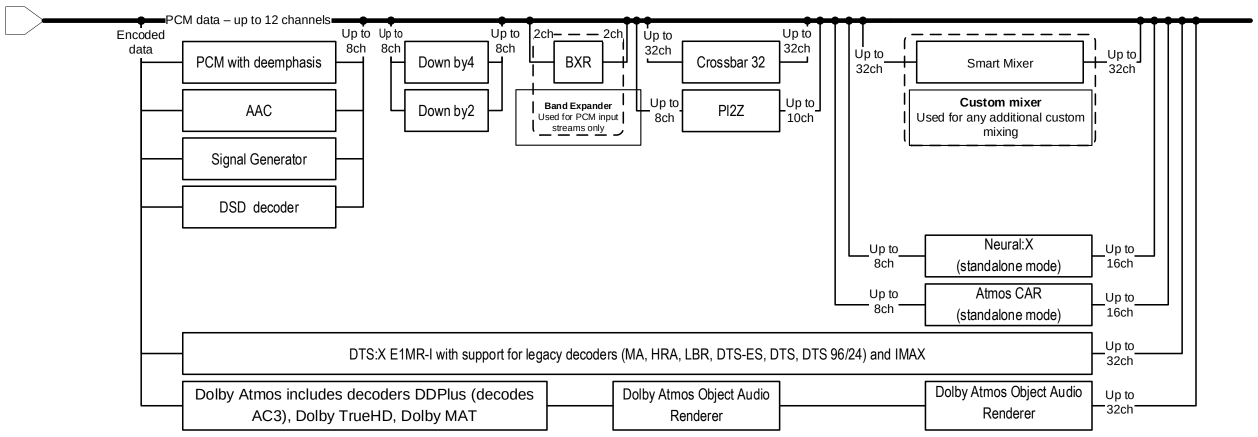
PCM data – up to 12 channels
Encoded data
PCM with deemphasis
AAC
Signal Generator
DSD decoder
Up to 8ch
Up to 8ch
Down by4
Down by2
Up to 8ch
2ch
2ch
BXR
Band Expander
Used for PCM input streams only
Up to 32ch
Crossbar 32
Up to 32ch
Up to 8ch
PI2Z
Up to 10ch
Up to 32ch
Smart Mixer
Up to 32ch
Custom mixer
Used for any additional custom mixing
Up to 8ch
Neural:X (standalone mode)
Up to 16ch
Up to 8ch
Atmos CAR (standalone mode)
Up to 16ch
DTS:X E1MR-I with support for legacy decoders (MA, HRA, LBR, DTS-ES, DTS, DTS 96/24) and IMAX
Up to 32ch
Dolby Atmos includes decoders DDPlus (decodes AC3), Dolby TrueHD, Dolby MAT
Dolby Atmos Object Audio Renderer
Dolby Atmos Object Audio Renderer
Up to 32ch

Fig. 1. Typical audio decoding chain in an HT system.

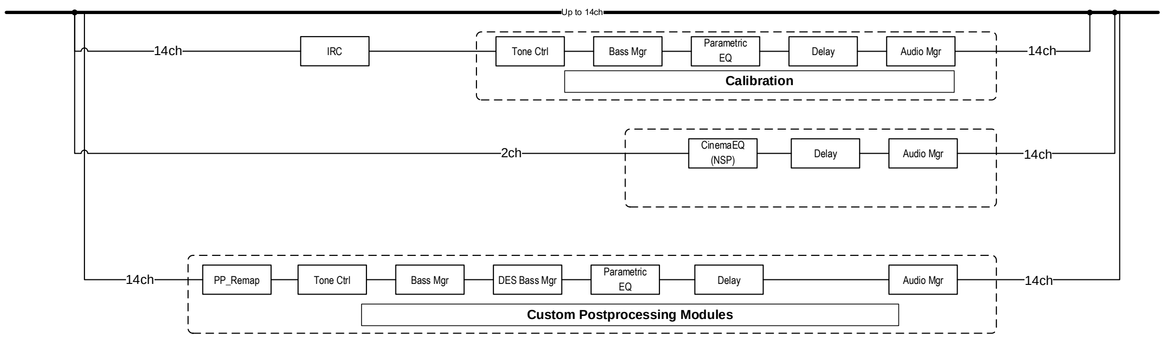

Fig. 2. Example of audio post-processing in HT system.

In this paper, we try to solve those problems by proposing a comprehensive, automated, highly efficient testing approach named "Smart Multi-Agent Framework for Automated Audio Testing", with intention to cover testing through the entire audio product life cycle. The proposed framework is as follows:

– *Easy to use* - enables on-click installation of complete software package (tools and technology to be evaluated) and simple hardware setup. This allows easy replication of the same testing ecosystem on different premises, which simplifies and accelerates the certification process significantly;

– *Comprehensive* - the same testing ecosystem can be used in development, certification, deployment, and product support phase;

– *Automated* - excluding human errors from both test execution and test setup;

– *Versatile* - enabling both local and remote testing with the same testing environment. Local testing is used mainly for unit tests on developers' computers, and the remote approach relies on dedicated test stations. Remote testing could be used for both internal testing and certification testing;

– *Hardware efficient* - highly efficient usage of all test stations. The same dedicated test stations could be used across all teams and development phases, 24/7;

– *Stable* - the framework enables stable over night/weekend testing based on smart plugs and IoT technology;

– *Scalable* - allows one to add tests stations and expand the testing throughput easily;

– *Time efficient* - this framework is highly customised for Home Audio, and it has no redundant or unnecessary feature.

The Smart Multi-Agent Framework is evaluated during the development and certifications of various immersive audio technologies on a 4 core DSP CS49844 processor [8].

II. RELATED WORKS

Testing is one of the last and probably the most important parts of the development of new product life cycle. In [9], Bajer, Szlagor, and Wrzesniak provide a description of the chosen aspects of testing software for embedded devices, describe the advantages of automated testing, give an overview of many different types of tests (black and white box testing, functional tests, integration tests, certification tests, etc.) that are used in embedded systems. Knowledge of the presented techniques is strongly recommended for all scientists who design software for embedded devices.

Automation has been one of the main drivers of testing: reducing human intervention meant not only predictable, reliable, and reusable results, but also important cost reduction and increased inter-vendor portability [10]. In [10], Portolan showed how several innovations in the fields of automated testing are slowed down or made unnecessary complicated due to legacy implementation choices, which are usually implicitly accepted and considered unavoidable. He introduced a more nuanced flow based on the experience of computer science, to be able to overcome most of the actual technological locks, and propose a solution better suited for future evolution.

In [11], Garousi, Felderer, Karapicak, and Yilmaz gave an overview of the current state of the art and practise of embedded software testing. The paper shows that the largest number of works in the field of testing refers to test execution, test evaluation, and test automation.

During the research, we focused on the related literature that seemed most relevant:

– Embedded systems testing;

– Framework for automated testing;

– Multi-agent testing;

– IoT Testing Framework.

To identify relevant studies and articles in our field of work, we used the following search terms: “Automated testing framework”, “Automatic testing”, “Multi-agent testing framework”, “IoT testing systems”, “Embedded system testing”, etc.

We found papers [12]–[14] that propose similar multi-agent testing frameworks, but none of them use Smart Plugs and IoT technology. Smart Plugs save a lot of time during testing, allowing testers to remotely troubleshoot problems in the event of a power outage or hardware lockup.

Multi-agent systems have been used in a variety of fields, such as transportation, healthcare, and manufacturing, to improve efficiency, flexibility, and scalability. These systems consist of multiple agents working together to achieve a common goal. In the context of automated audio testing, Multi-agent systems can be used to improve testing efficiency and effectiveness by dividing the testing process into multiple tasks, which can be performed simultaneously by different agents. Falco and Robiolo [15] proposed a systematic review of the literature to understand the progress of multi-agent systems from 2009 to the present. Agents are autonomous, proactive, adaptive, and aware of context. They often deliver the functionality through emergent behaviours that involve many agents. Thus, the correctness of their behaviours must be judged in the context of the dynamic and open environments and the histories they have experienced in previous executions.

The rapid development of communication and computer technology has accelerated the application of cloud computing. Cloud testing can refer to testing cloud-based systems (testing of the cloud) or to leveraging the cloud for testing purposes (testing in the cloud): both approaches (and their combination into testing the cloud in the cloud) have drawn research interest [16]. Bertolino *et al.* [16] have published a systematic review of the literature that covers these research directions of cloud testing.

Communication between the Gateway and Smart Plugs based on the ZigBee protocol will be described in Section IV. This WISE protocol was described in detail by Papp, Pavlovic, and Antic [17], and we will use it in our work for the mutual communication of smart plugs and gateway.

In general, related works show that there is a small number of works related to frameworks for automated testing of embedded systems based on IoT technologies.

III. OVERVIEW OF THE SMART MULTI-AGENT FRAMEWORK SYSTEM

Key components of the Smart Multi-Agent Framework are illustrated in Fig. 3. Each component will be described in more detail in separate chapters, but first a system overview is given, with a short description of the components and the way they are connected.

The heart of the proposed system is test station, a hardware and software ecosystem designed to cover all audio-specific testing. In addition to a regular PC, several specific hardware components are an important part of the station. Their purpose is to generate and deliver inputs and capture outputs of the device under test (Device Under Test (DUT)), as well as to control it. Test stations also require specific software stack to be able to receive and understand test requests, to properly execute required tests and control the mentioned specific hardware components, to assess and report results, etc. Some part of the software stack depends on a particular technology/product being tested, but hardware setup usually does not need to change.

All electrically powered components of the test station are connected to the power grid through Smart plugs. That way they can be automatically restarted, if needed. This feature ensures that a test station can always recover the test setup and continue its work.

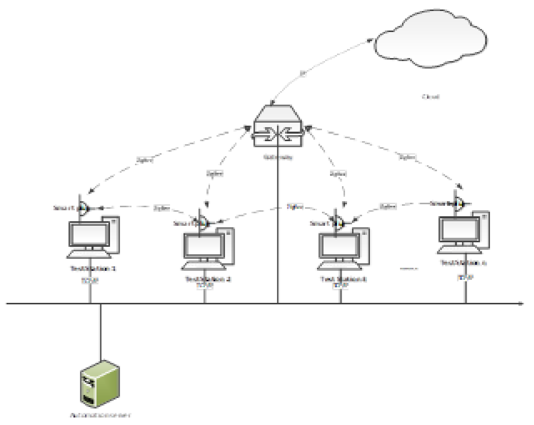

Fig. 3. Overview of the Smart Multi-Agent Framework System.

The test stations are connected to the Local Area Network (LAN) using Transmission Control Protocol/Internet Protocol (TCP/IP). That is the channel for reporting results and communicating with the automation server, which can send jobs to test stations. Automation servers can also perform load balancing when multiple test stations execute a single test group.

Smart plugs, on the other hand, are connected to the IoT network. It is based on the ZigBee protocol [18], has one central gateway, and multiple smart plugs are organised in the mesh topology. The plugs are controlled by the test stations through an IoT cloud service.

IV. TEST STATION

The Test Station is a central part of Smart Multi-Agent Framework. It is a testing environment tailored for testing of audio-based products. It includes both dedicated hardware and a specially designed software stack.

A. Test Station Hardware

The Test Station hardware setup is shown in Fig. 4, and it consists of the following:

1. *Computer (PC)* as main station;
2. *Device Under Test* (DUT): development board, evaluation board, or actual product such as AVR or SB;
3. *External sound card*. The sound card is used to deliver signals from computer to DUT and record digital outputs from DUT. It is usually custom made as it must allow the reproduction of compressed multichannel audio signals up to 192 kHz/32-bit resolutions. Likewise, this sound card must be capable of recording uncompressed raw data, which implies the possibility of recording more than 15-channel outputs. The Test Station is based on in-house developed audio grabber, named “Real Time Audio Grabber” (RT-AG) [19]–[21];
4. *Nvidia graphic card* is used to generate HDMI input, as some IP providers specifically rely on an application for input stream generation that is based on this card;
5. *Audio Analyzer* is used typically in System and Product Level Testing, for simultaneous analyses of analog multichannel signals. It is used to capture analog audio signals and provide HDMI input for DUT. We use Analog Precision APx585 [22], but only one Test Station has this equipment due to the APx585 very high price;
6. *Smart IoT-based power plug*: computer, DUT, RT-AG, and APx585 are all connected to the smart plug, which allows remote hardware resetting of the station.

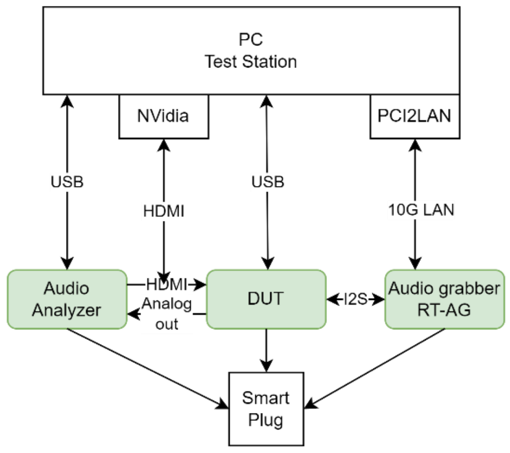

Fig. 4. Test Station hardware and interfaces.

B. Test Station Software Stack

Test Station software stack is designed in three software layers (Fig. 5):

1. *Application Layer* includes the following software components:

– *Jenkins agent*: The test station is designed for both local and remote testing. Installation of Jenkins agent (client) and adding station to the network allow connection to local Jenkins server (Automation server) and remote testing. Additionally, Jenkins server enables distribution of test cases on multiple Test stations;

– *Test execution framework:* Tool for the automated execution of test cases. It executes tests from the test plan sequentially. The test plan can be full test cycle coverage or partial, along with the technology-specific test environment configuration file, it makes input parameters for test execution;

– *Technology-specific test environment:* The software component of the test environment, result of the installation of the technology-specific test environment (explained in Chapter VII in more details);

2. *Middleware layer*: The middleware layer is the Test Executor Interface. The Test Executor will be described further in a separate chapter;

3. *Hardware abstraction layer*: This software layer includes all necessary low-level software: drivers for external sound cards (RT-AG), Audio Precision driver, SDK development platform, and Nvidia graphic card driver.

In some specific cases, audio technology environment may require installation of additional software: a specific version of the Python interpreter or tools for Unix-like system support for Windows (VM).

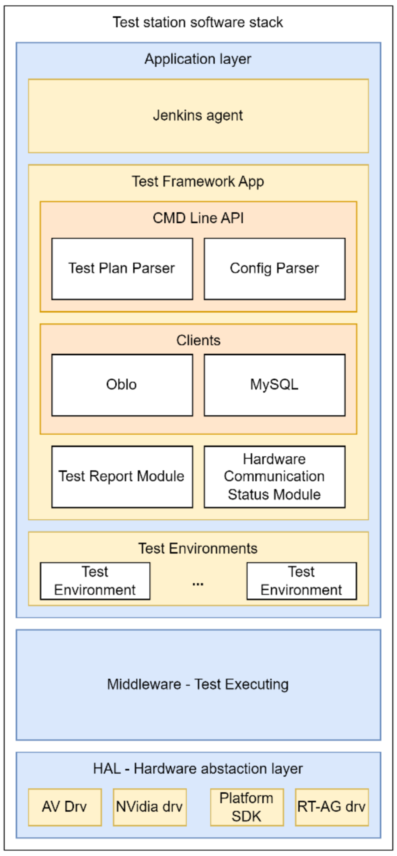

Fig. 5. Test Station software stack.

V. IOT-BASED SMART PLUG

If during a test cycle the DUT becomes unresponsive, it effectively stops the cycle. Usually, a simple restart of the device, followed by repeated test setup, is enough to continue running other tests in the cycle. The reasons that can cause the device to not respond can be various. There are some environmental reasons, such as power surge or something similar. For example, the device under testing is often on a development board, and it tends to be more sensitive to such things. Also, software that is being tested can fail in such a way that it leaves the device in an unresponsive state (by accessing unintended parts of memory, etc.). In any case, the goal of our test framework was to enable automated detection of those occurrences and automated recovery and continuation of testing. When these things are done manually, several hours or days of testing time can be lost. The recovery itself does take some time, but the main problem is that often when tests are being run (over night or during the weekend), there is no one available to detect the problem and do the recovery.

Smart Multi-Agent Framework uses the Oblo ZigBee Smart Plug [23] to turn off the power of the device and turn it back on, restarting it in that way. The plug is part of the cloud-based Oblo IoT environment [17] and therefore can be controlled remotely using the cloud API. That is why the cloud API client application is part of the softer stack on test stations.

When a test fails because the device is unresponsive, Test Executor, which runs on the PC of the test station, will send a command to the smart plug to turn off the power, then wait for five seconds, and finally will send the command to turn the power on again.

VI. TEST EXECUTOR - AUTOMATED TESTING TOOL

The Test Executor is a software tool for test case automation. It enables automatic execution of the entire test cycle.

A technology-specific test environment is prerequisite for test execution in Smart Multi-Agent Framework, and it consists of:

– *Test database* in the form of a CSV document, which represents a summary of all tests defined in a package and contains the following information on every test case: name of the test, input and output test stream; location of test folders, test streams/vectors; command-line parameters for configuring firmware; number of channels and sample rates, bit width, mode-specific configurations of source code;

– *Configuration file* represented in the form of the .ini file, defining technology-specific information such as tool for preparing input file, DUT configuration tool (e.g., number of input/output channels, sampling frequency), paths where other streams and tools are placed, etc.;

– *Test handler* (Technology-specific Test Executor interface) is a Python implementation of a pre-defined interface;

– *Installation files* - installers of any software which needs to be installed for successful test execution such as SDK, Python, Cygwin, etc.;

– *Load files* - flash boot image with developed audio library to be downloaded to IC;

– *Tools* - various tools for formatting streams, DUT configurations tool (enable/disable virtualisation (enhance for speaker or headphones), set reproduction channels, set dynamic range control, enable/disable upmix, etc.);

– *Evaluation tools* - evaluation tools provided by IP house.

Every *test case* consists of the following steps:

1. Fetch input stream;
2. Prepare input stream for external sound card - audio grabber playback;
3. Generate configuration file using configuration tool with parameters specified in test database;
4. Load image file with generated configuration file;
5. Play/Record stream;
6. Prepare recorded stream for evaluation: align referent and output stream, split recorded stream if needed;
7. Store recorded data in folder pre-defined by evaluation tool;
8. Clean up.

If necessary, test cases can be executed manually, via the command prompt. The Test Executor makes this process faster and more reliable, without inevitable human errors. Additionally, Test Executor as a tool allows automatisation of the complete testing process, not only execution.

A. *Test Executor Workflow*

To run Test Executor, the user must specify the path to the *Configuration file* that contains the test environment variables and tests to be executed during the test cycle. During execution, if the test is marked as FAIL or timeout expiries, the Test Executor will check the HW. If a test reaches a pre-defined timeout for the test execution, Test Executor will check the hardware status. If the hardware is unresponsive, the Test Executor will trigger a smart plug reset via the integrated IoT client application. The test case that reaches timeout will be flagged as failed. The Test Executor will restart the hardware and resume testing, following the test case list. In this way, the entire test cycle will not be lost due to one or more bad test cases. Test Execution workflow is shown in Fig. 6.

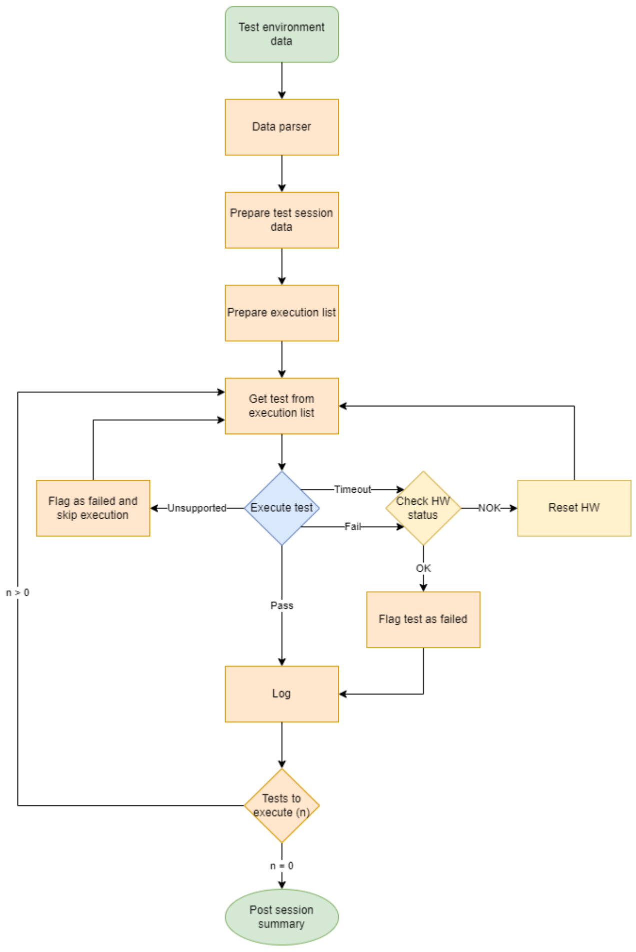

Fig. 6. Workflow of the Test Executor.

B. *Software Architecture*

The Test Executor is designed in three interconnected software modules: *Test Executor Framework*, *Test Executor Interface*, and *Test Executor Engine* (Fig. 7.).

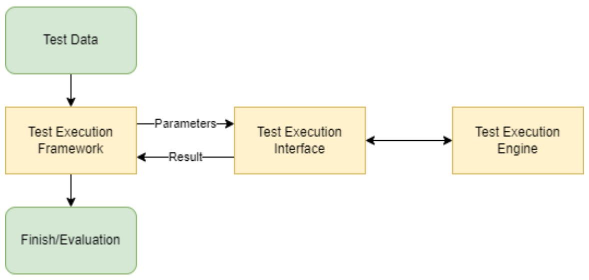

Fig. 7. Design of the Test Executor software.

The *Test Executor Framework* (Fig. 8) is a module that provides a command-line user interface, receives input data, and passes it to the *Test Executor Interface*. It consists of command-line parsers (test plan parser and configuration parser), web client applications (IoT cloud client and MySQL database client), test report module (for creating and managing log files), and hardware communication status module that communicates with testing hardware to determine if hardware is unresponsive.

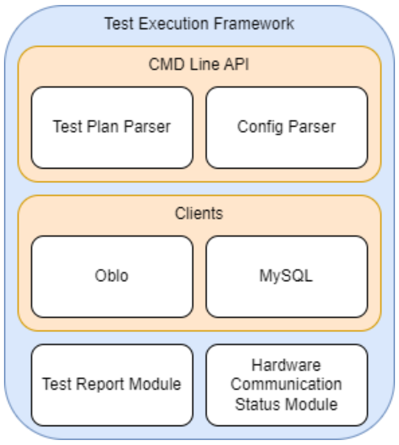

Fig. 8. Software modules for the Test Executor Framework.

The *Test Executor Interface* is a technology-specific implementation of test case steps. Every method of *Test Executor Interface* represents one step in the test execution flow. Implementation of those methods relies on pre-defined functionalities in the *Test Executor Engine*, which is the module that communicates with the hardware setup. The inter-module dynamic is shown in Fig. 9.

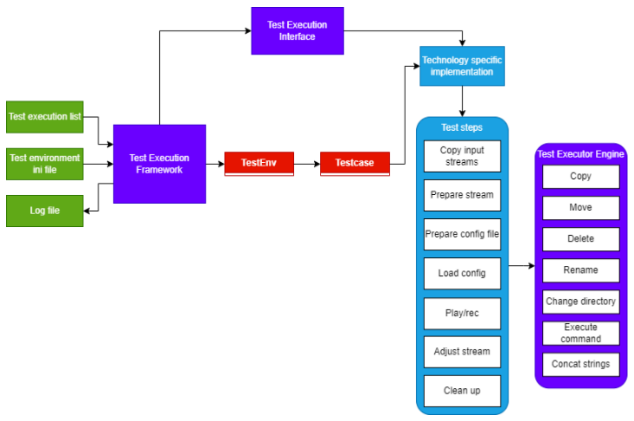

Fig. 9. Inter-module dynamic of the Test Executor.

VII. AUTOMATION SERVER

The Jenkins server [24] enables automation of software development related to building, testing, and deployment, facilitating continuous integration and continuous delivery. It is based on multiple clients' severe topology.

A. *Jenkins Job*

To execute tests on the Jenkins server, a defined and developed Jenkins job is required. *Jenkins job* is a set of sequential user-defined tasks. In the Smart Multi-agent Framework, one Jenkins job represents one test cycle: turning on and checking the hardware setup, installing technology-specific test environment, testing execution, test validation, and final evaluation of audio technology.

Within *Jenkins job*, every step prior to test execution is fully automated (Fig. 10). This automation enables testing during non-working hours and weekends.

The *Jenkins job* is executed on the Jenkins agent, which can be installed on a local computer or a dedicated Test Station. This enables local and remote testing. Once created, the *Jenkins job* can be called and executed across all development and certification phases. Additionally, the same *Jenkins job* can be used in both the deployment phase and the product support phase, as it allows for easy execution of *customised* test plans.

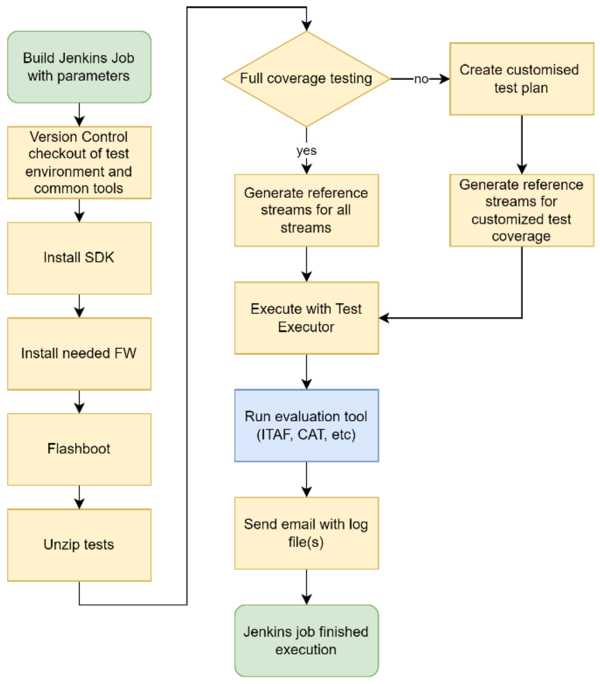
Build Jenkins Job with parameters
Version Control checkout of test environment and common tools
Install SDK
Install needed FW
Flashboot
Unzip tests
Full coverage testing
no
yes
Create customised test plan
Generate reference streams for all streams
Generate reference streams for customized test coverage
Execute with Test Executor
Run evaluation tool (ITAF, CAT, etc)
Send email with log file(s)
Jenkins job finished execution

Fig. 101. Jenkins job - fully automated test cycle.

B. *Multi-Agent Testing*

One test cycle comprises several test cases (sometimes it is up to a few thousand cases). The length of the input test stream (test vector) dominates the execution time of a test. Test streams for each test case differ in their duration: some are a few seconds long, and some last for several minutes.

To obtain the duration of each test case, after the first cycle is run, the test execution time is recorded and stored on the local MySQL server. This information allows for fair distribution and splitting of one test cycle on several Jenkins agents simultaneously. In this way, the time required for one test cycle could be significantly reduced. The main idea is to divide the entire test cycle into multiple test stations: the more Test Stations (Jenkins agents) are available, the less time is needed for one test cycle.

Currently, the test division algorithm is quite simple and is based on the number of available Jenkins agents and the execution time of each test case. Before starting a new test cycle, the Jenkins server triggers the MySQL client application to fetch execution time from the database. Then, it splits the test plan into several smaller tests plans of approximately the same duration. Each newly generated test plan subset is forwarded to the first idle test station.

This approach allows different test scenarios, e.g., we can divide only tests flagged as failed or tests executed on faulty test station to try at another one that works properly and the like.

Since all test cases can be executed independently, it is easy to utilise multiple test stations for a single test cycle. The amount of speed gain that can be gained in this way depend on how evenly the load can be distributed at the available stations. The execution time of the test case is linearly dependent on the input stream length, so this load balancing can be reduced to the multi-way number partitioning problem [25]. Furthermore, it is better that partitioning happens ahead of time, for all test cases (instead of dynamically sending one test case jobs to stations), because that way only one job per station is created, which reduces the communication overhead. That is why input stream lengths are recorded on the first test run and that information is used to partition test case set.

There are several algorithms that find the optimal solution to a multiway partitioning problem, but they are relatively complex and computationally intensive. In practise though, very often, much simpler, heuristic algorithms give good enough results. We rely on simple Greedy partitioning algorithm (also known as longest processing time algorithm) where test stream times are sorted in non-increasing order and then algorithm goes through the list and places current test to the partition that has the shortest cumulative length of test streams. Ideal partitioning will result in completely equal partitions and their length (size) would be the total length divided by the number of partitions. Of course, ideal partitioning is usually not possible, so optimal is one where the difference between the largest and the smallest partition is minimal. Greedy partitioning algorithm that our solution uses will never give a result that is more than 1/3 worse than optimal [26]. However, that is very dependent on the input value set, and in general large sets with smaller values tend to cause even smaller deviation of the resulting partitioning from the optimal one.

VIII. RESULTS

For purposes of testing the Smart Multi-Agent Framework, we used DSP evaluation board CRD49834 with DSP CS49834 [8], [24], both made by Cirrus Logic, Inc. (Fig. 11).

We evaluated our testing approach on several immersive audio technologies implemented on CS49834 DSP.

Fig. 112. Test Station hardware setup: CRD49834 DSP Evaluation Board and RT-AG.

Our first experiment was to compare manual vs. automatic system setup as a prerequisite for any kind of testing, including certification testing. When interviewing tester engineers about the time it took to set up the Test Station and install the certification package, the answer we received was “not long”. Further investigation discovered that “not long” is only happening when there are no problems with software versions, when the operating system is updated, when tools are already installed, etc., and this was rarely happening.

To objectively evaluate the proposed full testing automation, we performed full coverage testing of MPEG-H

3D Audio, implemented on four cores CS49834 DSP.

Testing was based on 1857 test vectors. Firstly, technology-specific test environment was created according to steps defined in Section VI. Then, we prepared certification package which includes both hardware and software needed for testing. Two identical Test Stations were used. The first one was set up manually, and the average time for set up was around 2 hours. During this experiment, test engineers were focused on their work (no coffee breaks, social media chats, or the like). After the setup, the Test Executor was manually called, as well as results evaluation.

On the other side, the full automated Jenkins job was run on the other Test Station. The results, including detailed time measurements of the Jenkins job steps, are given in Table I.

TABLE I. FULL TEST AUTOMATION VS. MANUAL TEST ENVIRONMENT SETUP.

Full test automation (one Jenkins job)	**Time**	**Action**	**Time**
Download CL SDK	0:00:01	Setup of the manual test station ~2:00:00	
Install CL SDK	0:02:10		
Prepare flash image	0:00:03		
Flash-boot	0:02:18		
Get test package from SVN	0:15:00		
Unzip tests	0:02:30		
Copy input streams to pre-defined location	0:02:15		
Execute tests with Tests Executor	11:32:58	Execute tests with the Test Executor 12:00:00	
Write to data base	0:01:55		
Evaluate results	2:52:00	Evaluate results	~3:00:00
14:51:10		17:00:00	

We performed similar testing with other technologies as well, with same results: automatic Test Station setup is much faster, excluding human error, and fully repeatable. Once developed, Jenkins job can be called and executed during the entire audio product lifetime.

In the second experiment, we wanted to measure the efficiency of our Smart Multi-Agent Framework on the entire testing time of complex audio technology, composed of several components. We performed testing with and without Jenkins automation on the component level (single component testing) and integration level (the entire audio chain testing: decoder, renderer, and post-processing). The number of test cycle that are run is the number of times that the full coverage test is performed during development, certification, and deployment of technology. The average for this technology is around thirty-six, but this depends on how complex technology is, how the project is managed, what is the coding and development style, etc. As can be seen in Table II, our approach saved time by up to 28 %.

In our third experiment, we wanted to evaluate multi-agent approach of our framework, and how efficient it is related to one cycle time.

In Table III, we show the resulting portioning for test sets of four different technologies. In each case, we partition into 1 to 6 groups, using the Greedy partitioning algorithm. We compare the longest partition to the average length (total length divided by number of groups) and an express relative deviation. At worst, the maximum partition length is 2 % longer than the average length. Therefore, the results confirm that real life test sets can be easily partitioned, and that liner speedup can be expected with the increase of the number of test stations.

In our final experiment, we wanted to evaluate Smart Plugs. Analysing all 220 complete test cycle runs, we determined that in about 7.7 % of the instances, the Smart plug was used to reset the system and continue testing. Almost all complete test cycle runs were done overnight or during the weekend. Therefore, if manual recovery was the only option, at least one full day would be lost in each case, a total of 17 days. This just gives a rough quantification of the usefulness of this approach, but it does consider that very often test cycles are run close to a deadline, so the loss of time is more critical.

Also, there are some qualitative benefits that stem from the fact that engineers can now work remotely.

TABLE II. COMPLEX AUDIO TECHNOLOGY TESTED WITH AND WITHOUT SMART MULTI-AGENT FRAMEWORK.

Technology	**Number of test cases**	**Number of test cycle runs**	**Cycle time manual env. setup (h)**	**Cycle time with Jenkins automation (h)**	**Entire testing time, manual env. setup (h)**	**Entire testing time with Jenkins (h)**	**Time savings**
Integration Level Testing Lossy Audio Decoder	1647	57	28	26	1596	1482	7 %
Component-Level Testing Lossy Audio Decoder	1674	51	9	6.4	459	326.4	29 %
Integration Level Testing Audio Transmission Decoder	1228	32	23.5	21.6	752	691.2	8 %
Component-Level Testing Audio Transmission Decoder	402	20	8.5	6.4	170	128	25 %
Integration Level Testing Lossless Audio Decoder	681	40	13	10.9	520	436	16 %
Component-Level Testing Audio Object Renderer	1192	20	7.51	5.4	150.2	108	28 %

TABLE III. PARTITIONING TEST SET BASED ON INPUT STREAM LENGTHS, USING GREEDY ALGORITHM.

Technology	Number of tests	Partition lengths	Number of partitions (Test Stations)					
			1	2	3	4	5	6
Integration Level Testing Lossy Audio Decoder	1647	Average (ideal) (s)	89232.5	44616.25	29744.17	22308.13	17846.5	14872.08
		Maximal (s)	89232.5	44620.9	29761.8	22360.8	17929	14980.1
		Deviation from average (%)	0	0.01	0.06	0.24	0.46	0.73
Integration Level Testing Audio Transmission Decoder	1228	Average (ideal) (s)	72309.6	36154.8	24103.2	18077.4	14461.92	12051.6
		Maximal (s)	72309.6	36164.4	24170.1	18129.3	14526.2	12089.5
		Deviation from average (%)	0	0.03	0.28	0.29	0.44	0.31
Integration Level Testing Lossless Audio Decoder	681	Average (ideal) (s)	36675.2	18337.6	12225.07	9168.8	7335.04	6112.53
		Maximal (s)	36675.2	18404	12325.6	9282.6	7461.3	6240.6
		Deviation from average (%)	0	0.36	0.82	1.24	1.72	2.1
MPEG-H	1857	Average (ideal) (s)	43110	21555	14370	10777.5	8622	7185
		Maximal (s)	43110	21557.6	14370.3	10786.8	8627.2	7190.3
		Deviation from average (%)	0	0.01	0.01	0.09	0.06	0.07

IX. CONCLUSIONS

This paper proposes a Smart Multi-Agent Framework for Automated Audio Testing, which utilises multiple agents to perform various tasks related to audio testing. The framework is designed to be flexible, adaptable, and scalable, making it suitable for use in a wide range of audio testing scenarios. Our results show that the use of this framework can help improve the efficiency and effectiveness of audio testing, reducing the cost and time required for testing, and improving the overall development process.

For our future work, we plan to minimise human involvement in testing process, as much as possible. First, we plan to enable runtime generation of test cases, currently developed offline. Then automate generation of technology-specific test environment, partially or fully, currently developed by test engineers. Finally, our research will be guided by feedback from IP providers, developers, system integration and test engineers who are starting to use our approach.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] R. Alexander, *The Inventor of Stereo: The Life and Works of Alan Dower Blumlein*. Routledge, 2000.

[2] J. Maes and M. Vercammen, *Digital Audio Technology: A Guide to CD, MiniDisc, SACD, DVD(A), MP3 and DAT*. Waltham: Focal Press, 2001. DOI: 10.4324/9780080494531.

[3] J. Herre *et al.*, "Spatial audio coding: Next-generation efficient and compatible coding of multi-channel audio", in *Proc. of 117th Conv. Aud. Eng. Soc.*, 2004, pp. 1–13.

[4] S. R. Quackenbush and J. Herre, "MPEG standards for compressed representation of immersive audio", *Proceedings of the IEEE*, vol. 109, no. 9, pp. 1578–1589, 2021. DOI: 10.1109/JPROC.2021.3075390.

[5] B. Van Daele, "The immersive sound format: Requirements and challenges for tools and workflow", *Int. Conf. Spatial Audio (ICSA)*, 2014.

[6] J. Herre, J. Hilpert, A. Kuntz, and J. Plogsties, "MPEG-H audio - The new standard for universal spatial/3D audio coding", *Journal of the Audio Engineering Society*, vol. 62, no. 12, pp. 821–830, 2014. DOI: 10.17743/jaes.2014.0049.

[7] N. Pekez, R. Celic, R. Peckai-Kovac, and J. Kovacevic, "Measuring audio processing latency for lip-sync purposes in DSP-based home theatre systems", in *Proc. of 2019 27th Telecommunications Forum*, 2019, pp. 1–4. DOI: 10.1109/TELFOR48224.2019.8971274.

[8] CS49834/44 description page, Cirrus Logic Corporation. [Online]. Available: https://www.cirrus.com/products/cs49834-44/

[9] M. Bajer, M. Szlagor, and M. Wrzesniak, "Embedded software testing in research environment. A practical guide for non-experts", in *Proc. of 2015 4th Mediterranean Conference on Embedded Computing (MECO)*, 2015, pp. 100–105. DOI: 10.1109/MECO.2015.7181877.

[10] M. Portolan, "Automated testing flow: The present and the future", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2952–2963, 2020. DOI: 10.1109/TCAD.2019.2961328.

[11] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz, "What we know about testing embedded software", *IEEE Software*, vol. 35, no. 4, pp. 62–69, 2018. DOI: 10.1109/MS.2018.2801541.

[12] S. Wang and H. Zhu, "CATest: A test automation framework for multi-agent systems", in *Proc. of 2012 IEEE 36th Annual Computer Software and Applications Conference*, 2012, pp. 148–157. DOI: 10.1109/COMPSAC.2012.24.

[13] J. Gao and Y. Lan, "Agent-based distributed automated testing executing framework", in *Proc. of 2009 International Conference on Computational Intelligence and Software Engineering*, 2009, pp. 1–5. DOI: 10.1109/CISE.2009.5366469.

[14] C. Zhao, G. Ai, X. Yu, and X. Wang, "Research on automated testing framework based on ontology and multi-agent", in *Proc. of 2010 Third International Symposium on Knowledge Acquisition and Modeling*, 2010, pp. 206–209. DOI: 10.1109/KAM.2010.5646259.

[15] M. Falco and G. Robiolo, "Tendencies in multi-agent systems: A systematic literature review", *clei electronic journal*, vol. 23, no. 1, 2020. DOI: 10.19153/cleiej.23.1.1.

[16] A. Bertolino *et al.*, "A systematic review on cloud testing", *ACM Computing Surveys*, vol. 52, no. 5, pp. 1–42, 2019. DOI: 10.1145/3331447.

[17] I. Papp, R. Pavlovic, and M. Antic, "WISE: MQTT-based protocol for IP device provisioning and abstraction in IoT solutions", *Elektronika ir Elektrotechnika*, vol. 27, no. 2, pp. 86–95, 2021. DOI: 10.5755/j02.eie.28826.

[18] P. Dhillon and H. Sadawarti, "A review paper on Zigbee (IEEE 802.15.4) standard", *International Journal of Engineering Research & Technology (IJERT)*, vol. 3, no. 4, pp. 141–145, 2014.

[19] N. Pekez, A. Popovic, and J. Kovacevic, "Performance analysis on TCP/IP audio streaming in point-to-point communication", in *Proc. of 2019 Zooming Innovation in Consumer Technologies Conference (ZINC)*, 2019, pp. 70–75. DOI: 10.1109/ZINC.2019.8769384.

[20] N. Pekez, N. Kaprocki, and J. Kovacevic, "Implementation of PC application for controlling RT-AG external sound card", in *Proc. of 2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, 2018, pp. 135–139. DOI: 10.1109/ZINC.2018.8448593.

[21] N. Pekez, A. Popovic, and J. Kovacevic, "Ethernet TCP/IP-based audio interface for DSP system verification", *IEEE Consumer Electronics Magazine*, vol. 10, no. 1, pp. 45–50. DOI: 10.1109/MCE.2020.2988616.

[22] Audio Precision Home page, Audio Precision, Inc. [Online].

Available: https://www.ap.com

[23] S. Jakovljev, M. Subotić, and I. Papp, "Realization of a smart plug device based on Wi-Fi technology for use in home automation systems", in *Proc. of 2017 IEEE International Conference on Consumer Electronics (ICCE)*, 2017, pp. 327–328. DOI: 10.1109/ICCE.2017.7889340.

[24] J. F. Smart, *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. Newton, MA, USA: O'Reilly Media Inc., 2011.

[25] R. E. Korf, "Multi-way number partitioning", in *Proc. of the 21st International Joint Conference on Artificial Intelligence*, 2009, pp. 538–543.

[26] R. L. Graham, "Bounds for certain multiprocessing anomalies", *The Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966. DOI: 10.1002/j.1538-7305.1966.tb01709.x.