UTTOS: A Tool for Testing UEFI Code in OS Environment

Eder C.M. Gomes[1], Paulo R.P. Amora[1](✉), Elvis M. Teixeira[1], Antonio G.S. Lima[1], Felipe T. Brito[1], Juliano F.C. Ciocari[2], and Javam C. Machado[1]

[1] Department of Computer Science, Federal University of Ceará, Fortaleza, Brazil
{eder.clayton,paulo.amora,elvis.teixeira,gerbson.lima,felipe.timbo, javam.machado}@lsbd.ufc.br

[2] Hewlett-Packard Inc., Porto Alegre, Brazil
juliano.ciocari@hp.com

Abstract. Unit tests are one of the most widely used tools to assure a minimal level of quality and compliance during development. However, they are not used in many projects where development takes place at low-level contexts. The main reason is that unit test development itself demands more time and becomes expensive in this context and tools that assist test creation are rare or absent. In UEFI development this scenario matches the reality of most teams and unit testing as well as other testing techniques are often not used. To address this fault we propose UTTOS, a tool that parses EDKII build configuration files, mocks the UEFI-specific functions for C development and enables UEFI test suite code to run in the operating system. We show that UTTOS is able to run the test suit in the operating system and save development time.

Keywords: UEFI · Unit test · C · Code coverage · Embedded systems

1 Introduction

Testing has being often considered the crucial phase in the process of creating high quality software systems and most development frameworks include functionality to assist the use of some automatic testing strategy. In low level systems such as embedded systems, BIOS and device drivers development the situation is much less well established. Here, requirements are often fixed by protocol specification. This decreases complexity and most testing is basically checking for conformity to the protocols. On the other hand automated testing is not generally easy here since the code is targeted to run in devices different from the developer's workstation often through the use of cross compiling. Tools to assist the generation of automatic tests are rare in such platforms.

UEFI (Unified Extensible Firmware Interface) is the current specification for the interface between the platform firmware and the operating system, it is meant

F. Wotawa et al. (Eds.): ICTSS 2016, LNCS 9976, pp. 218–224, 2016.
DOI: 10.1007/978-3-319-47443-4_14

to eventually replace the BIOS (Basic Input Output System) and addresses many of it's limitations. First of all, the BIOS development is made in assembly language thus the complexity of keeping it modularized and maintainable was a big issue, BIOS runs on 16 bit mode so the amount of memory usable is very limited. In UEFI there is a call stack and most of the development is done in C, also the execution environment is 32bit or 64bit depending on the processor architecture. These features make it a much richer platform with many application possibilities, therefore going beyond the basic task of initializing the hardware and calling the OS loader [12,14].

UEFI also enables OEMs (Original Equipment Manufacturer) to bundle applications and drivers with the machine itself then providing a minimal OS-like environment. But with the greater flexibility provided by UEFI and the variety of software that is being written to that platform comes complexity, and then the need for robust development and quality assurance practices. The issue at hand is that the DXE (Driver Execution Environment) code is targeted to run in the pre-boot phases, so it is usually written and compiled in a workstation then run in the target machine in order to be tested and then brought back to the developers workstation to be validated and debugged.

It is possible to identify two alternatives to use automated test strategies for UEFI software: to develop or use testing tools that run in the DXE or develop tools to mimic the DXE by mocking its basic functions. We favour the second approach in order to run the code in to be tested in the OS context, thus overcoming the lack of test tools available by enabling DXE code to be run in the OS by using unit tests with the external dependencies, such as the functions in the boot services table, mocked.

Since the code is run in the OS, developers and testers have access to many other tools that exist to help them ensure code quality, such as code coverage and other tools that are not available in UEFI environment.

Contributions. In this paper we discuss a tool that enables unit tests for UEFI code to run as standard operating system programs and show a case study with a compliant driver to evaluate it's benefits. During the discussion of the driver implementation ideas about how to use the test suit to minimize the number of times a developer needs to rewrite the machine's flash memory and protocol conformity.

Section 2 describes the state-of-the-art techniques for unit testing of DXE code. It shows that the existing techniques are not extensible, and do not allow for further code metrics. In Sect. 3, the proposed tool and the way it works is explained. Section 4 describes the case study and presents results. Finally, Sect. 5 concludes this paper and proposes future work.

2 Related Work

There are several works advocating the use of unit testing in the development of software [3,7,8]. While in UEFI there are no specific unit test tools, there are validation tools for firmwares and drivers such as PI SCT [11] provided by the UEFI open source community, FWTS [1] and Chipsec [4]. All of these run functional tests.

The field that most closely resembles tasks accomplished in UEFI development is that of embedded systems development, mainly because of the use of C and the fact that many operating system services may not be available. In that context there are some tools that try to make test-driven development more pleasant, Unity [13] for example, provides several assertion macros to guide unit test development for C modules. CMock [13] parses the included header files and checks for function declarations to provide mocked or stub implementations of them, this allows one to remove the dependencies on the functionality of third party modules avoiding effects of their behaviour in the results of the tests. Ceedling [13] is a build management system that integrates those tools and allow custom configurations through its project descriptor file.

EDKII (EFI Development Kit) [10] module development uses the following structure: At first, a package is created. Inside the package are the platform descriptor file (.dsc), responsible for describing all the dependencies used by the modules in the platform. The declaration file (.dec) contains all the include paths used by the modules, as well as GUID declarations for protocol communication. Then, each module inside the package, be it an application or driver, has its descriptor file (.inf), which contains information about the module such as compiled source codes, libraries, dependencies and protocol GUIDs published or consumed, separated into sections, such as [Packages], [Guids] and [Protocols].

According to Saadat, H. [9], hardware diagnostics are not effective in detecting code errors, therefore, UEFI code must be tested from a software perspective. Also, it is mentioned that only one test tool may be insufficient to cover all the phases in UEFI, which is why a combination of tools must be used.

There are some groups interested in testing embedded systems code during development and UEFI code as well. However, the UEFI approach to unit testing still uses the UEFI environment, making the use of debbugers and other OS specific tools impossible. Our solution, called UTTOS, aims to execute the code in the OS, mocking UEFI specific dependencies, thus allowing unit tests to be run in the same development environment and enabling use of other tools, like code coverage tools.

3 The UTTOS Solution

It is possible to write unit tests that run in UEFI through the use of CuTestLib [5] or some other similar tool. However, the fact that the firmware run in a process-less, single threaded fashion poses a number of limitations. If there is a problem with a statement and the code breaks in a way that should generate a segmentation fault in an OS context, the firmware is likely to just freeze, leaving the test developer without any feedback or clue to what happened. This is an issue that may happen to even more sophisticated test approaches that involve transmission of test results over network or serial port.

Those limitations and the extra complexity needed to achieve simple tasks like writing test results to a screen or to a file in the firmware level makes good practices like test-driven development often impractical or even absent in many projects.

To avoid these problems, we propose an UEFI Test Tool for Operating Systems (UTTOS). UTTOS addresses this issue by enabling code that was written to the EDKII platform and meant to run in the DXE context to be compiled and executed as an ordinary executable in the operating system of the developer platform. The unit tests generated in this fashion are much more flexible as they enable the test developer to use the full stack of debugging tools that are available for C. If there is a segmentation fault, for example, the developer can make use of the core dump generated by the operating system and a debugger, like GDB, without any specialized hardware-aided solution.

UTTOS strategy for running EDKII in the operating system runtime consists of looking for dependencies declared in the descriptor of the module being compiled, that is our UEFI driver or application, and then generates stub versions of the EDKII specific functions. The developer is responsible for configuring the expected arguments and return values of these functions for each test suite. This effort is not repetitive since the most used functions can be reused.

The main tool used by UTTOS is Ceedling, that tracks dependencies and have a configuration file feature-rich enough to permit customization of the whole process. Ceedling itself makes use of a few other open source tools, namely: CMock for mocking and Unity for test creation. UTTOS acts in the beginning of the process parsing the EDKII build files and generating the Ceedling configuration accordingly. And as we are in an operating system environment we also include Gcov for coverage evaluation.

A more detailed view of the UTTOS workflow is described in Fig. 1. It first parses the module descriptor file and gathers information about the dependencies and the consumed resources. The resources in a UEFI firmware are identified by a Globally Unique Identifier (GUID). In the process of constructing the dependency tree, the descriptor files and declaration files (.dec extension) of other modules are read to gather all the required headers, include paths and GUIDs

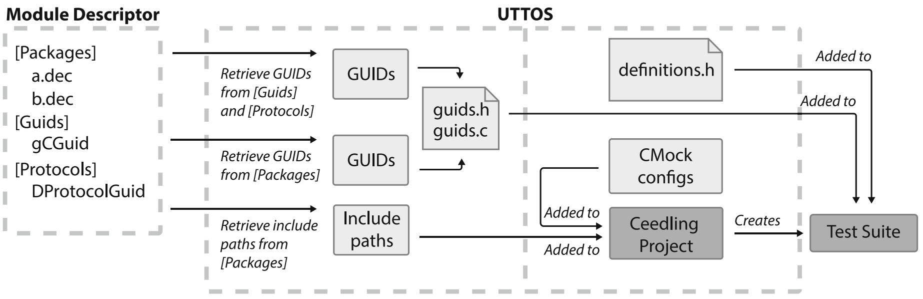


Fig. 1. Workflow of UTTOS, from module descriptor to test suites.

used by the current module. From this procedure, two files, guids.h and guids.c are output for use in the test suite compilation.

The next step is the generation of a Ceedling project where the module descriptor is located. The project is created using Ceedling's default configurations for a C project. After this, the Ceedling project descriptor is modified by UTTOS, that adds the include paths for the dependencies, the source files to be built, the suitable compiler options and the instructions to CMock on which functions have to be mocked.

The final step is the placing of the generated files inside the created test suite to allow correct code compilation. The test suite is generated with two stub functions for setup and clean up. At this stage, it is ready to use. Ceedling manages the test suite and its dependencies, assuring transparent use of the UEFI code. If further configuration is needed, the project is customizable after UTTOS executes.

With the suite ready to be executed, the user can add individual test cases with custom assertions and control flows. The user can also specify the expected parameter and return values for the mocks created by CMock to strengthen the tests coverage further. Ceedling also monitors suite execution and reports formatted results, allowing the developer to ensure quickly that the written code is working.

4 Case Study

As a case study, a simple UEFI DXE driver fully compliant with the UEFI Driver Model was developed. The driver installs a protocol and an associated GUID to the UEFI runtime that provides a service for UEFI applications to convert roman numbers in string format to a regular C integer type. Because one of the most interesting scenarios that requires testing in the context of driver development is the process of providing a new protocol, we decided to go with an algorithm that, although simple, provides a good number of test cases as there are a number of possibilities for invalid roman numbers.

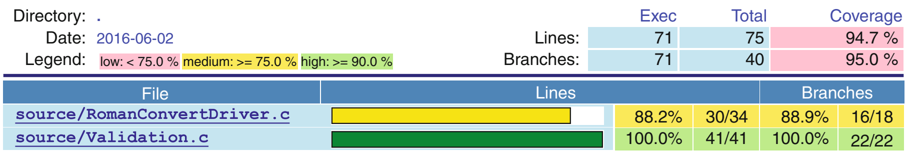
The driver was developed under the EDKII version 2.6 source tree in a Debian GNU/Linux 8 machine using the GCC compiler version 4.9.

This process starts by creating a set of functions called the Driver Binding Protocol [2]. These functions are responsible for querying all the nodes of the UEFI service database checking if any of these corresponds to a device that the driver can control. This query process is performed by the Supported function and it has to return a success status for a device that is suitable for the driver and an unsupported status otherwise. If such a device is found, the Start function is called to initialize the driver data structures and the device itself.

If for some reason the user wants to remove the driver from the environment, the driver binding protocol also provides the Stop function, that is responsible for releasing all the resources that were allocated by Start, and for the removal of any protocol installed by the driver and their respective GUIDs. Stop makes sure the services provided by the driver will not appear to be available to applications after it is unloaded.

In UEFI, all the available memory is shared as a single address space to all drivers and applications. Therefore, an error in any of these functions is likely to freeze the whole system. Consequently, all these steps must be previously validated before one takes the job of rewriting the firmware of the machine to check if the driver works as expected. Here is where the ability to run unit tests in the developer's workstation as a prior step shows it's value, since most of the trivial mistakes can be avoided before actually touching the target hardware.

The Gcov [6] utility was used to evaluate the code coverage of the tests performed. Gcov is another python tool that translates the symbols generated by GCC to human readable results allowing a nice html format to be output.



			Exec	Total	Coverage
Directory:	.				
Date:	2016-06-02	Lines:	71	75	94.7 %
Legend:	low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches:	71	40	95.0 %

File	Lines			Branches	
source/RomanConvertDriver.c		88.2%	30/34	88.9%	16/18
source/Validation.c		100.0%	41/41	100.0%	22/22

Fig. 2. Code coverage report of the implemented driver.

Figure 2 shows the test coverage results for our case study. It can be seen that most of the control flows were covered. The results of the individual test cases were all positive and they have successfully assured that the driver binding process would occur without errors when embedded in the machine's firmware. This is important in the sense that it allows the developer to be concerned with the driver's functionality correctness straight from the first firmware writing, and not with these initial service and protocol setup that often takes a considerable time in real world projects.

In comparison with the approach of testing the code in DXE environment directly, the first major benefit observed is that ones does not have to rewrite the firmware in the flash memory so many times to test code, a process that is time consuming and decreases flash memory life time. Additionally, code coverage measuring tools are one of many that are not available in the DXE phase so these results would not the easily achieved today.

5 Conclusion and Future Works

In this paper, we proposed UTTOS, a tool that makes use of a set of OS-based test tools to test UEFI code. It adapts the test framework to be able to run UEFI code as a regular operating system executable with that UEFI specific functions mocked. A case study was also conducted, with the implementation of an example driver in order to test the whole process of driver installation, production usage and removal to simulate as close as possible the reality of device driver development.

In the development time it was observed a decrease in the need for rewriting the machine's firmware and the ability to focus in functionality rather than in protocols conformity which can in practice be, using UTTOS, considered to be boilerplate code.

During the experiments, we were able to not only create unit tests, but to make use of code coverage, as we were running the code in an OS-based environment. This would not be possible by running the code only in UEFI DXE phase.

For future works, it is intended to integrate this tool with an IDE. Support for other code metrics, such as cyclomatic complexity and nesting depth can also be considered.

Acknowledgements. This publication is the result of a project promoted by the Brazilian Informatics Law (Law No. 8248 of 1991 and subsequent updates) and was developed under the Cooperation Agreement 11/2016 between Federal University of Ceara and HP Brazil Industry and Trade Electronic Equipment Ltda. This research was also partially supported by LSBD/UFC.

References

1. Canonical: Firmware Test Suite 16.01.00 (2016). https://wiki.ubuntu.com/FirmwareTestSuite. Accessed 23 May 2016
2. Corporation, I.: Uefi driver writer's guide (2013). https://github.com/tianocore/tianocore.github.io/wiki/UEFI-Driver-Writers-Guide. Accessed 01 June 2016
3. Greening, J.W.: Test Driven Development for Embedded C, 1st edn. Pragmatic Bookshelf, Raleigh (2011)
4. Intel Corporation: CHIPSEC: Platform Security Assessment Framework 1.2.2 (2015). http://www.intelsecurity.com/advanced-threat-researchchipsec.html. Accessed 23 May 2016
5. Jalis, A.: CuTest: C Unit Testing Framework 1.5 (2013). http://cutest.sourceforge.net. Accessed 23 May 2016
6. Ledru, S., Cai, K., Woydziak, L., Schumacher, N., Hart, W.: Gcovr (2014). http://gcovr.com. Accessed 23 May 2016
7. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, 3rd edn. Wiley, Hoboken (2011)
8. Pressman, R.S., Maxim, B.: Software Engineering: A Practitioner's Approach, 8th edn. McGraw-Hill Education, New York (2014)
9. Saadat, H.: Design and development of an automated regression test suite for UEFI (2014)
10. Tianocore: Edk2 code repository (2016). https://github.com/tianocore/edk2. Accessed 01 June 2016
11. UEFI Forum: Platform Initialization Self Certification Test 2.4B (2015). http://www.uefi.org/testtools. Accessed 23 May 2016
12. UEFI Forum: UEFI Specification 2.6 (2016). http://www.uefi.org/specifications. Accessed 23 May 2016
13. VanderVoord, M., Karlesky, M., Williams, G.: ThrowTheSwitch.org (2016). http://www.throwtheswitch.org/tools. Accessed 23 May 2016
14. Zimmer, V., Rothman, M., Marisetty, S.: Beyond BIOS: Developing with the Unified Extensible Firmware Interface, 2nd edn. Intel Press, Mountain View (2010)