Test-Driven Development Strategies applied to Embedded Software

Cordemans, Piet\textsuperscript{a1}, Boydens, Jeroen\textsuperscript{b1,2}, Van Landschoot, Sille\textsuperscript{a1}, Steegmans, Eric\textsuperscript{b2}

January 30, 2012

\textsuperscript{a1}KHBO” Dept of Industrial Engineering Science & Technology
Zeedijk 101, B-8400 Oostende, Belgium
Piet.Cordemans@khbo.be

\textsuperscript{b2}KU Leuven” Dept of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract

Test-Driven Development (TDD) can contribute to the overall quality of embedded software. However, applying TDD in an embedded environment is not obvious. Hardware dependencies, a slow upload process and limited resources are major constraints developing in a TDD fashion. To deal with these limitations four strategies were defined. According to Test on host, hardware dependencies are replaced by mocks, which enables developing tests and embedded software on the host system. Consequently, verifying the expected behavior of mocks is necessary. Therefore Test on target migrates tests, a testing framework and software to the target embedded system. Another approach called Remote testing employs Inter-Process Communication (IPC). This enables tests running on host, while the code under test executes on target. Finally, both tests and a part of the embedded software can run on host, delegating hardware-related calls to the target with IPC. This strategy is called Remote prototyping. A mathematical model was developed to evaluate these strategies using a single relevant metric, namely the number of required code uploads.

Keywords: embedded software, test-driven development.

\textsuperscript{a}P. Cordemans and S. Van Landschoot are scientific staff members at KHBO funded by IWT-090191.
\textsuperscript{b}J. Boydens is a professor at KHBO and an affiliated researcher at KU Leuven, dept. CS, research group DISTRINET.
\textsuperscript{c}E. Steegmans is a professor and a faculty member at KU Leuven, dept. CS, research group DISTRINET.
1 Problem statement

In the Test-Driven Development (TDD) cycle, a test is written before code covered by that test is implemented [4]. TDD increases code quality by shifting focus from implementation details to the public interface of software components. Moreover, the code base is continuously tested, while the test suite and code base is incrementally developed. This facilitates code maintenance and reduces the number of bugs found in later phases of the software development cycle [7].

However, applying TDD to embedded software is perceived to be impractical, because of a number of embedded software properties. Most importantly, TDD requires an automated test suite, which is executed during each step in the TDD cycle. Yet, embedded environments are not suited for such an automated test suite. First, frequent code uploads to the embedded target are impractical, since they require considerably more time than executing tests on the development system. Also, the embedded target resources might be too limited to run the total of a testing framework, tests and the code under test. Furthermore the embedded target might not be available, as hardware is co-developed with software. Finally, hardware dependencies introduce non-deterministic behavior, unacceptable slow tests or require complex test setups.

Nevertheless, embedded software can be developed according to the TDD principles. Though strategies to cope with the limitations posed by the embedded environment are in order.

These strategies are described in section 2. Next, section 3 describes the case studies. In section 4 we evaluate the strategies. Finally, section 5 and section 6 show related and future work respectively.

2 Embedded TDD strategies

In order to deal with embedded constraints, four strategies were defined: Test on host, Test on target, Remote testing, and Remote prototyping. Each of these strategies deals with a specific problem and in certain situations they are complementary to each other.

2.1 Test on host

In an embedded environment, frequent code uploads slow down the development cycle, because of delay introduced by programming the target memory. Repeatedly uploading software to target should be shunned, thus Test on
HOST avoids this altogether. Tests run in the host environment as long as hardware-specific code is not called. Therefore calls to driver code are delegated to fake components instead. These fake components are commonly referred to as mocks, as they mock the expected driver behavior. Applying TEST ON HOST requires two build configurations, namely one build for host and one for target, as shown in figure 1. Mocks allow running tests on host, yet cross-platform issues are not detected this way. These issues are related to (1) the cross-compilation process, (2) data representation on target or (3) behavior in mocks, which do not resemble the real driver. In order to identify these problems, the second build can be deployed to target. However verification on target is not required in every step of the TDD cycle, which still allows to save time.

![Diagram](image)

**Figure 1:** Test on host strategy.

Developing on host has an additional benefit, as code must be developed to run on several target platforms. Platform independent software allows reusing code and implies to organize code in appropriate layers of abstraction. Isolation from hardware-related behavior is critical with the purpose of dynamically delegating calls to the real or mock implementation. Moreover interfaces hide the internal implementation of components, which allows to switch easily between mock and real code. Two methods are preferred to switch between host and target configurations. On the one hand is dependency injection, using run-time composition to inject either mock or real driver in the test. On the other hand is link-time polymorphism, which uses the build system to include the desired component.
2.2 Test on target

Ignoring the long upload times of software to the embedded target, Test on target requires tests, testing framework and code under development placed in target memory, as shown in figure 2. This approach has two distinct advantages. On the one hand, some hardware-specific features, for instance real-time behavior, can only be tested on target. On the other hand, if the application makes use of a substantial legacy code base on target, extending legacy software might break the system. Test on target can validate existing behavior, when a host build configuration does not exist yet.

![Diagram](image)

**Figure 2**: Test on target strategy.

Fundamental to this strategy is a portable testing framework, which introduces only a minimal overhead. Moreover timed asserts or memory leak detection are interesting features, which might be provided by the testing framework.

2.3 Remote testing

Tests and software under test do not have to reside in the same physical memory. Inter-Process Communication (IPC) allows running tests on host, while the software under test executes on target. This strategy is called Remote testing. However most IPC implementations are too broad and introduce too much overhead, which would unnecessarily burden the embedded system. Yet considering the typical heterogeneous nature and the role of target and host systems, only a subset of IPC is effectively needed. Namely, two fundamental properties allow to simplify the system. First, objects of interest will only exist on target. Furthermore all communication will be initiated by the host system. Basically as shown in figure 3, a broker is needed on host and target side, allowing communication between both. However,
Unlike typical IPC systems, the broker on host and target differ. On the one hand the broker on target will contain a list of remotely addressable classes and a list of skeletons, which are remotely instantiated objects. On the other hand the broker on host will maintain the stubs, which are interfaces of the objects residing in target memory.

**Figure 3:** Remote testing strategy.

Remote testing starts with initializing both brokers and a test runner on host. Consequently a test is written on host, following the TDD cycle. In this test, one or more calls are made to functions on target. Either these functions are already remotely defined or not. In the first case developing tests and embedded software remain unchanged. In the latter case boilerplate code is needed to set up communication, namely stubs and skeletons must be generated. Stubs and skeletons provide a system, which allows to address local or remote functions alike. Namely, stubs provide an interface identical to the one on target, as if the effective function would be implemented on host. Rather than executing the called function's implementation, a stub merely redirects the call to target and delivers a return value as should the function have been called locally.

Most IPC implementations deal with two fundamental problems, specifically consistency and object serialization. In the Remote testing context these problems can be simplified. On the one hand objects can be dynamically created and destroyed, as the broker on target will maintain references on target. These references are branded with an identification tag, which binds a stub with a corresponding skeleton. This allows a simple consistency scheme between host and target environment. On the other hand object serialization is avoided altogether for three reasons. First, the heterogeneous setup of host and target does not uniformly define its types, which would entail for
object serialization to include an additional translation step. Next, it would require additional resources from the embedded system. Finally, the simple consistency scheme only allows remote addressable objects on target. The lack of object serialization limits testing capabilities, yet these limitations can be circumvented. Rather than returning objects, these objects can be asserted on a field-per-field basis through IPC.

2.4 Remote prototyping

Whereas Remote testing reduces the number of uploads to target by incorporating IPC, Remote prototyping builds further upon this idea. It is based on the assumption that code under development is subject to change, while a finished component will only rarely alter. Principally, Remote prototyping consists of two phases. In effect the first phase is similar to Test on host, namely it aims to develop as much code as possible on host. Yet, instead of replacing drivers with mocks, it delegates hardware-specific calls on host to target. As illustrated in figure 4, on the embedded target a minimal layer of remote addressable code is needed, which consists of the hardware dependent function.

![Diagram showing Remote prototyping concept](image)

**Figure 4:** Remote prototyping

The second phase of Remote prototyping begins once the software component under development has reached a stable state. Then it can be incorporated in the stable code base on target. A thorough testing phase is in order to detect any cross-platform issues. Since the IPC infrastructure is already used, Remote testing is an obvious choice. Thus stubs and skeletons need to be generated for the added software component. Also, once these issues have been solved, the remote addressable functions will be
available for continuing development on the host system.

Considering typical embedded software implementations, dynamic allocation of remote objects is not always required nor preferred. For instance hardware drivers typically map to a physical component. Rather than dynamically configuring the entire system in test setup, a selection of these components can be statically configured in the stable code base. This saves test execution time at the cost of test environment control and memory footprint.

3 Case studies

Three case studies have been built as a proof of concept to evaluate the four strategies. These case studies cover a broad range of embedded systems as they differ in embedded platform, code size and the inclusion of legacy code on target. Table 1 gives an overview of size (LOC) and McCabe VG complexity (MVG) as measured by LocMetrics [2].

<table>
<thead>
<tr>
<th></th>
<th>(1) Case 1</th>
<th>(2) Case 2</th>
<th>(3) Case 2</th>
<th>(4) Case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>1535</td>
<td>251</td>
<td>531</td>
<td>621</td>
</tr>
<tr>
<td>MVG</td>
<td>99</td>
<td>43</td>
<td>48</td>
<td>140</td>
</tr>
<tr>
<td>Test on Target</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test on Host</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Testing</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Remote Prototyping</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A first case study (1) is a proof of concept for each of the strategies. Each part of its code was developed according to a strategy chosen in an ad hoc manner. Next, a second case study consists of multiple embedded platforms, whereas each focused on a different strategy, namely Remote Testing (2) and Test on Host (3). As a proof of concept the final case study was conducted on the extension of an industrial embedded system. Starting from a legacy code base (155 KLOC), an application (4) was developed with the Test on Target strategy. However not all the strategies have been applied in each of the case studies. Therefore Test on Target is used as a
baseline for comparison between the case studies and the other development strategies, as indicated in section 4.

Test-related infrastructure introduces overhead on target, which differs for Test on Target and Host on the one hand and Remote Testing and Prototyping on the other hand. EmbUnit [1] was the testing framework of choice for Test on Target, because of its small size (153 LOC). To support the Remote Testing and Remote Prototyping strategies an IPC broker was developed for the first case and compared with an existing embedded RPC library[3] for the second case. The size of the in-house developed broker is 192 LOC, while the infrastructure requires 20 LOC per remote class and 2 LOC per remote method. Similarly, the size of the RPC library is 126 LOC, requiring 5 LOC for each remote function.

4 Evaluation

Four strategies to apply TDD to embedded software have been defined. Furthermore an implementation according to these strategies has been made in three case studies of comparable size, yet differing in complexity. Other factors might influence the development process as well. For instance experience with the techniques and strategies, familiarity with the embedded environment and the abstraction level of embedded software. This list of factors is not complete, but indicates that measuring the effect of a development strategy is often subject to factors, which are not quantifiable. Nevertheless, an attempt is made to provide a quantifiable evaluation strategy for a specific metric.

Principally in TDD, tests run very often. Even in a cycle without unexpected faults, each step requires a test run. If it takes long to run the tests, the TDD cycle is unintentionally slowed down. This is a major issue for embedded software development, since uploading software to target is a time-intensive activity. Thus, reducing the number of required uploads is a prime consideration for each embedded software development strategy.

Regarding the number of code uploads, Test on Target requires the most, as the tests must run on target every step of the TDD cycle. At minimum, each test leads to two uploads, namely one to prove the test is effectively failing and a second one, containing the implementation to make the test pass. Note that this does not consider any additional uploads, for instance during refactoring or when the tests or implementation change.

Considering the required code uploads as the sole metric, the other strategies can be evaluated with Test on Target as the baseline.

First, Remote Testing decreases the number of uploads required by tests.
However, it does not eliminate all uploads associated with tests. Basically, if a test calls a new remote function, an empty skeleton implementation needs to be uploaded nonetheless. This will limit the benefit of Remote testing when compared to Test on Target. More precise, the number of uploads saved by Remote testing can be expressed as:

$$\frac{T - R}{T + C} \times 100\%$$

(1)

Where T is the number of tests, C is the number of implementation uploads to target and R is the number of remote addressable functions. As demonstrated in Table 2, this leads to a benefit between 0% and almost 50%.

<table>
<thead>
<tr>
<th>Table 2: Remote testing benefit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T : C : R</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Worst case</td>
</tr>
<tr>
<td>2 TDD cycles</td>
</tr>
<tr>
<td>N TDD cycles</td>
</tr>
</tbody>
</table>

The worst case for Remote testing implies that every test calls at least one new function on target. This is the case where T equals R. In that case every test will require a code upload nevertheless and no gain is obtained of the strategy. On the other hand, a test might immediately lead to a correct implementation and no refactoring is needed. In this case an ideal TDD cycle is achieved, indicating a benefit of almost 50%.

However it is observed that a test is less likely to change than its code under test, reducing the benefits of Remote testing. The possibility of changing code to reach a passing test, either during implementation or refactoring, is higher than the (re)definition of tests. This is partially due to the higher cyclomatic complexity of production code.

Where Remote prototyping is concerned, it is possible to set up a situation which is in complete accordance to Remote testing. For instance when a test is added and the code implementation is developed on target ($C_T$). However code developed on host instead, will reduce the number of uploads ($C_H$). Namely each code change which would otherwise provoke an additional upload will add to the benefit of Remote prototyping. A simple model to express the relation between Remote testing ($\alpha$) and the added benefit of Remote prototyping ($\beta$) is the relative size of their respective source code. On the one hand the number lines of code related to
code and tests developed on host \( (\text{LOC}_H) \). On the other hand, the lines of code belonging to the IPC infrastructure and minimal layer of driver code on target \( (\text{LOC}_T) \). The general benefit of Remote Prototyping over Test on Target can be expressed as

\[
\alpha \left( \frac{T - R}{T + C_T} + \beta \frac{C_H}{T + C_H} \right) \times 100\% \tag{2}
\]

where:

\[
\alpha = \frac{\text{LOC}_T}{\text{LOC}_{\text{TOTAL}}} \quad \beta = \frac{\text{LOC}_H}{\text{LOC}_{\text{TOTAL}}} \tag{3}
\]

A selected number of samples is given in Table 3.

<table>
<thead>
<tr>
<th>T</th>
<th>Target ( (\alpha : C_T : R) )</th>
<th>Host ( (\beta : C_H) )</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 : 0 : 0</td>
<td>1 : 1</td>
<td>50%</td>
</tr>
<tr>
<td>1</td>
<td>0 : 0 : 0</td>
<td>1 : 2</td>
<td>66%</td>
</tr>
<tr>
<td>1</td>
<td>0 : 0 : 0</td>
<td>1 : N</td>
<td>Max = 99.99%</td>
</tr>
<tr>
<td>1</td>
<td>0.75 : 1 : 1</td>
<td>0.25 : 1</td>
<td>12.5%</td>
</tr>
<tr>
<td>1</td>
<td>1-\beta : 1 : 1</td>
<td>\beta : 1</td>
<td>Min = \frac{1}{2} \beta * 100%</td>
</tr>
</tbody>
</table>

Remote Prototyping has a potential improvement of almost 100%, as in theory a single code upload would be sufficient. On the other hand, its minimum improvement indicates that is strictly superior to Remote Testing and Test on Target.

Finally, the Test on Host strategy can be expressed in a similar fashion. This strategy’s benefit corresponds to the Remote Prototyping profit, as in

\[
\left( 1 - \frac{U}{T + C} \right) \times 100\% \tag{4}
\]

, where the verification upload to target \( (U) \) is a recurrent irregular task, executed at the discretion of the programmer.

In the evaluation, only the number of uploads was considered as a metric for the embedded TDD strategies. However, especially in the Test on Host strategy, this is not the only important metric. The effort to mock hardware on host or to evade non-deterministic hardware properties, should also be taken into account. Moreover, for the strategies depending on IPC
infrastructure, the overhead of developing this infrastructure is not negligible. However future work might enable to further refine this evaluation model.

5 Related work

Both Test on Host and Test on Target strategies have been described in literature. Smith et al. presented E-TDD [9], an effort to port several unit testing frameworks to DSP platforms. This corresponds to the Test on Target strategy.

Alternatively Granning proposes a technique called dual targeting [5], which corresponds to Test on Host. In his work, he indicated several techniques to apply TDD without the use of Object Oriented principles. Also, Karlesky et al. developed the Model Conductor Hardware (MCH) pattern [6]. This pattern describes a code organization technique to separate hardware state from the effective implementation. To our knowledge, the Remote Testing and Remote Prototyping strategies are novel concepts.

The evaluation of TDD in general has already been extensively examined by various studies [7, 10] and specifically for embedded software by Van Schooenderwoert [8]. Yet these studies are inconclusive and sometimes contradictory to one another. They also rely on an ad hoc comparison method, which makes it difficult to validate the results. In this respect our research proposes a new evaluation model, which offers a unified comparison strategy.

6 Future work

In an effort to compare the embedded TDD strategies, a qualitative evaluation model has been developed. However this model must still be validated on an industrial level. For instance a number of quantitative case studies could indicate the model’s correctness. Furthermore these would allow refining the model with the incorporation of additional parameters, such as boilerplate code overhead for mocks or IPC infrastructure.

Another item of future work concerns the overhead introduced by the IPC infrastructure. Yet, all information to automatically generate the necessary boilerplate code is already available in the test. Namely, the hardware-dependent call contains all information about parameters, function name and return type. Development according to Remote Testing or Remote Prototyping could be largely supported by a tool to generate the relevant stubs and skeletons. This tool would be similar to the use of a mocking framework tool in a Test on Host or similar strategy.
7 Conclusion

In order to successfully apply TDD to embedded software, several obstacles must be solved. Mainly, the number of required code uploads must be limited and hardware dependencies must be dealt with.

This paper presents four strategies dealing with the impediments to apply TDD to embedded software. As indicated in the case studies, each of these strategies has its particular use. Thus the evaluation of which strategy is preferred in a specific situation depends on various metrics. In our evaluation we considered the number of necessary code uploads as the most important metric. This way it was possible to present a qualitative model to indicate a preferred development strategy.

References