E-TDD – Embedded Test Driven Development a Tool for Hardware-Software Co-design Projects

Michael Smith[1], Andrew Kwan[1], Alan Martin[1], and James Miller[2]

[1] Department of Electrical and Computer Engineering
University of Calgary, Calgary, Alberta, Canada T2N 1N4
smithmr@ucalgary.ca
[2] Department of Electrical and Computer Engineering
University of Edmonton, Edmonton, Alberta, Canada T6G 2V4
jm@ee.ualberta.ca

Abstract. Test driven development (TDD) is one of the key Agile practices. A version of *CppUnitLite* was modified to meet the memory and speed constraints present on self-contained, high performance, digital signal processing (DSP) systems. The specific characteristics of DSP systems required that the concept of refactoring be extended to include concepts such as "refactoring for speed". We provide an experience report describing the instructor-related advantages of introducing an embedded test driven development tool E-TDD into third and fourth year undergraduate Computer Engineering Hardware-Software Co-design Laboratories. The TDD format permitted customer (instructor) hardware and software tests to be specified as "targets" so that the requirements for the components and full project were known "up-front". Details of *CppUnitLit* extensions necessary to permit tests specific for a small hardware-software co-design project, and lessons learnt when using the current E-TDD tool, are given. The next stage is to explore the use of the tool in an industrial context of a video project using the hybrid communication-media (HCM) dual core Analog Devices ADSP-BF561 Blackfin processor.

1 Introduction

Many commentators agree that we are on the threshold of new era of computing characterized not by the expansion of desktop or mainframe systems but by a new breed of embedded (and invisible) product entering the marketplace. Third era (3E) computing products will contain a processor, rather than "a computer", and include everything from wearable phones, to guiding the blind using intelligent GPS, to drive-by-wire automobiles, to wireless remote sensing of patient life signs. These 3E systems promise to become all-pervasive in modern life of the industrialized world. No longer will there be the traditional relationship of one computer per person; but instead each individual will be served by many machines that will be highly embedded, fully interconnected and often highly mobile.

While these new systems provide a highly exciting view of the next era, the Information Technology (I.T.) industry has a steep hill to climb to play its part in the pervasive computing revolution. These products will often have extreme characteristics when compared with current existing systems. On the software side, dramatic increases in, for example: the safety-related properties; the reliability and availability of software-intensive systems. The very expected mobility of the software (including

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 145–153, 2005.

object-code) and data imply increased security problems that will move the software industry into new intellectual and technical domains. On the hardware side, the issues are equally demanding whether with embedded systems or the envisioned more complex and diverse ubiquitous systems. The problems include the system level validation of mixed signal computations (analog, digital, wireless), hard real-time and throughput requirements, size, weight, area and power (SWAP) constraints, quality of service, environmental effects, *etc*.

According to a newly released study commissioned by the National Institute of Standards and Technology (NIST) [1], software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually. The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved verification infrastructure that enables earlier and more effective identification and removal of software defects. Early identification of defects is more than just desirable with the new systems, as the firmware in the product may not be upgradeable; being burnt directly into the custom silicon. The concepts of "Getting it right the first time" [2] and the "Blue screen of death" gain a real personal interpretation when applied to the firmware inside YOUR surgically-implanted heart defibrillator!

Although challenges exist right across the board, we argue that if these types of systems are to be successful then the production methodology needs to be concentrated on the verification and validation of these systems and that these activities must drive the production process! Many traditional production processes, such as the waterfall or spiral models defer the key faultfinding verification and validation activities too late in the life cycle, resulting in these components often being considered as an "afterthought" to the production component. Further the "testing" activities often get disassembled into artificial sub-activities such as unit testing, integration testing and system testing. The popular "V model" of an integrated testing process is a good example of this type of structuring; this model results in the production team being forced into conducting these sub-activities regardless of the relative cost-benefit issues associated with each of these sub-activities. Clearly in any arbitrary project, the costs and benefits associated with any verification and validation activity will be highly dependent on the domain of operation and the product under development.

Since testing now accounts for more than half of the costs on many projects, any production methodology that fails to actively consider the costs and benefits of the testing activities is potentially wasting an extremely large amount of resources and failing to perfect the product for the marketplace. In addition, Pervasive Computing Systems are highly likely to present a different and potentially more demanding verification and validation puzzle, further suggesting that an alteration to current practices are required. Even in comparatively simple embedded systems, system testing is an undertaking that is often beyond the capacity and ability of many organizations; and hence this component is often less than satisfactory on release date. This implies that even large well-organized companies, utilizing the latest technology, struggle to meet the demands of system testing. Unfortunately, these demands are set to explode as the numbers of critical non-functional requirements multiply.

Recently the software engineering community has started to migrate towards a design process specifically intended to minimize the number of faults during development of software. Considerable effort has been directed towards Test Driven Devel-

opment (TDD) and Extreme Programming (XP) as initial attempts towards establishing such a fault-intolerant design process. However the focus to date has been directed towards functional correctness for business / desk-top applications. We propose to adapt and extend these initial ideas to develop and demonstrate a fault-intolerant design framework initially for embedded systems, and subsequently for ubiquitous 3E devices described in the earlier paragraphs of this introduction.

However, it is very quickly obvious that embedded system Extreme Programming (E-XP) and embedded system Test Driven Development (E-TDD) require the acquisition of a new mind set from the traditional Agile development team familiar with delivering functional incremental software releases combined with improvement ("refactoring") of existing code. By comparison, the new systems may well be monolithic with system components having low cross-cohesion and cross-coupling, but yet "nothing works unless everything works!" The real time functionality, mobile nature, high-volume and low margin found with these systems implies that refactoring may refer to "speed improvements" and "reduced power consumption", even when these factors involves many tradeoffs between portability, clarity and modularity [2].

In this paper, we describe the modifications necessary to permit a test driven development tool, *CppUnitLite* [3], to be used with embedded systems where every last cycle, watt and developer's work-second count in releasing a product. We detail the advantages of using the prototype E-TDD tool as a teaching tool within an undergraduate "hands-on" hardware-software co-design laboratory course in Fall 2004.

2 Implementation of a *CppUnitLite* TTD Tool Variant to Meet the Requirements of Embedded System

A key identified problem is that embedded systems software development is frequently limited to the compiler environment and (very low level) debugging tools that examine the hardware interface directly. In addition, as embedded systems are real-time and performance constrained, it is difficult to gather extensive execution tracing logs, or capture the entire execution context [2] without completely disrupting the functionality of the system.

We chose to adapt the small *CppUnitLite* developed by Michael Feathers [3]. A key element in adapting this tool was to minimize memory usage to ensure that it would fit within the embedded system environment. In particular, any print statements which involved formatting were replaced by simpler statements (***puts(astring)***). The sheer generality of the formatting associated with statements such as **cout << value** or **printf("%d", value)** can generate sufficient instructions to occupy most of the available program memory space available (on-chip) within the target system's processor; possibly leaving little room for other code.

Further reason for choosing *CppUnitLite* as a basis for the E-TDD tool was the fact that it was "non-scripted" means that the framework detects the tests to be run automatically, freeing the developer from working with a script running on an external development environment (PC). Such external scripts require that the embedded system be stopped in order to run, or report on, or be interrogated about, a specific test. Messages are sent back from the target to the development environment over a serial, USB or JTAG connection. Basically the target must be stopped and switched into "emulator interrupt" mode. This permits the message to be "wangled" out of the tar-

get “one character at time”, but completely disrupts the real-time operation of the embedded systems. For some of the development environments examined, there is a “background telemetry channel (BTC)” [4] which we have investigated as a mechanism to permit reports, in particular failure reports, from the non-scripted tests to be interchanged with the external development environment with no, or minimal, disruption of real time operation.

The following listing, **test macro elements bolded,** can be considered classical (embedded) TDD. The test is for the validation of the response of a finite impulse response filter. Initial values are established, before the test macro **CHECK( )** assert is used to generate a report on the validity of the result from the *FilterASM( )* function under test. For an FIR filter, the output is equal to the j^{th} filter coefficients when an impulse vector ($v_i = 0 \leq I < \text{NEEDED}, v_j = 1$) is input to the filter.

```
#define NEEDED FIR_length
TEST(FilterASM_impulse, DEVELOPER_TEST) {
  int value, test[NEEDED],coeffs[NEEDED];
  // Impulse response tested
  Set_FilterCoeffs(coeffs, NEEDED);
  for (int i = 0; i < NEEDED; i++) {
     for (int j = 0; j < NEEDED; j++) {
         test[j] = 0; coeffs[j] = j;
     }
     test[i] = 1;
     value = FilterASM(test, coeffs, NEEDED);
     CHECK(value == coeffs[i]);
  }
}
```

In the following listing, some new features of E-TDD are introduced. The real time nature of the embedded environment implies that it is critical that certain functions be guaranteed to perform within a specified time period. Here the **MEASURE_EXECUTION_TIME( )** macro automatically uses the on-chip clock to measure the execution (performance) time of *Function(parameters)*. Whether this execution time meets critical performance characteristics can be checked through the **MAXTIME_ASSERT( )** and **MAXPOWER_ASSERT( )** statements. These are the first of a series of functional and non-functional tests for embedded systems planned for development.

```
void FilterRelease(float* , float*, int);
void FilterASM(float* , float*, int);
TEST(SPEED_REPORT_Filterfloat, CUSTOMER) {
  float *pt, test [NEEDED];
  float coeffs[NEEDED];
  unsigned long int timeRELEASE, timeASM;

  Set_FilterCoeffs(coeffs, NEEDED);
  EstablishTestData(test, NEEDED);
  MEASURE_EXECUTION_TIME(timeRELEASE,
       FilterRelease(test, coeffs, NEEDED));
  MEASURE_EXECUTION_TIME(timeASM,
       FilterASM(test, coeffs, NEEDED));
```

```
  for (int i = 0; i < 100; i++) {
    MAXTIME_ASSERT(timeRELEASE,
      FilterASM(test, coeffs, NEEDED));
  }
}
```

The following test macros demonstrate some specialized embedded system extensions **WatchDataClass( )** and **WATCH_MEMORY_RANGE( )**

```
#define SCALE 0
#define PERIOD 0x2000
#define COUNT 0x2000
  typedef ulong unsigned long int;
  void SetCoreTimerASM(ulong, ulong, ushort);
  TEST(Test_SetCoreTimer, SET_UP) {
    __SaveUserRegAndReset ( );
    WatchDataClass <unsigned long> coretimer_reg(
            4,  pTCNTL, pTPERIOD, pTSCALE, pTCOUNT);
  // Setup expected final memory mapped register values
    ulong expected_value[] = {0x0, PERIOD, SCALE, COUNT};
    WATCH_MEMORY_RANGE(coretimer_reg,
             (SetCoreTimerASM(COUNT, PERIOD, SCALE)),
                              READ_CHECK | WRITE_CHECK);
    __RecoverUserReg( );
    CHECK(coretimer_reg.getReadsWrites( ) == 4);
    ARRAYS_EQUAL(expected_value,
                    coretimer_reg.getFinalValue( ), 4);
  }
```

The **WatchDataClass( )** and **WATCH_MEMORY_RANGE( )** macros to provide the ability to watch (in real time) the performance of specific processor and peripheral memory mapped registers during system initialization and time critical sections of code. These tests should be considered as an embedded extension to, rather than a strict departure from, the "oracle" style of classical TDD since a given computation is run and its output, the number of memory mapped register hardware operations performed and the results of those operations, compared to values predicted in advance. These tests make use of "hardware instruction and data breakpoints" [4] on a running system rather than via "static profiling" on an architectural simulator or "statistical profiling" (occasional snapshots of the program counter) on a running system. The timer and watch-data E-TDD classes are general in concept, but must be specifically implemented using processor resources , and at the same time, not remove resources need for the normal development of code.

The **WatchDataClass( )** class was initially developed for use by an expert (*e.g.* an instructor with very intimate prior knowledge of the system architecture) might write such an EXPERT test to automatically examine whether those developing code for the processor have properly configured the registers of a peripheral correctly. Both register values and register access operations are evaluated *e.g.* checking that registers have been specifically set to the required values rather than left with the default (reset) values which "just happen" to be the same as the required values. However, in practice, this test class proved to be unexpectedly much more utilitarian.

- From personal experience, this has proved to be a test that is useful when the developer knows exactly what needs to be done, and how to do it, but gets distracted by an interruption in the middle of the creative process.
- The 3rd year undergraduate class, in the very first assignment after being trained with E-TDD in a hardware-software co-design laboratory, discovered unexpected (undocumented) behaviour of a new processor's core timer resources. Such behaviour could play (could be playing) havoc within industrial products if it remained unrecognized. The students, through the methods of the **WatchDataClass** test class, were able to identify, and then stabilize, the error's behaviour (to prove that it existed) so that the problem could be reported to, and recognized as an issue by, the chip manufacturer [5][6].

3 Experiences and Lessons Learned

The modified test-driven development environment source code for E-TDD has been compiled, linked and downloaded to a number of embedded systems; a single instruction single data (SISD) processor (ADSP-BF533 Blackfin), a single instruction multiple data (SIMD) processor (ADSP-21161) and a variable length instruction word (VLIW) processor (ADSP-TS201 TigerSHARC). Given that the basic TDD code was written in "C++", we were not expecting any compatibility issues across these processors from Analog Devices, nor across processors from other manufacturers. However, we were concerned about code size issues associated with the memory constraints across such a wide range of processors designed for different applications.

A key issue element of the TDD is that all tests be available for running at all times. The non-scripted environment, with its test macros expanded, proved to make use of a larger amount of heap space than had been anticipated, and not necessarily available within the constraints of an embedded system. Three approaches are being used to solve this issue. First a menu driven system has been created to provide automated selective linking to the test library banks. Use of a second C++ heap within external memory for test storage is also being explored. External memory can be two to ten times slower than the internal chip memory because of bandwidth and other issues. However this speed difference is not expected to be critical, since it is the tests themselves, rather than the functions being tested, that are stored in the slower memory. Finally, the report information can be reduced to the passage of tokens requiring minimal memory storage, which are then expanded for use within reports generated by the development environment running on an external work-station.

The current E-TDD prototype provides the ability to set the hardware environment to a known state prior to issuing a series of tests. To capture this functionality, three new hardware oriented TDD procedures were developed

__CaptureKnownState( ) – This procedure is called as the first line of a *main( )* function run on a system that has just been powered up. It automatically captures and saves the "C++" initial environmental setup to a file.

__SaveUserRegAndReset( ) saves the current user processor state, and resets the system to the known state established by the *__CaptureKnownState( )* procedure. *RecoverUserReg( ),* restores the initial user processor state.

Although a fundamental assumption underlying the TDD "oracle" style of testing is to set the system into a known state, the jury is out on the utility of applying this style of

testing in an embedded environment. It is true that having the system in a known state before testing prevents many errors. However, testing with a system that is unintentionally in an unknown state does also uncover unexpected system configuration issues. In addition, resetting the system's registers prior to individual tests can be "a highly delicate, and difficult to perform, process" as we discovered recently when adjusting multiple instruction and system caches on the very long instruction word (VLIW), highly parallel, high speed TigerSHARC (ADSP-TS201S) digital signal processor [4]. The resetting of running hardware attached to the embedded system is also not something that can be treated casually.

The long term goal is to use E-TDD within an industrial environment using hybrid communication-media (HCM) processors. However the prototype tool has already proved to offer many immediate advantages to the undergraduate instructor [5][6]. The tests used by the instructor when developing "hands-on" embedded system laboratories make excellent presentation tools to provide the students with an insight of the requirements of all aspects of a hardware software co-design laboratory for both high and low level tasks. The instructor developed tests then form the basis for a bank of "customer" tests that students must satisfy to demonstrate their own design. For tests to have any meaning, their code must be made available for inspection by the tester rather indicating the failure of "secret" test conditions. However, the availability of the provided "high-level" customer test bank meant that many students did not generate their own developer tests during the initial stages of their product development; defeating the purpose of TDD training. We added successful, as well as failed, tests reports; providing the students with an initial comfort zone. These experiences have provided us with sufficient confidence to move to the next stage: using the E-TDD tool within an industrial context of a video project on the hybrid communication-media, dual core, Analog Devices ADSP-BF561 Blackfin processor.

By definition, since tests are developed before the code, there are many error messages from the immediately available "customer" tests covering all stages of the project generated when the tests are first run. This problem was significantly reduced after refactoring the testing environment so that associated groups of tests could be gathered into individual object files within a project library file, and then linked at will. In this manner, a programming pair would be able to activate the "customer" tests for the current and earlier incremental development phases of a project without being over-whelmed by expected failure messages from tests for later project stages.

While we believe that we have made great progress, and that the direction is right, we also believe that we are just scratching the surface of what is required to adapt agile processes into an embedded environment. Of considerable interest is extending the ideas of Feathers [9] on adding tests to "legacy systems", into the environment of testing the custom off the shelf (COTS) software commonly used with embedded systems. Adding nonfunctional unit testing facilities for hard deadlines

```
e.g. MAXTIME_ASSERT(timeRELEASE,
                    FilterASM(test, coeffs, NEEDED));
```

was relatively straight forward. However adding nonfunctional tests for soft deadlines is more demanding. Soft deadlines can often be examined at the unit level, but the test is passed / failed over a number of executions rather than a single run. Expression of

the pass / failed statement is problematic, as this decision usually involves some level of performance tolerance rather than any strict hard limit.

In addition, refactoring [7] becomes a more complex issue. While embedded systems will be refactored to improve code quality; it is equally likely that they will be refactored to increase performance, or decrease power consumption, or modify other specific and unique, but not code-related characteristics. Again, this takes us beyond the available literature and the definition of the technique. And, in fact, the complex relationship between code quality / maintainability and performance becomes a core concern in many embedded applications, while often ignored as non-critical in many desktop situations.

4 Conclusions

A key identified problem is that embedded systems software development is frequently limited to the compiler environment and (very low level) debugging tools that examine the hardware interface directly. In addition, as embedded systems are real-time and performance constrained, it is difficult to gather extensive execution tracing logs, or capture the entire execution context without completely disrupting the functionality of the system. These problems will become exacerbated as these embedded systems become evermore ubiquitous in nature and require the ability to run for extended periods without experiencing any significant faults.

We believe that agile methodologies, including pair programming [8], will play an important role in developing these zero-defect oriented devices of tomorrow. To-date, we have experimented with test driven development and unit testing. Evidence has been provided to demonstrate that current unit testing tools can be successfully adapted to work in an embedded environment. While the adaptation is not necessarily straightforward; it is viable. Our experiences in adapting these tools also strongly suggests that they need to be extended to explicitly cover the testing of non-functional requirements which often play a critical role in embedded and ubiquitous environments. We have demonstrated these needs by examining the testing of hard deadlines within unit testing. In our future work, we plan to undertake a rigorous experiment to validate the hypothesis that we are already witnessing a more rigorous production process, especially targeted at the elimination of defects during embedded system design and development.

Acknowledgements

Financed under a collaborative research and development research grant involving Analog Devices (Canada) and the Natural Sciences and Engineering Research Council (Canada). MS was awarded the Analog Devices University Ambassadorship for 2002 – 2005. Discussions with A. Geras on agile development were appreciated.

References

1. G. Tassey, “The Economic Impacts of Inadequate Infrastructure for Software Testing”, National Institute of Standards and Technology Report, 2002.
2. D. Dahlby, “Applying Agile Methods to Embedded Systems Development”, *Embedded Software Design Resources.,* Vol.41, pp.101-123, 2004.

3. M. Feathers, “CppUnitLite Source code”, http://c2.com/cgi/wiki?CppUnitLite (Dec. 2004).
4. Analog Devices Blackfin and TigerSHARC DSP “User and hardware manuals’, http://www.analog.com/dsp (Dec.2004)
5. M. Smith, A. Martin, L. Huang, M. Bariffi, A. Kwan, W. Flaman, A. Geras, J. Miller, “A look at test driven development (TDD) in the embedded environment: Part 1”, Circuit Cellar magazine, Vol. 176, pp 34 - 39, March 2005.
6. M. Smith, A. Martin, L. Huang, M. Bariffi, A. Kwan, W. Flaman, A. Geras, J. Miller, “A look at test driven development (TDD) in the embedded environment: Part 2” Circuit Cellar magazine, Vol. 177, pp 60 - 67, April 2005.
7. M. Fowler, “Refactoring: Improving the Design of Existing Code”, The Addison-Wesley Object Technology Series, 1999.
8. L. Williams, R.R. Kessler, W. Cunningham, R. Jeffries, “Strengthening the case for pair programming”, IEEE Transactions on Software Engineering, pp. 19 – 25, 2000.
9. M. C. Feathers “Working Effectively with Legacy Code” Upper Saddle River: Prentice Hall PTR, 2005.