Static Verification and Extreme Programming

Peter Amey, Roderick Chapman

Praxis Critical Systems
20, Manver Street
Bath, BA1 1PX, UK
+44 (0)1225 466991

peter.amey@praxis-cs.co.uk
rod.chapman@praxis-cs.co.uk

ABSTRACT

At first glance, the worlds of high-integrity software engineering and Extreme Programming (XP) seem to have little in common. Somewhat surprisingly, we have found the reverse to be the case—indeed it seems that many practices advocated by the XP community are familiar to us from many years' of experience in building safety- and security-critical systems. This paper discusses our experiences in applying some XP practices in critical projects. Secondly, we discuss how static verification can augment XP, particularly in the Pairwise Programming and Refactoring practices.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**] Software/Program Verification

General Terms

Design, Languages, Verification

Keywords

Extreme Programming, Static Verification, SPARK, Ada, Information-flow analysis, Program Proof.

1. INTRODUCTION

At first glance, the terms "High Integrity" and "Extreme Programming" in the same sentence might seem to be a contradiction in terms! While the word "extreme" might seem at odds with the naturally conservative and rigorous world of high-integrity engineering, we do not believe this is the case.

In his book, Beck[2] describes XP in terms of 12 core practices:

- The Planning Game
- Small Releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards

Most of these are not new or radical at all: they are well tried and tested ideas that have been known to the software engineering community for some time. Practices such as regression testing, continuous integration, collective ownership, and the use of coding standards should not come as a surprise to anyone involved with the development of high-integrity software systems.

Upon discovering XP, we were both surprised and pleased to find how much XP *we already do* on high-integrity projects. This was not as the result of some top-level management decision to adopt XP, but rather the discovery that XP advocates many of the practices that we've been doing for years as a matter of course. We also found some interesting differences where our practices differed from the XP viewpoint, or areas where XP didn't seem to address particular problems in high-integrity development. The remainder of this paper discusses these discoveries and insights, particularly focusing on the *pair programming*, and *refactoring* practices.

2. PAIR PROGRAMMING

XP advocates the use of "pairwise programming." This literally means having two engineers sat at a single desk, using a single computer, designing and creating code as a cooperative activity. Initially, this seems odd—how can two people sat at a single computer be more productive than two people working alone? The XP community claim it is—that such pairwise activity is actually more productive at producing real, working, *delivered* code. We note the distinction here—you can certainly produce more poor code working alone, but that's hardly the object of the exercise.

Beck points out that the members of a pairwise team are thinking in different modes. The person at the keyboard is thinking about how to implement the required functionality in terms of data structures, algorithms and so on. The partner is thinking "more strategically"—considering the whole design, which test cases are going to need correcting or adding, how the current solution fits within the overall design and so on. This distinction of roles is important, since problems overlooked by one person are more likely to be spotted by the partner. This is akin to moving the traditional code-review process up the lifecycle until it literally

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'03, December 7–11, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-476-2/03/0012…$5.00.

becomes “live” with one partner reviewing the code right there and then while it’s being constructed.

In the high-integrity world, we are used to the application of automatic static analysis technology in the construction and verification of software. Consider the properties that make a “good” partner in the XP world. Such a partner would be excel at considering problems not usually thought of by the code author, and the partner would be *pedantic*—not letting a single slip or defect go un-noticed. Sounds like a static analysis tool to me.

In the development of high-integrity software, we often use such tool support—the SPARK[1] Examiner, for instance, is written in SPARK[1] and is used in its own development. So does the combination of human designer plus a static analysis tool count as a “pairwise” team in the XP world? We would argue that it does. Consider the problem domains where human brain-power and tools differ:

- My brain is good at: requirements, safety and security issues, non-functional trade-offs such as efficiency, testability, and so on. (On the other hand, tools are not much use in these domains.)
- A tool is good at: being pedantic, data-flow analysis, information-flow analysis, static semantic analysis, subset checking, abstract interpretation, theorem proving. (All areas where my brain is severely limited.)

These sets of problems seem complementary: tools are good at problems that I can’t do in my head and vice versa. We therefore apply tools are far as is practical. SPARK, in particular, allows the tool-supported problem domain to be “turned up to 11.”[6] My (human) partner can then review my code *knowing* that the tools have already taken care of a great many problems. Reviewing time is then more productively spent considering the issues that really matter such as safety and security requirements.

Oddly, Beck’s book hardly mentions static analysis at all. How could the XP community have missed this point, when it is so well-known in the high-integrity domain? We offer two possibly reasons for this:

- The *depth* of static analysis available with traditional languages is limited by the ambiguous nature of their definition and the intractability of many static analysis problems on those languages. In short, if you’re using traditional, unsubsetted languages the dials only go up to 4! This issue limits the usefulness of static analysis, and so it remains less widely used than it might.
- The *efficiency* of any such analysis is crucial. If you want someone to use a tool in *preference* to compiling and testing, then the tool must be *fast!* Again, the intractable nature of many static analysis problems on traditional languages has limited the adoption of these tools. Beck illustrates this point with his “learning to drive” analogy—you need feedback as often as possible in order to make many, small corrections to a process.

3. REFACTORING

Fowler introduces refactoring[5] as “…the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.” The “*does not alter the external behaviour”* bit is important, since you want to be able to verify that the system still behaves as expected after the refactoring has been performed. Trying to add functionality at the same time as refactoring therefore hinders this ability. How do we know that the system’s behaviour has been preserved by such a redesign? The XP community’s mantra “test, test, test…” applies here. While we certainly agree that the regular and pedantic application of regression testing is useful, the limits of testing for ultra-dependable systems are well known[3][4]. Isn’t there any more that we can do?

Once again, static analysis has a role to play here, but this has not been widely recognized outside of the high-integrity community. Consider data-flow analysis: if we know that a program is statically free from data-flow errors prior to a refactoring, then it seems reasonable to expect that property to be preserved by the redesign—if it were not, it would almost certainly be indicative of a bug! What other properties can be statically verified across a refactoring? The answer largely lies in the *contractual strength* of the language under analysis. Consider the following function specification in Ada:

```
   function Sqrt (A : Natural) return Natural;
```

What properties (or “contract”) does this specification promise to its users? Actually, not much—the function promises to take a Natural parameter and to return a result of type Natural. The name “Sqrt” is something of a red-herring—this function might compute something like the square-root of its argument, but that is in no way guaranteed or implied. The function could do pretty much anything.

A conscientious Ada programmer might do better, writing:

```
subtype Sqrt_Domain is Natural range 0 .. 10000;
subtype Sqrt_Range  is Natural range 0 .. 100;
function Sqrt (A : Sqrt_Domain) return
                                     Sqrt_Range;
```

This is better, but still leaves many details unspecified. There is enough detail here for a compiler to generate code for a call to Sqrt, but not much else.

To enable static analysis to be useful in refactoring, we need more information. In SPARK, we can go as far as giving a specification of partial correctness for the function:

```
subtype Sqrt_Domain is Natural range 0 .. 10000;
subtype Sqrt_Range  is Natural range 0 .. 100;
function Sqrt (A : Sqrt_Domain) return
                                     Sqrt_Range;
--# return X =>     X ** 2 <= A and
--#             (X+1) ** 2 >  A;
```

Note also in SPARK that functions *never* have side-effects, so this property is implicit in the specification. Now we have enough information to make a static refactoring work. We could change the implementation of Sqrt (say from a binary-search, to a hardware-style “bit-bashing” algorithm) and re-verify that the implementation still meets the contract using the various analysis techniques at our disposal—in this case flow-analysis, verification condition generation, and theorem proving. Moreover, this level of verification can be achieved *prior* to re-testing of the system as a whole.

Even without going to the lengths of partial correctness proofs, we have found that the level of static analysis supported in “minimal SPARK” to be very useful in protecting the integrity of large systems during significant refactoring efforts. In particular,

[1] The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on SPARC™ architecture.

SPARK requires the specification of global data access and mode to be attached to all subprograms. Packages must also announce the existence of their own static state and the manner of its initialization. Many common refactorings involve moving the persistent state of a system (for example, changing an abstract state machine package into an instance of an abstract data type). SPARK protects the engineer from mistakes, since the top-level information-flow for a subsystem can be shown to be preserved even if the internal structure and state is completely altered.

Once again, though, we find that this static-analysis based approach to refactoring is little known. Only two languages support the level of design-by-contract™[2] in specifications required to really make the approach useful: SPARK and Eiffel. Only SPARK emphasizes static verification of contracts over the dynamic (i.e. testing).

With dynamic verification, XP advocates a "write the tests first" approach. Is there an analogue of this approach in the use of static analysis? It seems there is—we have long supported a "write the contracts first" design approach with SPARK, where the contracts of units are written as a design activity and then rigorously checked against the eventual implementation using the SPARK Examiner.

3.1 Refactoring tests

We often come across systems that are crying out to be refactored. High-integrity systems, in particular, typically have a long lifetime, and requirements change both during development and maintenance. Such systems are clearly candidates for regular refactoring, yet we often find this is not done: changes are added which distort the original design, rather than allowing the original design to be refactored so that the new requirements may be implemented elegantly. Why is this so? A number of reasons are often cited:

- "Because the code is finished." There seems to be a strong psychological effect whereby "finished" code is deemed to be set-in-stone and never to be touched again.
- Deadlines are always "too tight" to allow for a refactoring activity. Very few projects we've ever seen actually *plan* for refactoring.
- Refactoring code might be easy, but re-working a large test set is seen as prohibitively expensive.

The latter point is interesting. XP says you can refactor and all the tests should still work with no changes. This is true for "black box" requirements based tests—of course, you should be able to refactor the inner working of a system while preserving the top-level behavior. Unfortunately, in the high-integrity world, we often have somewhat Draconian requirements for structural test coverage. Such tests are definitely affected by refactoring: we have often heard that a piece of code cannot be re-designed, not owing to the cost of changing the code, but because "it will break all the coverage tests."

We do not have a simple solution to this issue. Collecting coverage data as a side-effect of requirements based tests is certainly a good idea, but writing structural-based coverage test cases just for the sake of getting a particular (typically "100%") coverage result seems to be of dubious value, especially if these very test cases inhibit the application of other useful activities such as refactoring. One would hope that the simplification of code resulting from a refactoring would actually improve coverage results. We could therefore propose a principal that says: using a stable set of requirements-based tests, refactoring should only ever be allowed to maintain or improve structural coverage of the code being refactored. Or, put another way, if a refactoring degrades coverage, then you probably shouldn't be doing it!

4. OTHER XP PRACTICES

This section offers some brief comments on some of the other core XP practices.

4.1 Coding Standards

The use of coding standards is well established in the development of high-integrity software. The rules in such standards range from simple lexical conventions ("Indent by 3 spaces and don't use tabs...") to semantically subtle program properties ("Functions shall not have side-effects.") We often find two problems. Firstly, coding standards evolve within a company or project based on historical events and the sometimes rather personal whims of the engineers involved. Secondly, such standards are often enforced by entirely manual review, rather than through the use of tools, which is inefficient and often incomplete, especially for the more subtle rules—no human I know of can do global data-flow analysis in their head! We strongly advocate the use of well-designed language subsets supported as far as possible by tools. On SPARK projects, our coding standards are remarkably short (rule 1 is "The code shall be SPARK as defined by [...]") with only a few additional guidelines for non-SPARK code. We also set goals for static analysis as entry criteria for subsequent code review, such as requiring no unjustified flow-errors, or setting a target for the number of verification conditions to be automatically proven by the theorem-prover. We find such criteria more meaningful and useful than the simpler forms of complexity metric gathered by other tools. We always enforce simple lexical style rules, often using the built-in support offered by recent Ada compilers.

4.2 On-site customer

This practice advocates having a customer as a full-time member of your software development team. In the (unfortunately) rare cases where we have been able to do this, we find this practice to be of use. Customers come with a large amount of built-in knowledge, particularly relating to the environment and domain of the system that you're building, which acts as an excellent "ambiguity detector." In interpreting and implementing a particular requirement or function, an engineer is prone to jump to a possibly wrong conclusions "Oh, that means X...". An on-site customer is likely to respond "Actually, no it means Y...", exposing ambiguity and incompleteness immediately, rather than delaying such discovery until later in the life-cycle.

4.3 Simple Design

Beck sums up this practice as "Design the simplest thing that will work now" for each integrated system build, rather than going for a "Big Up-Front Design"[3] that tries to encompass the entire

[2] "Design-by-contract" is a trademark of Interactive Software Engineering Inc.

[3] One XP advocate has coined this approach "BUFD the project slayer..."

planned system. The motivation for this is that requirements will invariably change, so most of a "big up-front design" ends up being thrown away anyway, and a simple design can always be re-factored later to add or adapt to requirements change.

While this Occam-like approach to software design has some merit, we find one particular difficulty with it from out perspective of building high-integrity systems: the need to accommodate non-functional requirements, such as safety, security and real-time properties. From our experience, it is very difficult (or at least prohibitively expensive) to "refactor in" such non-functional properties. Software safety and security properties are often achieved by architectural means (such as partitioning), which *have* to be considered from the outset of a project, particularly if that project is to be subject to an external audit or inspection by a regulator. With this in mind, we advocate a design approach that uses non-functional properties to drive the top-level architecture. The "keep it simple" rule can be used for individual subsystems, with some assurance that their eventual composition will preserve the required top-level behavior.

5. TWO PROJECTS

This section describes how we have adopted some XP practices in two very different projects: the MULTOS CA and the SPARK Examiner.

5.1 The MULTOS CA

The development of the MULTOS CA is fully described in [7]. This project initially led to our discovery of XP and how many XP techniques we were already using. In particular the CA development made use of the following techniques:

The Planning Game. We never stopped planning and re-planning the development as the system evolved and (invariably) requirements began to change.

Small Releases and Continuous Integration. The system was built as a series of controlled releases, each adding a planned set of functionality. The user-interface was development one cycle "ahead" of the core functionality so that the GUI developed and testing in build N could be used as the test harness for the functions implemented in build N+1. This avoided the need for expensive test harnesses and simulation.

Simple design. This was implemented within the guidance above. We *very* carefully considered security properties and performance before committing to a design.

Testing. The system was continually regression tested using an ever-growing set of tests. Almost all tests were purely requirements-based and were derived from the top-level formal specification of the system. Automation of tests proved very important: we used the Rational VisualTest environment to create scripts and groups of tests that could be run individually or as a group with as little human intervention as possible.

Refactoring. There was some refactoring as the system was built, but it is interesting to note how many requirements changes were accommodated by the design without the need for refactoring. In particular, the system was designed to be extensible for several versions of the MULTOS system—we started the project implementing just a single version, but ended up delivering support for three versions, *without* having to refactor the original design.

Pair programming and Coding standards. While we didn't use the "two engineers at one desk" approach advocated by XP, we did use static analysis and regular reviewing. Where possible, coding standards were enforced by tools, such as the SPARK Examiner for SPARK, and the BoundsChecker tool for C++.

Collective ownership. Within the restrictions of security imposed by the CM system, all development engineers were able to access and work on all the code. Note, though, that security and independence requirements meant that the development and test teams were not permitted write-access to each other's sections of the CM system.

Some XP practices were not used in the development of the CA. These were:

Metaphor. The use of metaphor or "stories" as a requirements engineering approach has some merit, but the CA was constructed using a rigorous requirements engineering process (REVEAL®) and was formally specified using the Z language.

40-hour week. A laudable goal, which was almost attained on average, although there were some weeks where this was exceeded owing to hard deadlines.

On-site customer. Unfortunately, we were not able to implement this practice. In retrospect this would have been useful—a small number of defects were discovered during the customer's acceptable test process that almost certainly *would* have been spotted much earlier by an on-site customer.

5.2 The SPARK Examiner

The SPARK Examiner is a software tool that supports the SPARK language. It implements most of the functions of the front-end of a compiler, but then goes on to implement a number of static analyses such as checking the semantic rules of SPARK, information flow analysis, and verification-condition generation.

Our discovery of XP led to some changes in the way that the Examiner is developed and maintained. This section briefly describes our current development and quality control approach in terms of the core XP practices.

Development. Consider a simple addition to the SPARK language—the addition of modular types, for instance. We start with the SPARK language design itself, writing the rules and any necessary subset restrictions. There may also be a phase of defining the syntax and semantics of any additional annotations that may be required. Development then proceeds incrementally, with code and new test cases being developed at the same time. Both positive tests (i.e. legal SPARK programs that produce some expected output from the Examiner) and negative tests (i.e. illegal programs that deliberately break the language rules) are constructed. New tests are independently reviewed (i.e. by another member of the team) before being added to the CM system.

The SPARK Examiner is, of course, written in SPARK, so static analysis is performed interactively, before code is ever compiled or checked back in to the CM system.

Testing. The SPARK development environment contains a suite several thousand tests, both positive and negative. The source and expected results for each test are stored in our CM system, so a complete history for each test is available. A developer can run the tests in about 15 minutes on a contemporary PC. Alternatively, an "overnight" regression test run is used. This script checks out the latest "wavefront" sources for all the SPARK tools, builds them, and runs both static analysis and tests. An automated "diff" is then generated that highlights any discrepancy between the master copy of the expected results and those just obtained. A summary of the results is emailed to all team members. Any differences have to be reviewed and approved by a team member. If acceptable, then the new results are checked back in to the CM system.

Regression Analysis. Part of the "overnight" regression testing actually includes a complete static analysis of the Examiner source. Once again, the expected results of this are stored in the CM system, just like any other test result. This is often the first place we look if a difference in behavior is observed – it's certainly easier to review a static analysis report than it is to look at possibly hundreds of other test results. This is rare, though, since all developers are encouraged to run the self-analysis interactively during development, rather than waiting for the subsequent "overnight" run.[4] The SPARK language and the Examiner are carefully designed so that this style of *constructive* static analysis is tractable for large programs.

Supporting Regression Test. During the development of this analysis and test strategy, we found one particular problem that caused some difficulty. The Examiner's analysis report normally includes an error number and a source-file line number for each error or warning reported. It is common that, during development, we add or remove code that preserved all existing errors and warnings, but simply changes the line numbers. Secondly, new code may raise a new (expected) warning, but that will then change the numbering of all subsequent warnings, so these show up as wholly spurious "diffs" in the regression analysis. To solve this problem, we implemented a new switch in the Examiner called "plain output mode." This instructs the Examiner to suppress all line numbers, error numbers, and time-stamps in its output, which significantly reduces the number of "spurious diffs" in the regression analysis. This switch was initially only used within the SPARK development team, but proved so useful that it is now available to all users.[5]

Refactoring. Over the years, several significant refactorings of the Examiner have been achieved. Indeed—many of these were performed before the term "refactoring" had even been coined and become commonly used. For example, the main Examiner data-structures used to be library-level abstract state machines. Some years ago, these were re-designed to be instances of simple abstract data types, with the various analysis procedures declaring their own "heaps" as local variables. These "heaps" (actually arrays) are used to build the various data structures needed by Examiner, but are then deallocated "for free" when the procedure in question returns, just like any other local variable.[6]

Another example concerns the static semantic processing routines for expressions. There are two kinds of expressions in SPAR—"expression" and "annotation expression". The former is a pure subset of Ada's normal form of expression, while the latter has an extended syntax and semantics covering quantified terms, logical implication and so on. The procedures for semantic analysis of these two forms initially grew entirely separately, and remained so for many years. Eventually, though, it became apparent that there was so much in common between the processing of the two forms that a refactoring was called for. The common processing routines (e.g. the treatment of "primary" expressions, which is common to both) were "lifted" into the enclosing package and parameterized as necessary. This major simplification produced a significant *reduction* in the amount of code, with absolutely no visible difference in external behavior. Secondly, static analysis was used during this process to ensure that each "lifted" procedure preserved the contracts of the original form.

Aside—Porting the Examiner. SPARK is designed to have an "unambiguous semantics"—it is free from all erroneous and implementation-dependent behavior. One pleasant side-effect(sic) of this is that SPARK is a remarkable *portable* language**.** A testament to this is the ease with which we are able to port the Examiner to new host platforms. The same self-analysis and regression testing approach is used on all currently supported platforms (IA32/Windows, SPARC/Solaris, and VAX/VMS) with very nearly identical test results of all three. We have also been able to construct (as yet unsupported) demonstration versions of the Examiner on other platforms such as IA32/Linux, Mac OS X, and Alpha/OpenVMS – these ports have been constructed and have passed regression testing with little or no effort.

6. CONCLUSIONS

It seems that XP and the development of High-Integrity systems are not such strange bedfellows after all. In particular:

- Many XP practices (such as regression testing and coding standards) are well known and widely used in the world of high-integrity software, where such rigorous practices are the norm, rather than the exception.
- Modern hardware, even with modest commodity PCs, offers the kind of processing power that enables extraordinary depths of analysis and volumes of tests to be run in "coffee break" timescales.
- Static Verification offers an extra level of defense in showing that a refactoring has preserved program behavior. We have employed this technique for many years on a long-lived and highly portable toolset development.
- Static Verification offers an extra weapon in our verification arsenal that we can use to *complement* regression testing. A language like SPARK allows deep analysis of subtle program properties, and can be employed both during program construction and as a pre-curser to regression testing.

The catch, of course, is that the depth and efficiency of static verification critically depends on the language under analysis. SPARK sets a particular high watermark in this domain, but remains (at present) largely unknown to the XP community. This may explain why the benefits of static verification have not been reported more widely in the XP literature to date. There are signs of life, though—Eiffel has brought design-by-contract to a wider audience, and its benefits have been recognized by efforts such as the Java Modeling Language (JML) which does aim to address both static and dynamic verification. Static verification may yet become a sexy and fashionable practice!

[4] One SPARK developer (who shall remain nameless…) is now so dependent on the self-analysis that he often forgets to compile his code at all, and then complains that his tests aren't working as expected, much to the amusement of the rest of the team!

[5] If only they read the manual…

[6] Remember that SPARK has no access types, so we don't use allocators to build linked data structures.

7. REFERENCES

[1] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003 ISBN 0-321-13616-0.

[2] Kent Beck. *Extreme Programming Explained.* Addison Wesley. ISBN 0-201-61641-6.

[3] Butler, Ricky W.; and Finelli, George B.: *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*. IEEE Transactions on Software Engineering, vol. 19, no. 1, Jan. 1993, pp 3-12.

[4] Littlewood, Bev; and Strigini, Lorenzo: *Validation of Ultrahigh Dependability for Software-Based Systems*. CACM 36(11): 69-80 (1993)

[5] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2000. ISBN 0-201-48567-2.

[6] Reiner, Rob: *This is Spinal Tap*, 1984. (The bit with the amplifier…)

[7] Hall, Anthony and Chapman, Roderick: *Correctness by Construction: Building a Commercial Secure System.* IEEE Software Jan/Feb 2002. Also on www.sparkada.com