Test Automation for NFC ICs using Jenkins and NUnit

Stephan Puri-Jobi
Corporate Software
ams AG
Unterpremstätten, Austria
stephan.puri-jobi@ams.com

Abstract— **This article gives a detailed overview of the setup of a test environment which is used for Near Field Communication (NFC) Integrated Circuits (ICs) at ams AG. The test environment is used for the verification and validation of the NFC ICs, as well as for pre-certification test runs, and is useable in two ways: (1) manual execution for every developer to be able to run tests on their desk before committing code changes and to reproduce failing test cases and (2) automated execution which is necessary for the Continuous Integration (CI) approach which is followed during development and to ensure that all tests are run against the Device Under Test (DUT). First a description of the system which shall be tested is given. Afterwards the used tools and methods to execute the various test benches are discussed. The hurdles which showed up during the process of setting up the environment for manual as well as automated execution are explained and the used solution is discussed.**

Keywords—NFC; test automation; Jenkins; NUnit;

I. Motivation and Introduction

There are several good reasons why to do automated test execution. The time which is spent during setting up the test environment is gained again during the firmware development. Every added test case is automatically executed and can strengthen the confidence in the maturity of the code. Regressions are found immediately.

Another good reason for doing automated test execution is that all steps necessary to execute specific tests are at least written down in some scripts which do the setup of the environment. The risk is quite high that preparation steps which are mandatory to execute tests successfully, are with the test engineer who actually does the tests. This is not documented because to him it is clear that it has to be done. When automating the test execution, every step has to be implemented, otherwise the test fails. By this it is guaranteed that every step is documented.

Also, reproducibility is an important advantage which is derived from test automation. An automated test environment allows to rerun tests with the exact same conditions several times. This can be very helpful for debugging of "random" failures.

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
13th User Symposium on Software Quality, Test and Innovation (ASQT 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

For this case study we wanted to get one step further. The goal was to also automate the execution of newly developed test cases. Adding test cases and getting them automatically executed is something you don't even have to think about when doing unit testing of a PC application, but for black box testing of an embedded device, this is something which can become tricky.

Whenever automated testing has to be done, especially when dedicated hardware is needed for these tests, some thoughts in the arrangement of the test cases should be put in advance. A bad layout of a test bench can make automated testing almost impossible, whereas a proper layout helps to avoid problems already by design. It all comes down to a good understanding of the system under test to find the most suitable way.

II. Challenges in NFC Test Automation

When designing a professional test system for hardware / software systems, one faces the following challenges:

(1) Different hardware setups: Testing a system like an NFC controller requires different hardware setups because the DUT can act in different roles. For example, depending on the operating mode of the DUT (tag or reader), it requires different hardware to communicate with.

(2) Automation: In order to automate the testing process, manual interference (e.g., the exchange of communication devices) must be avoided. This results in some controlling devices which need to be integrated into the test system in addition to the DUT and its communication counterpart.

(3) Professional test environment: A professional test environment delivers reports, failure logs, and trend charts with an easy to use interface (e.g., a web browser). Such elements already exist and shall not be re-implemented from scratch. Nevertheless existing solutions require adaptation to e.g., very specific hardware interfaces.

III. System Under Test

The test setup described was originally developed for the NFC ICs designed at ams AG [1], for example the AS3911 [2],

which is an NFC development kit, and the AS3953 [3], which is a Tag NFC interface chip (NFiC). NFC ICs have a number of different interfaces for the communication with the surrounding world. All these interfaces have a well-defined command set and use different well defined communication protocols. Using these interfaces the behavior of the NFC ICs can be controlled. Further, the expected behavior is clearly defined in different standards by the NFC Forum. Figure 1 shows three different use cases for NFC ICs. Within NFC enabled phones they can be used to exchange contact information (vCard). When detecting a Smart Poster, they are able to read out the content (e.g., an URL) of the embedded tag. In combination with a Payment Terminal the NFC IC handles the payment transaction.

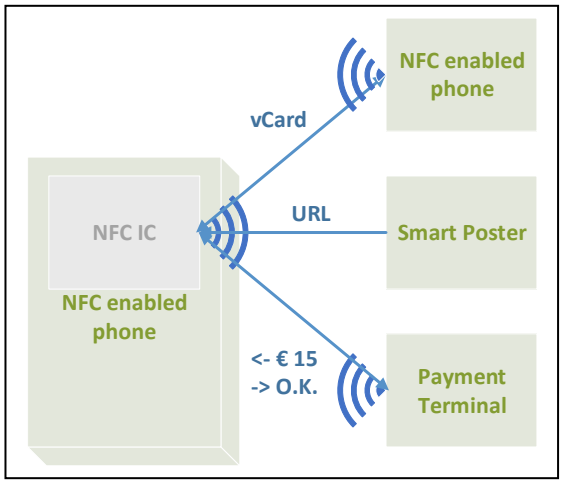

Fig. 1. Three different use cases for NFC ICs

The following case study is for the above described NFC ICs, but as already mentioned, any system with a well described behavioral model and which allows controllability and observability can be used.

IV. Hardware / Software Test-Environment

To be able to do black box tests of a system like our NFC ICs, we need to have full control over the inputs and outputs of the system. It is necessary to interact with the system in a defined way using the interfaces it provides. Specific hardware is needed to hook up to the hardware interface, which can be a UART or a Radio Frequency (RF) field. Also some software has to be in place to provide an Application Programming Interface (API) for the hardware. This API can then be used by the dedicated test benches to run test cases against the DUT. In the area of NFC there are already some test benches available, which test against specific standards. This is because the NFC Forum published a number of specifications on explicit test cases. Still there are some functionalities for which the NFC Forum does not provide test benches yet. In order to test these features, we built our own test bench. Also system, integrational, and interoperability tests were performed using our own implemented test bench to be sure to address all features of our system. Figure 2 outlines the test environment with the different test entities used for verification, and the physical access of the DUT.

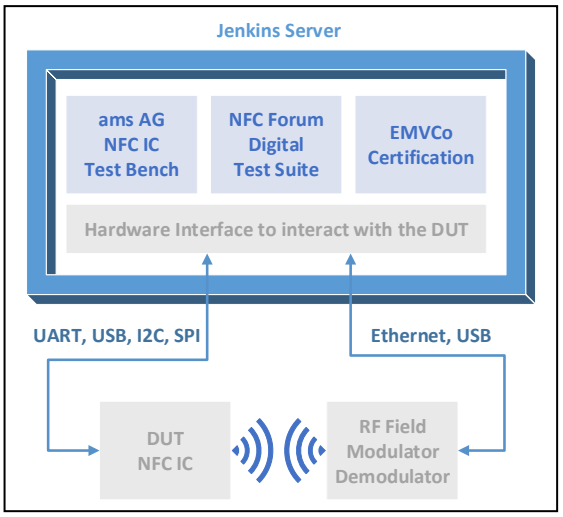

Fig. 2. Test Environment for NFC ICs

V. Test Execution by Jenkins

During test execution, a test bench which verifies the implemented functionality is run against the DUT. Test execution can be initiated manually by every developer at their desk to do a quick regression check before committing changes into the common code base, or for debugging purposes when a bug is found during test implementation. We also wanted to have test execution automated so that the code quality can be verified in the background. One requirement was to be able to run different tests at different points in time, depending on the duration of the executed tests. The idea was to have continuous information on the state of the firmware. This is achieved by executing a short running regression test set with every commit to the repository. Another requirement was to test the firmware in detail which takes quite long due to the rather slow RF communication and the high number of test cases. Jenkins [5] allows us to do this by offering a wide array of "build triggers". We can either start a build on a change in the repository, which is done for CI, or use time triggered builds like for nightly test runs, or test runs over the weekend.

Figure 3 outlines the test execution by Jenkins. Every "build trigger" is linked to so-called "Multi Job Projects". A "job" is a task, which Jenkins can execute, such as updating the local repository with the latest source code or executing the test bench in a specific configuration. Many jobs are repeated several times, often only with a different input parameter. This is the reason why we created a job for every step necessary for test setup and execution, and combine them in the way we need for a specific build trigger. The hierarchy of jobs is constructed in such a way that the last step of the defined job chain is executed by the build trigger. Every job knows its prerequisites to successfully execute. By this all prerequisite jobs are executed first, and then the originally specified job is executed. The advantage over starting with the first job and going down the chain is that it is easy to add an additional test by defining

its content and defining only the prerequisite job, everything else that is necessary will be executed in accordance with the prerequisite tree.

To be scalable and to outsource the maintenance of the Jenkins server to our IT department, we decided to use Virtual Machines (VMs) on which Jenkins can manage job execution. This enables us to instantiate additional Jenkins servers "in the cloud" without having hardware PCs standing around. Jenkins has load balancing already built in. To make use of this feature one Jenkins server is the dedicated master and controls the slaves. The master knows which jobs to execute and the point in time when this has to happen, and is then able to delegate the actual work to the slaves. This can only be done at job level as Jenkins is not able to split jobs into sub jobs on its own. So for load balancing, jobs need to be of a reasonable duration. This should be kept in mind especially when defining test run jobs. One large job which runs all tests at once is not suitable for load balancing.

Since we don't only want to do black box testing of a virtual simulation of our DUT but also want to run tests against the physical device, we need to access the actual hardware "out of the cloud". This is done using Ethernet2USB adapters which are connected to the virtual local area network (vLAN) of the Jenkins server environment. Via these adapters we are able to connect to our NFC ICs using USB2UART converters and access the hardware which does the physical interaction with the DUT.

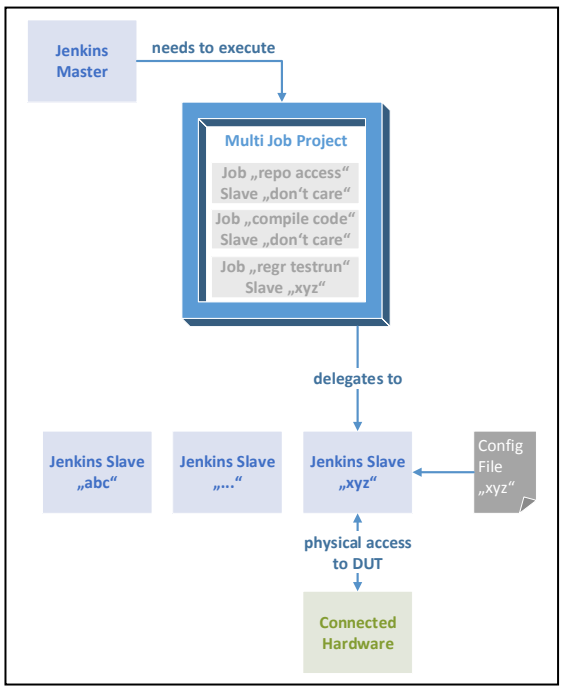

Fig. 3. Jenkins Master executing Multi Job Projects on different Jenkins Slaves depending on the required hardware configuration for the test run

To avoid problems with concurrent access of the same hardware by two different VMs, we decided to virtually connect one hardware setup to one VM. By this means, the different VMs have different capabilities according to the specific hardware connected to them. We restricted the execution of Jenkins jobs to dedicated VMs, which have access to the hardware required for the job. Information required for the specific hardware related setup is stored in configuration files which have the name of the host to which they belong to in their filename. By this, the test bench can parse its configuration file to get the correct parameters to access the hardware connected to the host it is running on. Depending on the purpose of the test set the correct configuration of the different hardware setups for the test benches is used.

VI. Black Box Testing using NUnit

To improve general test coverage and also to test specific features of our product, we developed our own test bench in addition to the existing third party test benches. For the tests which we had to implement on our own we used C# as the programming language and NUnit [4] as the test runner. This test bench is also used to automate the execution of the third party test suites. The implemented steps are: (1) the preparation of the environment, (2) starting of the executable, (3) cleaning up at the end, and (4) providing a standardized result file for Jenkins to have the test results integrated into the Jenkins Graphical User Interface (GUI).

For this test bench implementation we don't use NUnit in the usual way, meaning we don't use it for unit testing. We use NUnit as a test runner for our test cases. The reason is that NUnit provides some advantages which we also can make use of for black box testing.

First, we use it as graphical front end to our test cases. It visualizes our test cases and groups them according to the namespaces used during implementation and gives the feedback on passed and failed test cases in a nice way. It also generates a test result file which is analyzed by Jenkins to visualize the past test results and by this give a trend on the maturity level of the firmware on the DUT. The integration into Jenkins is very simple using an existing plug in. By this manual and automated test execution is very easy.

The implementations of the test cases are grouped according to the hardware they need to be executed. Test cases which need the same hardware and are testing similar behavior are combined into one test class. Thus the setup and teardown methods of NUnit can be used to initialize the necessary hardware. Tests which target similar functionality of the DUT very often require the same setup of the test environment. This might be a specific setup of the communication to inject protocol errors, or the simulation of a specific use case for system tests. NUnit executes several setup methods hierarchically before any test case is executed, and does the same with the teardown methods. The hierarchy is composed from the test case, the test class, and all namespaces. One thing to keep in mind is that, whenever an assertion is thrown in the test framework, you will end up in the next higher teardown method. This means that when you for example open a device in the test setup method, and an assertion is thrown because the open was not successful, the execution will continue in the test fixture teardown method.

The NUnit categories are used in our test environment to group all test cases which share the same hardware setup

necessary to run the tests against the DUT. Also the hardware platform or variants of the DUT which the tests are written for is defined as a category for NUnit. When executing NUnit, various categories can be passed and even combined logically. This helps to execute tests which are only possible for the given hardware which is connected. When starting NUnit we would, for example, pass the category for a specific variant of our product and the category which identifies test cases which expect a Type2Tag within the RF field of the DUT. Only those tests then are executed.

Another helpful feature is the ability to parameterize tests. Thereby the same implementation can be used for different configurations of the DUT. We use this feature for tests which do not rely on external changes of the test environment but only on the test bench. In our case the DUT has the possibility to activate different levels on the communication protocol. This feature is not suitable for changes of the communication interface since for this the test environment would also need to change.

To be able to use NUnit for black box tests we had to implement some adaptors to interact with the DUT. Since we don't use NUnit for unit testing but for black box tests, we need access to the hardware interface of the DUT. Therefore we developed a framework which provides an API to access low layers like UART, USB, or Ethernet, as well as higher layer helpers which provide an API to the command set of the test hardware.

VII. Results

The described test setup enabled continuous testing. This means a lot of tests are executed nightly. The development team can react immediately on fails which avoids bad surprises close to the release deadline. Currently we are running more than 10,000 tests every night. Figure 4 shows the Trend Graph which is generated by Jenkins for one test configuration.

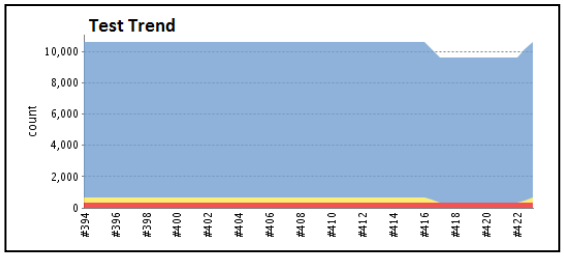

Fig. 4. Test Trend Graph by Jenkins

Test effort and cost reduced significantly. The execution does not involve manual effort and one entire test run is done overnight. Manual testing would require a lot of configuration steps which would result in a duration of two to three days for one test run. In numbers this would mean that for a manual test run which would take about 3 days (equals 24 hours), the cost is about € 2,400 per test run. Considering one run per week over a development duration of one year (50 weeks) shows us savings due to automation of € 120,000.

Seamless integration of new tests into existing test execution is achieved. New tests which are added with the proper attributes are automatically added to the next test execution run and are automatically executed using the correct configuration of the hardware setup.

VIII. Conclusion

For ams, the described approach definitely elevates the quality of software. We are able to monitor the maturity of the code during the development and identify regressions almost immediately. This reduces the time needed for debugging because the developer is still in the context of the specific part of the code when the regression pops up.

Redesigns and restructuring of the firmware can now be done with significantly lower risk of having broken functionality at the end or introducing backward incompatibilities. Applying this methods, a new RF front end was already successfully integrated into the firmware, and the firmware was ported to a different microcontroller.

Due to the success we will now adapt this test technique for other projects within ams and will have a closer look on different test methods to even further improve our quality.

References

[1] ams AG [Online]. Available: http://ams.com/eng

[2] AS3911 [Online]. Available: http://ams.com/eng/Products/NFC-HF-RFID/NFC-HF-RFID-Reader-ICs/AS3911B

[3] AS3953 [Online]. Available: http://ams.com/eng/Products/NFC-HF-RFID/NFC-HF-Interface-and-Sensor-Tags/AS3953

[4] NUnit 2.6.2 [Online]. Available: http://nunit.org

[5] Jenkins 1.599 [Online]. Available: http://jenkins-ci.org

[6] M. Pezze, and M. Young, "Software Testing and Analysis: Process, Principles and Techniques," Wiley; 1 edition (March 27, 2007).

[7] N. G. Leveson, "Safeware: System Safety and Computers," Addison-Wesley Professional; 1 edition (April 17, 1995).

[8] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, "The Art of Software Testing," Wiley; 2 edition (June 21, 2004).

[9] S. Grünfelder, "Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter," dpunkt.verlag GmbH; Auflage: 1., Auflage (29. Mai 2013).

[10] U. Vigenschow, "Testen von Software und Embedded Systems: Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen," dpunkt Verlag; Auflage: 2., überarb. u. akt. Aufl. (22. Februar 2010).

[11] A. M. Berg, "Jenkins Continuous Integration Cookbook," Packt Publishing (21. Juni 2012).

[12] R. Osherove, "The Art of Unit Testing," Manning Publications; 1 edition (July 8, 2009).