Embedded Test-Driven Development Strategies

Piet Cordemans, Jeroen Boydens and Sille Van Landschoot

Abstract – Software quality plays an essential role in embedded systems. Test-Driven Development increases the quality of the source code, by implementing tests before the actual implementation is developed. However, the embedded environment, heavily depending on the hardware, poses some challenges to adopt TDD. In this paper, three migration strategies are defined, which gradually move the development cycle from target-based to host-based. These strategies are illustrated in a step-wise workshop, which indicates several advantages and disadvantages of each technique.

Keywords – Embedded software, Test-Driven Development

I. INTRODUCTION

Software quality is an important issue nowadays and this is definitely the case when embedded software is considered, as embedded software bugs can result in damage to machinery or even physical injury of human beings. However, testing embedded software is generally limited to ad-hoc debugging or final testing. As testing software is commonly put off until the end of the project, the quality assurance phase might be shortened when the deadline is near.

Earlier integration of testing in the development cycle ensures that the quality of embedded software is guaranteed. Test-Driven Development (TDD) is a software development process that promotes writing a test before the implementation itself. Following this paradigm ensures that the code is always covered by automated tests.

Though, characteristics inherent to embedded systems present some obstacles for TDD. Typically embedded software is strongly related to the underlying hardware. Co-design of hard- and software is therefore essential for embedded systems. However, embedded hardware is not always available during software development, as a prototype might not be complete or does not even exist.

In section II, the TDD methodology is discussed, as well as the implications posed by the embedded system. Next, section III illustrates three complementary strategies to migrate the TDD principles to embedded software. In section IV, these principles are demonstrated in a step-wise workshop. Finally, section V indicates the topics of interest, which were not presented in the workshop.

II. PROBLEM STATEMENT

First, a general overview of Test-Driven Development and embedded software is given. Next, the main challenges to migrate the TDD principles to embedded software are stated.

A. Test-Driven Development

The software development methodology of Test-Driven Development (TDD) is a main pillar in several incremental processes, for instance in eXtreme Programming and Scrum. Over the past decade TDD has risen to prominence in the business software application development world. A TDD cycle starts with the implementation of a desired functionality in an executable test. This test is added to a test suite, which is run frequently during the development cycle. Next, the test needs to compile without any errors, hence all references need to be resolved. Since only skeleton code for the unit under test is available at this point, this will ultimately lead to a failing test. The failing test indicates that, as expected, new behavior is added to the program. Ideally, in TDD no functional logic can be implemented without a failing test. Now, the necessary code of the unit under test can be written, focusing only on the current issue. The implementation is ready when the test is passing. Finally, the program is refactored, that is increasing the quality of the code without changing or extending its behavior. As an automated test suite is frequently run in this process, any unintended alterations in behavior are quickly detected. To relieve the burden of the programmer, a unit testing framework is an essential tool to implement, organize and execute unit tests[8].

The benefits to TDD are numerous. First, bugs are detected by tests early on in the development phase. As Boehm[2] stated already in the eighties: the earlier bugs are found, the cheaper it is to solve them. Next, TDD emphasizes on refactoring, so the program stays readable, maintainable[6] and all duplication[1] is removed. Finally, international research[9][7], as illustrated in figure 1, states that even though TDD increases development time, it should reduce the quality assurance phase significantly. This reduction should compensate for the increase of development time or even lead to a net gain.

B. Embedded software obstacles

Embedded system design is typically a co-design of hardware and software[10]. This implies that the target hardware might not be available when software development is started. Other obstructions to apply TDD
on embedded software development are resources, for instance memory footprint or processing power, which are typically limited on an embedded system. Finally, the build process of embedded software is extended with a deployment phase to the target, which adds additional delay to the process.

The adaption of the TDD techniques for embedded software however, is not as smooth as it has been for business application software development. This can mainly be credited to the typical characteristics of embedded systems. The main impediment is the difficulty to automate tests in an embedded environment. As manual tests are error-prone, labor intensive and boring, they are not suited in a fast-paced process like TDD. It is vital that the developer can execute the tests with minimal effort and receives feedback in a reasonable timespan.

III. MIGRATION STRATEGIES

In order to deal with the impediments raised by embedded systems, three strategies have been defined: (1) TDD on target, (2) TDD with a testwrapper and (3) TDD on host. These strategies try to balance between limitations posed by the embedded target hardware and making extensive use of host-based prototyping\[3\].

A. TDD on target

Figure 2 illustrates the first strategy where tests are placed alongside the program on the target hardware. Typically in TDD the memory footprint of the tests is approximately of a similar size as the footprint of the program. To limit the burden of the tests, a minimalistic testing framework can be used, such as MinUnit[12] or CppUnitLite[11]. However, this strategy does not solve any of the inhibitions posed by the embedded system. Nevertheless, testing on target still has its merits. First, it is ideally suited for final testing, when the whole system is considered. Next, only a selection of tests, which cover specific embedded capabilities, should be considered. Finally, testing on target verifies the assumptions made when prototyping on the host system.

B. TDD with testwrapper

The second strategy differs from the first as the tests are migrated from target to host. A testwrapper on the target system is needed to make the program testable, as shown in figure 3. Furthermore, a communication channel between the testwrapper and the tests on host should be provided. This strategy has some resemblance with In-Circuit Emulation(ICE) debugging. Its main benefit, as opposed to ICE, is the automatic execution of a number of tests.

C. TDD on host

This technique is based on host-based prototyping, which removes the need for effective hardware. Instead of dealing with real drivers, virtual drivers are implemented, as shown in figure 4.

In business application software a virtual component is called a mock. However mocking the target’s drivers contains a risk since wrong assumptions can be made about their functionality. A verification of the correct implementation of the mocks is necessary. The other strategies, such as TDD on target or TDD with a
testwrapper, can be used in that case. Since no deployment of the program on the embedded system is involved, the strategy is fast, comparable to developing a business application in TDD. Furthermore, it also deals with the major delay in embedded development of unavailable hardware at the beginning of the development phase. The final advantage is that the strategy forces to make an abstraction of the drivers on a higher level, as real and mocked drivers should be easily switched. This switching is realized by programming to an interface[5], which is one of the key design principles for OO. Switching easily of implementation underneath the interface is supported by polymorphism.

However, the dominant language for embedded systems, C, does not support polymorphism. Several techniques exist to deal with this inadequacy, such as link-time polymorphism, preprocessor linking and the use of function pointers to include either the real or the mocked driver[4].

IV. ILLUSTRATING THE STRATEGIES IN A STEPWISE WORKSHOP

The three strategies, as defined in section III, are clarified in a workshop for embedded software developers with no TDD experience. The workshop consists of seven steps to gradually introduce the TDD principles for embedded software.

1) The workshop starts with the implementation of a C++ library with the general TDD strategy. The library provides functionality to store and retrieve temperature values in Kelvin, Celsius and Fahrenheit. Development in this phase is done on host, without implementing any hardware drivers, either mocked or real. This step serves to demystify the process of TDD, without any limitations posed by the hardware. The testing framework of choice is UnitTest++[13], a xUnit[17] testing framework that also can be ported to an embedded environment. UnitTest++ provides the most important features of a testing framework. These features are: assertion functionality, test organization and support for setup and teardown to isolate each test.

2) Next, the temperature library is migrated to the target. In this case, the target is an ARM development board with an ARM7TDMI processor, a temperature sensor, 2x16 LCD display and an UART providing RS232 communication with the host PC. Although UnitTest++ can be ported to an embedded system, a more minimalistic framework was chosen to implement the test on target strategy. MinUnit is a minimalistic unit testing framework, which consists of only two macros written in C. The first macro provides a simple assertion mechanism and the other one runs the tests. In order to setup tests and display test results, additional functions are provided. The choice of displaying the test results is dependent on the available options of the embedded system. Usually embedded systems are provided with an interface for debugging, for instance USB, JTAG or UART, or has an interface to communicate with the user, such as a LCD display, where the results can be directed. Some of the unit tests can be ported to the target as well. Considering the simplicity of the temperature library and its hardware independence, no tests are expected to report a failure.

3) The third step introduces TDD for embedded software, making use of the strategy of TDD on target. In this step, raw data is converted from the temperature sensor and is displayed in degrees Celsius on the LCD. This conversion is hardware-specific, so a consultation of the datasheet reveals the specifications of the temperature sensor. This information makes it possible to write the test validating the program’s behavior before implementing the code itself.

The strength of TDD for embedded development becomes apparent as the conversion is tested without the need to output the result and manually validate it on the LCD. Thus dependencies on hardware and drivers to test functionality, are significantly reduced.

4) Until now, every test ran on target in the adapted MinUnit framework. As testcode equally grows with production code, additional program memory is occupied with each new test. As illustrated in the second strategy, a testwrapper is introduced to solve the limited memory footprint problem, which is characteristic to embedded systems. With a testwrapper, the tests are migrated to the host, but the production code stays on the target. To communicate between the code on target and the tests on host, the system’s UART is used. The UART has a couple of advantages as it is already available on the hardware and is relatively easy to use. However the implementation of this communication channel is entirely dependent on the available hardware of the embedded system.

5) The final strategy of TDD for embedded development is migrating the development entirely to the host system. As the implementation on host has no reference hardware, extensive use of hardware mocks is necessary. To illustrate the development of a mock driver, the temperature sensor driver is mocked. To ensure conformity with the real driver, the same interface is used. However the implementation will differ, as the mock only needs to respond as expected. Typically, the mock object is initialized with some static values and offers the necessary mutators to update the dynamic variables. Before the actual test is executed the testcase must setup the mock object in the correct state. Additional functionality can be integrated into the mock object. On the one hand it could only consist of stubs and drivers, treating the unit under test as a black box. On the other hand the mock object could verify whether the production code has called a function or not, which is a form of white-box testing. The general advice is to limit these assertions to one per mock and to limit the mock objects to one per unit test.

6) When writing mock drivers, writing the full implementation of the hardware is unnecessary or even redundant. When providing only the necessary interfacing in the mock driver, errors or imperfections could be introduced. For instance, the mock driver might differ from the actual driver, as wrong assumptions are made or if the functionality does not exist in the mock. So there are no guarantees about the quality of the mock, as long as the mock is not verified with the real driver. When the hardware becomes available in a later stage of the project, the mock driver can be replaced with the real driver on target. By running the tests and by checking if they behave...
the same way as with the mock drivers, it can be verified that the mock drivers are a good representation of the real drivers. When introducing an interface, switching between a mock object and the real driver object should easily be done with the help of polymorphism. Both real driver and mock should implement the same interface so that production code is ignorant of the actual underlying driver that is being used.

(7) In many cases, C is used for embedded software development instead of C++. The C language does not support polymorphism, so when developing in C other techniques need to be applied for easy switching between a mock and the real driver. There are three possible solutions to deal with this problem. A first solution is replacing functions by function pointers, which can be used for calling the desired functions of the real driver or the mock driver. This option actually requires some manual labor to switch between mocks and real drivers and is therefore not desirable. A second option is to use preprocessor macros, which can respectively call a mock function or the real function by changing a user-specified preprocessor directive. However, as preprocessor macros are purely text replacements they do not provide any compiler guarantees. Hence, there is some resistance to using this solution. The final solution is called link-time polymorphism. This technique changes the pointers to the real or mock objectfile at link time. Link-time polymorphism can easily be automated by a build tool, by specifying different build commands, which link in the respective object files.

V. FUTURE WORK

Relieving the burden to manage mocked and real drivers is currently not discussed. In analogy to developing tests that are managed by a testing framework, mocks can be managed by a mocking framework. A mocking framework usually complements the testing framework, as both frameworks are tightly coupled. The options for C++ mocking frameworks are currently limited. However, in 2008 GoogleTest[14] has been released, which can be extended with GoogleMock[15], its complementary mocking framework. This could be a valuable addition to the test on host strategy.

Apart from unit tests, acceptance tests can be defined, which aid the user of the system to define executable specifications. An acceptance testing framework, for instance FitNesse[16], can be used to define, execute and report customer defined acceptance tests. Integration of FitNesse with the testing on host strategy, will enable the customer to define and follow-up software acceptance tests even before embedded hardware is available.

VI. CONCLUSION

This paper indicates that the migration of TDD principles to embedded software is not evident but possible, regardless of the impediments posed by the embedded system. The three strategies are mainly concerned to automate the software development and test cycle to ensure that TDD does not slow down the process. However, it is also shown that no single migration strategy is the ideal solution. Each has its specific use and it is up to the developer to adapt his strategy to the specific needs posed by the application. Nevertheless, regardless of the strategy, testing the embedded software will be done more frequently and will cover a broader range of testing scenarios. Finally it is illustrated that TDD complements co-design of hardware and software, which is an essential process in the development of embedded systems.

REFERENCES

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.