A More Agile Approach to Embedded System Development

Michael Smith, University of Calgary
James Miller, University of Alberta
Lily Huang, NovAtel
Albert Tran, DirectVoxx

Since 2004, we’ve successfully used a prototype test-driven development (TDD) embedded system test framework, called Embedded-Unit, in our undergraduate classes and research work. During the original course offering, we gave students a brief introduction to TDD concepts for embedded development. We then asked them to write tests that expressed the design ideas surrounding the initialization of the Analog Devices ADSP-BF5XX (Blackfin) processor’s core timer. When developing the code to satisfy the tests, the students split into two groups:

- those who followed agile’s standard “KIS—Keep it Simple” approach and initialized the timer’s hardware registers in the order the parameters were passed into the subroutine (TCOUNT, then TPERIOD; see Figure 1, lines 11, 12); and
- those who set the registers in a contrary order (TPERIOD, then TCOUNT; lines 16, 17).

The first group’s tests failed (lines 29, 30); the second group’s tests passed (lines 23, 24). Before the students informed us of this incongruence, they discussed the issue among themselves and identified a previously unrecognized silicon anomaly: the TPERIOD register was overwriting the TCOUNT register.

Why did inexperienced students successfully identify an undocumented feature that industrial firms, with product lines involving many tens of thousands of units of this processor, did not? We believe that the Embedded-Unit TDD environment enabled the students to easily apply basic agile concepts. In particular, the framework provided a stable, easy-to-learn environment through which the students could solve problems generated at both the hardware and software levels. Also, TDD reminds many students of other aspects of their training: strong parallels exist between the scientific method and TDD, with the establishment of a hypothesis (test) followed by its experimental realization (solution code).

Given this initial success, we asked ourselves a question related to our own research: Would generalizing other agile concepts into a full embedded system approach assist during medical embedded instrument development? Christian Denger and his colleagues indicate that people in this area often have no formal testing training, leading to many health and safety concerns. Bill Greene has suggested that agile’s barely sufficient processes should be acceptable to embedded system engineers who recognize the need for reliable products but aren’t...
eager to implement process improvement edicts from above.3 Indeed, with many embedded products having a near-zero tolerance for defects, a highly test-oriented process would seem to provide an appropriate design approach. However, Greene and those following in his footsteps point out the same basic problems in adopting many agile concepts: lack of tools to support specific embedded requirements, memory limitations when using existing tools translated into this new environment, stricter timing requirements, and issues specific to the particular processor used in the product’s design.

Recently we’ve identified how to use new debug hardware in modern embedded processors to solve many of the problems detailed by Greene.4,5 Therefore, the time was appropriate to explain why we felt that one simple embedded teaching success should be expanded into a full-blown Extreme Programming-inspired (XPI) embedded life cycle.

**XPI Embedded System Life Cycle**

Biomedical-instrument development typically starts with an initial scientific investigation to examine new hypotheses for medical diagnosis and analysis. This is followed by the proposed algorithm moving through several life-cycle phases to the finished medical embedded product. We believe that the lack of consideration of agile approaches for development within biomedical and other embedded environments is caused by two factors:

- **Lack of full life-cycle support.** Embedded applications have a life cycle different from that of desktop applications. So, agile production processes require significant alteration to meet this new domain’s demands.
- **Lack of tool support.** Without full life-cycle tool support, no change to production processes can succeed. Again, desktop tools must be adapted to accommodate this new environment.

While XP is normally considered a “general approach,” we’ve adapted an XP subset to define the core of a full life-cycle fault prevention approach to producing embedded software. Because industrial-based empirical studies have demonstrated that TDD is an effective defect reduction strategy in nonembedded situations,6 we anticipated that these advantages would transfer into the embedded domain. However, significant adaptation is required, so XP’s creators might well not view the adapted approach as being faithful to their original concept. Our XPI process seeks to transform their preexisting processes into a
structure that recognizes this new domain’s production objectives.

Stage 1: Envisioning XPI Embedded Products
In effect, this stage produces requirements. The customers or end users (domain-knowledgeable personnel) should be able to specify their ideas as a set of acceptance tests (in conjunction with standard user stories) in a straightforward manner. However, our approach to implementing Stage 1 deviates from the traditional viewpoint in four respects:

- **Coverage.** Current acceptance-test-oriented approaches provide limited information regarding test case volume and diversity. However, guidance toward the production of a comprehensive set of tests from the customer, in conjunction with the domain technical specialists, is paramount to successful production.
- **Nonfunctional tests.** Kent Beck describes XP nonfunctional testing as a process to be undertaken after system production.7 However, we believe that nonfunctional acceptance tests must be specified as early as possible, so we’re redeveloping acceptance-testing frameworks to support these essential components.
- **Domain-specific support.** The current test vectors supported by acceptance-testing frameworks are straightforward, and assume that you can encode knowledge using simple data types (such as strings and numbers), an approach appropriate for the business community. However, the embedded community has its own fundamental data types—for example, signals and images—which currently aren’t well supported.
- **High-level prototyping.** Unlike a standard XP process, our acceptance tests now drive a high-level prototyping stage (Stage 2) rather than the final production stage (Stage 4).

The tabular form used in the Fit/FitNesse test frameworks (illustrated later in Figure 2) is one way for the customer to produce executable test statements.8

Stage 2: XPI Prototyping
Consider a medical instrumentation application. At this point in development, domain-knowledgeable technical personnel produce an initial prototype to analyze the proposed embedded solution. The solution would revolve around numerical algorithms implemented in a mathematically oriented scripting language such as Matlab. This stage effectively defines the system’s functionality and the corresponding set of test objectives. This involves two steps:

- **Developing additional acceptance test cases.** During prototype production, interteam communication will lead to additional understanding of the system, captured as additional acceptance test cases.
- **Using TDD supported by a testing framework designed for Matlab development.**9 Despite the domain technical specialist being highly skilled in writing Matlab code, “formalized” testing tends to be an overlooked skill in the specialist’s repertoire. Adopting TDD is intended to resolve this problem.

In other words, whereas Stage 1’s acceptance tests provide the initial skeleton for the prototype, Stage 2 adds “flesh” onto this outline.

Stage 3: Initial XPI Production System
We must rewrite the system in a high-level compiled language such as C or C++. The biggest issues here surround the refactoring necessary to support moving Matlab floating-point calculations onto high-speed, low-cost embedded processors that can handle integer operations. This “translation” requires awareness of the issues of determining levels of signal scaling to maintain appropriate algorithmic accuracy.

Stage 3 basically repeats the pattern from Stage 2, but in a new programming language using a new unit-testing framework. However, because of the existence of the previous stages, we’ve already constructed numerous tests. So, much of this third stage is better described as a “design by contract meets TDD” process. This iterative, and gradual, form of producing contractual objectives should lower the task’s cognitive overhead, leading to “average” practitioners producing more-comprehensive contractual statements.

Stage 4: Full XPI Production System
By the fourth stage, we’re fully involved with the target embedded environment itself. The system, although repeating earlier patterns, now needs to adapt to its new hardware environment with limited memory resources and must support interactions with device-specific peripherals on the target system. The system undergoes further modification to tune for the required performance by

- rewriting inefficient programming constructs,
- using signal-processing-specific #pragma state-
ments to direct architecturally aware compilers, and
- rewriting critical components in low-level, highly customized machine code.

The unit test framework becomes a significant issue because the unit tests must now live on the embedded board. The transmission of any testing scripts from the host machine to the target, or the reverse transmission of test-relevant diagnostic information, becomes a major undertaking that possibly compromises real-time operations.

Refactoring
This central issue in TDD processes must be reexamined in light of this new environment. A recent study of software developers’ work habits by Emerson Murphy-Hill and Andrew P. Black suggests that developers use refactoring tools mainly for renaming functions or methods or for moving or extracting methods from classes; and experts perform more-complex refactoring by hand rather than with tool support, a concept that hardly seems agile (Murphy-Hill and Black’s comments also imply nonexperts don’t perform complex refactoring).

Although we need standard capabilities such as renaming in a Matlab refactoring tool, we also need features with no equivalence in desktop refactoring—for example, the ability to recognize and declare variables to automate the addition of the equivalent of int fooArray[XXX] for each dynamically allocated Matlab array. This would be useful in making automatic Matlab-to-embedded-system code translation tools more reliable and efficient. The advantages of a tool capable of refactoring classes are unclear; many developers avoid using Matlab classes, given the inefficiency with which translation programs convert these classes into code that embedded compilers can handle.

As for manual refactoring, it will be IT nonspecialists who will produce the Matlab code during Stage 2 prototyping. Would any real payback occur if they refactored the code at this earlier life-cycle stage? We believe the answer is no. Refactoring seeks to improve the code’s structure. However, early reorganization will likely hide possible embedded-system-specific efficiencies that, at Stage 4, would permit the embedded code to meet key real-time issues. Indeed, many standard refactoring practices are more appropriately applied only in Stage 4 of this embedded life cycle—and then only to the (many) embedded code sections that aren’t time critical. In addition, would the nonprogramming specialists in the (medical) embedded industry have the necessary capabilities or training for manual refactoring? We again believe the answer is no, despite unit tests offering some protection against the additional defects that manual processes are likely to introduce.

Embedded XPI Testing Frameworks
The following case studies investigate the issues surrounding development of the wide variety of testing frameworks needed to support our proposed XPI embedded life cycle.

Case Study 1: Supporting Continual Customer Involvement
As low-level embedded developers, we initially excluded any customer support component from our embedded life-cycle processes, assuming that only a low-level, embedded-xUnit, developer-oriented viewpoint was necessary. Initial experience and industrial feedback quickly disavowed us of this assumption.

We implemented a three-level Fit approach (see Figure 2) to fully support customer involvement. In Stage 1 (product envisioning), a Matlab-Fit server lets the customer get involved with the developers to explore the basic ideas behind the product. The developers then express these ideas as Matlab code (Stage 2) and then demonstrate to the customer, via Matlab-Fit, that they’re remaining true to the original concept. Using other servers lets the developers use the same customer tests when the product moves out of theory (Stage 2) and into simulation (Stage 3, using C++-FitNesse) and actual product testing (Stage 4, Embedded-FitNesse).

This work with Matlab-Fit was successful; however, several issues remain. Certain embedded test classes, such as those involving precise mathematical relationships, are easy to express through Fit tables. However, although existing Fit fixtures make representing images syntactically possible, there’s no real semantic support for Fit as a communication framework for discussing the image-related issues common in medical applications. Essentially, the support is limited to detailing whether or not corresponding pixels in two images are identical.

All the technical logistics are in place for Embedded-FitNesse, but true practicality currently eludes us. We could, for example, extend Fit tables to handle the timing relationships...
needed to generate the embedded system signals that activate devices accepting, rejecting, or packing medical samples on the basis of use-by dates. However, the Fit server (running in Java and C++ on a PC) must interact with the embedded development system API (proprietary code) to control the processor over an elongated test execution path (see Figure 2). Given the continual reloading of recompiled code occurring over this channel, the current Embedded-FitNesse prototype is too slow to act as a test framework to support practical acceptance-test-driven development.

Case Study 2: Generating Stage 2 xUnit Tools
We initially planned to support the latter three stages of the embedded life cycle using identical xUnit tests. The complex Matlab/C++ interface makes this possible but impractical.

We’re extending MUnit to enable us to use a TDD approach to perform fundamental research before tackling true product specification. For example, we need to modify this automated Matlab testing framework to better visualize whether a hypothesis test failed because of a fundamental, theoretical misunderstanding or simply because of a missing scaling factor during algorithm implementation. The standard MUnit visualization primitive “Press the PASS button if this picture looks correct” is totally inadequate for this purpose, in addition to being nonautomated. To compensate, we’ve extended MUnit to use new test approaches that extend the concept of what constitutes a failure beyond the basic interpretation of incorrectness (FAILURE() will “set the bar” red). We now permit a Gaussian error distribution to generate an ACCEPTABLE_FAILURE() (“set the bar” speckled green) if the experimental signals possess a low signal-to-noise ratio but force an EXPECTED_FAILURE() report (“set the bar” yellow) when the experimental signals have a high signal-to-noise ratio.

Case Study 3: Generating Stage 3 and 4 xUnit Tools
The agile community strongly holds one key tenet: tests should be written in the language of the code they’re validating. If that concept is worthwhile, then so should be its obvious extension: tests should be run on the system being validated. This approach is related to the hardware-in-a-loop testing ideas expressed at www.embedded.com/15201692. That requirement rapidly complicates the solution for developing an embedded system TDD framework, as we discussed earlier. Compared to desktop systems, there are limited memory resources, limited report-generating capabilities, and a stricter need to meet timing requirements.

Greene suggested taking an assembly xUnit path to handle embedded testing. With the wide range of assembly languages across many
processor families, we considered such an approach as both problematic and unnecessary given that C, a subset of C++, was originally designed to supply a universal (not processor-specific) assembly language. In a 2005 article on embedded TDD, we discussed modifying an existing CPPUnitLite testing framework to meet embedded systems requirements. When this framework is transposed and extended into the embedded environment as Embedded-Unit, its nonscripted nature can be used to full advantage to ensure that real-time embedded functionality isn’t disrupted by continual PC processor communication overhead associated with issuing test requests.

CHECK() is a fundamental assert in unit testing (see Figure 1) and is equivalent to the macro

```c
#define CHECK(condition) if (!(condition)) FAILURE(#condition);
```

Within CPPUnitLite, the low-level macro FAILURE() uses the C++ iostream (header file) to communicate with the host system, which retains the potential of causing a considerable impact on real-time operation. To provide a better match to the embedded environment, an additional FAILURE() channel was added within Embedded-Unit to make use of the JTAG Background I/O Telemetry Channel (BTC) available on current processors. BTC is a high-speed boundary-scan-based (JTAG) host-processor communications link that interferes minimally with normal processor operations. This is the first of many examples where co-opting specialized hardware in modern processors can overcome the obstacles to using agile methods in the embedded realm.

Additional forms of the CHECK() asserts proved useful given that a different set of variable types are experienced in the embedded domain than in the business environment. For example, a key byte pattern (0xF0) might arrive au naturel over an 8-bit Host Link port, hidden as part of a 16-bit serial peripheral interface channel packet (0x03F0), or automatically cast into a signed 32-bit value (0xFFFFF0) by the compiler.

In contrast to Embedded-FitNesse, Embedded-Unit has proved both practical and useful across a variety of processor architectures (single-core, multicore, and processor clusters). A local firm is performing trials using Embedded-Unit as a testing tool in an existing industry product involving a multithreaded natural-voice recognizer interacting with peripherals using a USB hardware stack. Embedded-Unit was primarily developed for the Analog Devices Blackfin (ADSP-BF5XX) processors. We have extended this embedded test framework to work with other processor families (Texas Instruments C64XX, C64XX+, and TMS470/ARM processors), in multiprocessor clusters (ADSP-TS201 TigerSHARC), and on multicore processors (ADSP-BF561 Blackfin).

### Case Study 4: Testing Standalone Embedded Systems

Don’t assume that porting tools into an embedded environment is simply a case of recompiling the code and hoping that it fits into the embedded system’s available memory. For example, consider the identification of race conditions in multithreaded applications. Currently, no tools deal effectively with general race conditions; most research focuses on handling data races. These conditions are notoriously hard to find because of their timing-dependent, nondeterministic nature.

Existing software-based dynamic detection tools rely on invasive instrumentation to pass information to the (host) runtime analysis engine. Readers familiar with Intel Thread Checker might have noted the following performance-related comment in the tool’s internal help pages:

> It is not uncommon for the memory to grow to twenty times the original memory requirements and for the execution time (CPU use) to increase to more than 300 times the original execution time.

The impact of such an instrumentation package is unacceptable given embedded systems’ real-time requirements and limited system memory. In addition, the overall slowness of such tools makes them inappropriate as part of a proactive TDD process oriented to identifying data-race problems in multithreaded code early in the design process.

To overcome these difficulties, we’ve prototyped a lightweight high-speed TDD framework plug-in, E-Race. This plug-in innovatively employs the hardware registers originally added to processors to help debug code by watching traffic along the data and instruction buses. With this approach, the original thread code runs without modification, with the instruction and data flow monitored by a low-impact hardware watchpoint unit combined with a short exception handler (see Figure 3 on the next page). Information on data race and other conflicts determined through the exception handler is analyzed through extensions to the existing Embedded-Unit testing-framework syntax.
Although using a hardware-assisted embedded TDD framework for data race analysis conceptually offers orders-of-magnitude performance advantages over software instrumentation, its actual implication isn’t straightforward. Although the issues discussed are specific to the Analog Devices Blackfin processor, we find similar issues when exporting this Embedded-Unit plug-in to other processors.

With the Blackfin processors, we had already successfully used the processor’s internal data watchpoint address and counter registers to develop WatchDataClass tests to monitor data activity.\textsuperscript{1,12} However, in a multithreaded test framework, action rather than passive monitoring is needed. With watch register hardware support, we can write non-software-instrumented tests that run in a threaded environment to check whether

- code segments that are setting and releasing locks are executing in an incorrect sequence,
- too many code executions are necessary to obtain a lock before access to shared variables is granted,
- a critical region of code isn’t being correctly protected, or
- appropriate locksets are absent when shared memory is being accessed.

We developed Embedded-Unit tests to investigate two aspects of the E-Race plug-in. First, we wanted to explore how to use the instruction watch registers in our design of a prototype hardware-assisted “happens-before” data race analysis tool—an application not envisioned by the original chip designers. We also wanted to ensure, using the hardware data watch register capability, that all accesses to shared memory from code occur under valid lock protection. This second issue involves a specific embedded issue given that, unlike Java, customized assembly code functions are frequently written to access memory with indirect pointer operations.

Unfortunately, writing code to satisfy the watch-data requirements failed because of hardware issues. The existing silicon implementation of the data watch registers throws the processor into an emulation mode, more appropriate for manual debugging than for an automated testing framework. Using Embedded-Unit to provide a stable hardware design environment, we explored new approaches that took into account two facts: we were using specialized hardware debug registers in a manner that the original silicon designers couldn’t have envisioned, and these debug registers didn’t behave as we had anticipated on the basis of the manufacturer’s documentation.

Without going into a great deal of low-level detail, it’s sufficient to say that we obtained a solution by setting the hardware watch registers to “sparingly” trace the program execution flow rather than examining the processor state at every instruction. Although we implemented this solution to solve hardware imperfections, it proved the key to meeting the performance requirements we needed for using E-Race in a testing framework capable of supporting TDD processes. Acquiring a data lock might take only a single C++ instruction in a thread-based application, but the same operation requires multiple machine instructions. There’s no need to watch “every” instruction for possible data manipulation operations. Compared to the 300-fold performance loss and 20-fold increase in code size reported for the Intel Thread Checker software tool, using the hardware-assisted E-Race tool for data race identification incurs only a 5- to 8-fold performance overhead, and an increase in memory requirements of only several hundred lines (not Mbytes) of assembly code, independent of the size of the code being checked.

The E-Race plug-in has demonstrated a further way that the wide range of hardware capabilities in modern processors can be co-opted to enable the testing frameworks necessary to support an XPI embedded life cycle. Other examples include the use of hardware registers that monitor code efficiency without additional software overhead, thus opening opportunities for nonfunctional tests that can directly evaluate power consumption associated with particular code variants for handheld embedded applications. A wide range of timers are available on modern microcontrollers. This provides a redundancy that allows the use of timers for standard purposes while simultane-
ously permitting the hardening of the testing environment so that it continues to function despite the nonarrival of signals (hardware handshaking) between the processor core and external peripherals. We’ve also explored applying a hardware program trace buffer to provide a low-impact alternative to the software instrumentation normally used for analyzing test coverage.5

Our XPI embedded life cycle deviates from accepted XP concepts in fundamental ways, but it offers a path to solve many embedded issues. Currently, our TDD approach to support embedded system customer acceptance tests through Matlab-Fit and Embedded-FitNesse is best described as showing “potential.” In contrast, we’ve had measurable success with the xUnit side of embedded TDD frameworks to support the XPI life cycle. MUnit can help generate a Matlab-executable specification document, but it requires further modification to be truly useful. Embedded-Unit, based around the non-scripted CppUnitLite, provides an easy-to-learn TDD framework that operates within the limited resources of a single-processor, multiprocessor, or multicore environment without disrupting the system’s real-time operation. This tool has proved useful in identifying previously undocumented silicon issues and has been extended to support multithreaded application development.6

Acknowledgments

Financial support was provided through a collaborative R&D grant from Analog Devices (in Canada and the US) and the Natural Sciences and Engineering Research Council of Canada (NSERC), the Province of Alberta, and the University of Calgary. We thank A. Geras of Ideaca Knowledge Services for producing the prototype of the Matlab-Fit tool and for many useful discussions related to industrial applications of agile methodologies.

References