Discussion Paper on Coding Conventions and Logic Verification

1 Introduction

This discussion paper addresses two topics related to the production and review of voting system source code.

"Coding conventions" are requirements on the specific programming practices used in voting system source code. Together with quality assurance and configuration management, their purpose is to enhance the workmanship of voting system software (and in some cases firmware).

"Logic verification" is a source code analysis technique that can provide greater assurance of the correctness of voting system logic than is obtainable through operational testing.

The directions that the current draft takes regarding coding conventions and logic verification **have not changed** since they received general TGDC approval at the September 2005 TGDC meeting. We provide this review in recognition of the importance of these topics and their potential to generate controversy, and as a service to new TGDC members who may be unaware of the directions previously presented.

2 Terminology

In [5] Volume I, Section 5.2, a subsection titled simply Coding Conventions is placed at the same level as other subsections that define specific coding conventions—practices to enhance software integrity, code structuring requirements, requirements on control constructs, naming and commenting conventions—which implies that coding conventions *per se* are something distinct from any of these. This implication is confusing. In this discussion paper and in the current draft, all of these subtopics are collectively referred to as "coding conventions."

3 Identified problems

Volume I, Section 5.2 and Volume II, Section 5.4 of [5] define coding conventions and a source code review to be conducted by test labs. [5] Volume I, Section 5.2.6 specifies that vendors are permitted to use current best practices in lieu of the coding conventions defined in the VVSG. However, the coding conventions in the VVSG are not aligned with the state of the practice, and if followed, could do more harm than good.

The misalignments are (1) that the conventions, some of which were carried over from [3], are out of date, and (2) that the conventions, limited by their language-independence, are variously incomplete and/or inappropriate in the context of different programming languages with their different idioms and practices.

While they address integrity and maintainability to an extent, the coding conventions are primarily a means to the end of facilitating test lab evaluation of the code's correctness to a level of assurance beyond that provided by operational testing. That evaluation is underspecified in [5], yielding a cart-before-horse situation in which adherence to the coding conventions could be verified much more rigorously than the correctness of the software.

In teleconferences between NIST and the Election Technology Council of the Information Technology Association of America, it was asserted by at least one vendor that compliance with the optional coding conventions defined in the VVSG has been a *de facto* mandate. If this is so, then it is all the more important that the inappropriate guidelines be remedied.

4 Authorization to correct problems

TGDC Resolution #29-05 reads:

> Volume I, Section 4.2 and Volume II, Section 5.4 of the 2002 VSS defines coding standards, as well as a source code review to be conducted by Independent Testing Authorities (ITAs) to enforce those coding standards. These coding standards are a means to an end, the end being an ITA evaluation of the code's correctness to a high level of assurance. The TGDC requests that NIST:
>
> 1. Recommend standards to be used in evaluating the correctness of voting system logic,

including but not limited to software implementations, and

2. Evaluate the 2002 VSS software coding standards with respect to their applicability to the recommended standards, and either revise them, delete them, or recommend new software coding standards, as appropriate.

5 Changes to coding conventions

Best practices for coding evolve on a different and generally more rapid time scale than standards for voting systems. Therefore, if we are to enable voting system vendors to use best practices on a continuing basis, we must resist the urge to transform current best practices into permanent mandates.

There are, however, certain "worst practices" that we would be remiss not to prohibit. Additionally, there is a small core of relatively language-independent coding practices that have a disproportionate impact on voting system integrity. Recognizing these special cases, we have taken a compromise approach that could be described using the famous "80/20" analogy: The 20 % of coding conventions that account for 80 % of the impact on voting system integrity have been retained and enhanced, while the 80 % of coding conventions that account for only 20 % of the impact on voting system integrity have been replaced with a general requirement to use current best practices.

5.1 Conventions with high impact on integrity

Most of the retained conventions are common sense for the engineering of high-integrity software and are applicable regardless of programming language. For example: self-modifying code is prohibited; input data shall be checked for validity before the system commits to using them; buffer overflows and similar types of errors shall be prevented; exceptions shall be handled. The specific requirements can be found in Volume III, Section 5.4.1 of the current draft.

The requirement most likely to stir controversy is the requirement to use block-structured exception handling, which in turn requires the use of a programming language that supports it. This rules out the C language, which remains in wide use, and forces a migration to a descendant language, namely C++, C#[1] or Java. Similarly, older versions of Visual Basic that lacked block-structured exception handling are superseded by Visual Basic .NET.

This requirement follows naturally from existing requirements in [4] and [5], specifically:

> [4] I.2.2.5.2.2.g / [5] I.2.1.5.1.b.vii. Nested error conditions shall be corrected in a controlled sequence such that system status shall be restored to the initial state existing before the first error occurred.
>
> [4] I.4.2.3.e / [5] I.5.2.3.e. Each module shall have a single entry point, and a single exit point, for normal process flow. ... The exception for the exit point is where a problem is so severe that execution cannot be resumed. In this case, the design must explicitly protect all recorded votes and audit log information and must implement formal exception handlers provided by the language.

It appears to be the intent of these requirements that the voting system software should (A) exhibit behaviors that are representative of block-structured exception handling, and (B) accomplish these using "formal exception handlers provided by the language."

[4] and [5] allowed the use of languages that did not support any semblance of formal exceptions. However, programming languages supporting block-structured exceptions have been widely available and widely used for some years now, and they contain other refinements and evolutionary advances, relative to their exceptionless ancestors, that contribute to enhanced software integrity, maintainability, and understandability. To require the use of block-structured exceptions now is in the same spirit of progress as the 1990 VSS [3] requirement for block-structured control constructs. The alternative is to accept less readable source code and a higher likelihood of masked failures.

It is possible to implement exception handling without use of formal exception handlers, just as it is possible to construct robust programs entirely in assembly language or using only GoTos for control flow. But these less structured techniques obfuscate the code and make logic verification more difficult. "One of the major difficulties of conventional defensive programming is that the fault tolerance actions are inseparably bound in with the normal processing which the design is to provide. This can significantly increase design complexity and, consequently, can compromise the reliability and maintainability of the software." [2]

Though potentially painful, the migration from languages not supporting block-structured exceptions is facilitated by closely related languages that evolved from one another: C and C++, C# or Java, Visual Basic and Visual Basic .NET. We believe that the costs of requiring this migration are exceeded by the benefits.

5.2 Conventions with low impact on integrity

Low impact coding conventions that were incorporated in [4] and [5] have been purged from the current draft, leaving the adoption of current best practices as the only option.

To write a precise and defensible definition of what would constitute current best practices at any given time in the future is problematic. The formulation used in [5] Volume I, Section 5.2.6, "published, reviewed, and industry-accepted coding conventions," is uncomfortably vague. The current draft has improved on this somewhat by adding normative definitions of what is required for coding conventions to be considered published and credible, but the resulting requirements are still uncomfortably vague.

At the September 2005 TGDC meeting, Ms. Quesenbery suggested the alternate approach of establishing an authority to periodically review current best practices and publish a list of those found to be acceptable for use in voting systems. Such an authority would need to be established by the EAC and would need to perform its review no less often than once a year. This would eliminate ambiguity and provide a safe path for vendors and test labs to follow. The TGDC may recommend that such an authority be established, but there is no action that we can take within the scope of the draft Guidelines.

5.3 Applicable scope

The voting system standards have always acknowledged that there are places where coding conventions cannot be applied. For example, [5] Volume I, Section 5.2.1 states, "The requirement for the use of high-level language for logical operations does not preclude the use of assembly language for hardware-related segments, such as device controllers and handler programs."

The circumstances in which coding conventions are not applicable are made more precise in the current draft through the introduction of new terms "application logic," "border logic," "configuration data," "COTS," and "third-party logic." Please see a companion document titled *COTS Discussion Paper* for details on these new terms and what they mean for the applicability of coding conventions.

6 Logic verification

6.1 Description

Logic verification is specified in Volume V, Section 4.7 of the current draft as a conformity assessment activity to be performed by the test lab using input from the vendor's quality assurance process.

Traditionally, testing methods have been divided into black-box and white-box test design. Neither method has universal applicability; they are useful in the testing of different items.

Black-box testing is usually described as focusing on testing functional requirements, these requirements being defined in an explicit specification. It treats the item being tested as a "black box," with no examination being made of the internal structure or workings of the item. Rather, the nature of black-box testing is to develop and utilize detailed scenarios, or test cases. These test cases include specific sets of input to be applied to the item being tested. The output produced by the given input is then compared to a previously defined set of expected results.

White-box testing (sometimes called clear-box or glass-box testing to suggest a more accurate metaphor) allows one to peek inside the "box," and focuses specifically on using knowledge of the internals of the item being tested to guide the testing procedure and the selection of test data. White-box testing can discover extra non-specified functions that black-box testing wouldn't know to look for and can exercise data paths that would not have been exercised by a fixed test suite. Such extras can only be discovered by inspecting the internals.

Complementary to any kind of operational testing is logic verification, in which it is shown that the logic of the system satisfies certain assertions. When it is impractical to test every case in which a failure might occur, logic verification can be used to show the correctness of the logic generally. However, verification is not a substitute for testing because there can be faults in a proof just as surely as there can be faults in a system. Used together, testing and verification can provide a high level of assurance that a system's logic is correct.

[1] provides the following description of logic verification, therein known as "program proving:"

> Assertions are made at various locations in the program which are used as pre- and post-conditions to various paths through the program. The proof consists of two parts. The first involves showing that the program transfers the pre-conditions into the post-conditions

according to a set of logical rules defining the semantics of the programming language, provided that the program actually terminates (i.e. reaches its proper conclusion). The second part is to demonstrate that the program does indeed terminate (e.g. does not go into an infinite loop). Both parts may need inductive arguments.

The inspection specified in the current draft does not assume that the programming language has formally specified semantics. Consequently, a formal proof at any level cannot be mandated. Instead, a combination of informal arguments and limitations on complexity seeks to make the correctness of software units at the lowest level intuitively obvious and to enable the verification of higher level units using the pre- and postconditions of invoked units as premises. The resulting inspection is not as rigorous as a formal proof, but still provides greater assurance than is provided by operational testing alone.

After reviewing the materials submitted, test labs are entitled to additional proof if the correctness of a software unit is insufficiently verifiable.

6.2 Applicable scope

Because of its high complexity, the scope of logic verification is pragmatically limited to "core logic," defined as the subset of application logic that is responsible for vote recording and tabulation. Software modules that are solely devoted to interacting with the user or formatting reports are not subject to logic verification. However, they are inspected in other portions of the conformity assessment process to establish confidence that they meet requirements for security and workmanship, and they are also subject to operational testing as part of the complete system.

6.3 Controversy

Although the objection has not yet been raised in any subcommittee discussions, a commonly raised objection to logic verification is the observation that, in the general case, it is exceedingly difficult and often impractical to verify any nontrivial property of software. This is not the general case. While the Guidelines try to avoid constraining the design, all voting system designs must preserve the ability to demonstrate that votes will be counted correctly. A voting system that is designed in such a way that it *cannot* be shown to count votes correctly is not certifiable.

The restriction of the scope to core logic and the relaxation of the level of formality normally demanded for logic verification further serve to make this recommendation both feasible and practical to implement, at some cost to the level of assurance obtained.

7 References

[1] F. J. Redmill, Ed., Dependability of Critical Computer Systems 1, Elsevier Applied Science, London and New York, 1988.

[2] M. R. Moulding, "Designing for high integrity: the software fault tolerance approach," Section

3.4. In C. T. Sennett, ed., High-Integrity Software, Plenum Press, New York and London, 1989.

[3] Performance and Test Standards for Punchcard, Marksense, and Direct Recording Electronic Voting Systems, January 1990 edition with April 1990 revisions, in Voting System Standards, U.S. Government Printing Office, 1990. Available at http://josephhall.org/fec_vss_1990_pdf/1990_VSS.pdf.

[4] 2002 Voting Systems Standards, available from http://www.eac.gov/election_resources/vss.html.

[5] 2005 Voluntary Voting System Guidelines, Version 1.0, 2006-03-06, available from http://www.eac.gov/vvsg_intro.htm.

Notes

[1] Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.