

Comments on 2009 VVSG 1.1 Draft

Eric Rescorla
RTFM, Inc.
ekr@rtfm.com

September 28, 2009

§ 5.2.2.

§ 5.2.2 requires that systems be written in programming languages which support block-structured exception handling:

The above requirement may be satisfied by using COTS extension packages to add missing control constructs to languages that could not otherwise conform. For example, C99[2] does not support block-structured exception handling, but the construct can be retrofitted using (e.g.) `cexcept`[3] or another COTS package.

The use of non-COTS extension packages or manufacturer-specific code for this purpose is not acceptable, as it would place an unreasonable burden on the VSTL to verify the soundness of an unproven extension (effectively a new programming language). The package must have a proven track record of performance supporting the assertion that it would be stable and suitable for use in voting systems, just as the compiler or interpreter for the base programming language must.

One could interpret this requirement as simply being that the language must support this functionality, not that it be used, in which case the requirement is unobjectionable. However, § 5.2.5 makes clear that programmers are expected to actually use exception constructs, which is significantly more problematic.

The first issue is that an exception-oriented programming style is significantly different than an error code-oriented programming style, so complying with the spirit of this requirement implies a very substantial rewrite of any existing system which uses error codes. However, experience with previous VVSG programming style requirements suggests that vendors will do the minimum required to comply with the letter of the VVSG. Because the two styles are similar enough that the conversion process can be done semi-mechanically, the likely result will be a program which superficially works but which now has subtle bugs which were introduced during the conversion process.

One class of bugs that deserves particular attention is the proper cleanup of objects before function exit. When a function creates a set of objects and then encounters an error, it is important to clean up those objects before returning with the error. Failure to do so can leak memory and, more importantly, lead to improper finalization of objects which may be connected to non-memory resources such as network connections, hardware, etc. In languages such as Java, this is handled by garbage collection, but C is not garbage collected. Thus, when an exception is thrown from a function deep in the call stack and caught in a higher function, it bypasses all explicit cleanup routines in intermediate functions, which can lead to serious errors. Handling this situation correctly requires extreme care comparable to that required with conventional error handling. Writing correct code under these circumstances is challenging under the best of conditions, but is likely to be impractical under conditions where programmers are required to convert existing error code-based software.

While it might be the case that a completely new system written along exception-oriented lines would be superior, I am aware of no evidence that a retrofitted system would be superior and there is a substantial risk that it will be worse.

The second issue is that because C has no native exception handling, systems written in C will need to use a COTS package. Unfortunately, because exception handling is not a native feature of C, any attempt to retrofit it involves tradeoffs. As an example, the `cexcept` package cited above, does not support conditional exception handling. In C++ exceptions, it is possible to have a `catch` statement which only catches some exceptions, e.g.,

```

try {

}
catch (memory_exception) {
    ;
}
catch (data_exception) {
    ;
}

// Other exceptions get passed through

```

But in `cexcept`, a `Catch` statement catches exceptions of all types and you need to use an explicit conditional in order to discover which exception was thrown. But this creates much the same opportunity to ignore/mishandle unexpected exceptions that error codes do.

Another problem with `cexcept` is that it is very brittle whenever exception handling is intermixed with conventional error handling. Any function which jumps out of a `try/catch` block can result in "undefined" behavior (i.e., the implementation can do anything whatsoever). This, of course, is an easy mistake to make when converting from return codes to exceptions.

`cexcept` is not, of course, the only C exception package. For instance, Doug Jones has developed a different exception package, which makes different tradeoffs (though the above intermixed exception/return problem seems to exist here too).

Third, the use of the term "COTS" to cover these packages seems to require a fairly loose definition of COTS. While it is true that there are a number of exception packages available for free download, it is not clear to what extent they are in wide use by programmers. In my experience as a professional programmer, I have yet to work on a system written in C which used one of these packages. As the stated purpose of the COTS requirement is to ensure that the packages have seen enough deployment that we can have confidence in their quality, it seems questionable whether any of the available packages meet this standard.

§ 5.2.6

Header comments and other commenting standards should be specified by the selected coding standard in a manner consistent with the idiom of the programming language chosen. If the coding standard specifies a coding style and commenting standard that make header comments redundant, then they may be omitted. Otherwise, in the event that the coding standard fails to specify the content of header comments, application logic modules should include header comments that provide at least the following information for each callable unit (function, method, operation, subroutine, procedure, etc.):

- a. The purpose of the unit and how it works (if not obvious);
- b. A description of input parameters, outputs and return values, exceptions thrown, and side-effects;
- c. Any protocols that must be observed (e.g., unit calling sequences);
- d. File references by name and method of access (read, write, modify, append, etc.);
- e. Global variables used (if applicable);
- f. Audit event generation;
- g. Date of creation; and
- h. Change log (revision record). Change logs need not cover the nascent period, but they must go back as far as the first baseline or release that is submitted for testing, and should go back as far as the first baseline or release that is deemed reasonably coherent.

As I read this requirement, it applies to every *subroutine*, not to the file as a whole. This seems to be strongly at variance with any programming practice I've ever seen; it's certainly common to have block comments which describe the behavior of a subroutine (though there is controversy about how useful this is) but creation dates and change logs on a per-subroutine rather than per-file basis strike me as more likely to get in the way of code comprehension than to assist it—as well as likely being fairly wrong because they are probably manually maintained.

In my opinion, a far better procedure would be to require that the software development process use some form of revision control (CVS, Subversion, etc.) and that the revision history be made available to the test labs. Revision control is a standard software engineering practice and would allow automatic extraction of the revision history. This seems like a useful requirement regardless of the disposition of this comment requirement.

§ 5.2.7

- ii. Application logic shall be free of race conditions, deadlocks, livelocks, and resource starvation.
- ...
- e. Application logic and border logic shall contain no inaccessible code (dead code) other than defensive code (including exception handlers) that is provided to defend against the occurrence of failures and "can't happen" conditions.

These both seem like good goals, but it's not clear to me that they are operationalizable: programmers don't set out to write code with race conditions, resource starvation, dead code, etc., and there's no practical way to demonstrate that programs don't have these defects. I agree that if these defects are detected that might be a reason to refuse to certify (I say "might" because there might be situations where, for instance, resource starvation was acceptable if they were extremely difficult to trigger).

§ 5.2.8(b)(i)

- i. If the application logic uses arrays, vectors, or any analogous data structures and the programming language does not provide automatic run-time range checking of the indices, the indices shall be range-checked on every access. Range checking code should not be duplicated before each access. Clean implementation approaches include: (1) consistently using dedicated accessors (functions, methods, operations, subroutines, procedures, etc.) that range-check the indices; (2) defining and consistently using a new data type or class that encapsulates the range-checking logic; (3) declaring the array using a template that causes all accessors to be range-checked; or (4) declaring the array index to be a data type whose enforced range is matched to the size of the array. Range-enforced data types or classes may be provided by the programming environment or they may be defined in application logic. If acceptable values of the index do not form a contiguous range, a map structure may be more appropriate than a vector.

This requirement strikes me as fairly onerous. Consider the following (natural) C stanza:

```
typedef struct {
    int count;
    int *values;
} array_of_ints;

void compute_statistics(array_of_ints *arr)
{
    int sum=0;
    int sumsq=0;
    int i;

    for(i=0; i<arr->count; i++){
        sum += arr->values[i];
        sumsq += arr->values[i];
        printf("Value %d = %d\n", i, arr->values[i]);
    }

    printf("Mean=%d  Sum of squares %d\n", sum, sumsq);
}
```

Now, if we are to take this requirement literally, then we need to range-check on every single access, even though by definition `i` can never take on any value other than `0..arr->count-1`. We could "correct" this code by the following indirection:

```
typedef struct {
    int count;
    int *values;
} array_of_ints;

int array_of_ints_get_value_by_index(array_of_ints *arr, index i)
{
    if (i < 0)
        throw OutOfRangeException;

    if(i >= arr->count)
        throw OutOfRangeException;

    return arr->values[i];
}
```

Our function then becomes:

```
void compute_statistics(array_of_ints *arr)
{
    int sum=0;
    int sumsq=0;
    int i;

    for(i=0; i<arr->count; i++){
        sum += array_of_ints_get_value_by_index(arr,i);
        sumsq += array_of_ints_get_value_by_index(arr,i);
        printf("Value %d = %d\n",i,array_of_ints_get_value_by_index(arr,i));
    }

    printf("Mean=%d Sum of squares %d\n", sum, sumsq);
}
```

This is not an improvement in the situation either in terms of correctness or in terms of readability, but it directly follows from this requirement. The issue here is not that range checking is not important, but blindly requiring checking at every access does not help. What needs to happen is that the inputs to each routine need to fulfill certain invariants that the routine can rely on and that the routine's behavior is then safe given those invariants. In the case of this program, the invariant is that the `array_of_ints` type passed into the first version of `compute_statistics` has the right length value. If that's true, then that version is safe. If it's untrue, then it's very hard to salvage the program. This is harder to formalize, but is actually consistent with standard C programming idioms, whereas the above requirement is not.

This is not to say that it's not useful to have some mechanism for checking that pointers are valid and failing if they are not. Some C compilers do have features for pointer checking, but that's not really something that can be done in the application logic.

§ 5.2.8(c)(i)

- i. If application logic uses pointers or a similar mechanism for specifying absolute memory locations, the application logic should validate pointers or addresses before they are used. Improper overwriting should be prevented in general as required by Requirements 5.2.7.b and c. Nevertheless, even if read-only

memory would prevent the overwrite from succeeding, an attempted overwrite indicates a logic fault that must be corrected. Pointer use that is fully encapsulated within a standard platform library is treated as COTS software.

I don't really understand this requirement. There is no portable way to validate a pointer in C or C++, and of course most other common languages (java, python, perl...) don't have pointers. You can check that it is nonzero, but there are many other ways for a pointer to be invalid. It certainly seems useful for application logic to ensure that pointers are always valid (e.g., by zeroing them on `free()`, although this is actually harder than it sounds), but that's not the same as validating them.

§ 5.2.8(g)

g. Error checks detailed in Requirements b and c shall remain active in production code. These errors are incompatible with voting integrity, so masking them is unacceptable. Manufacturers should not implement error checks using the C/C++ `assert()` macro. It is often disabled, sometimes automatically, when software is compiled in production mode. Furthermore, it does not appropriately throw an exception, but instead aborts the program.

Without necessarily defending `assert()`, the complaint that `assert()` aborts the program seems confused in this context. The errors in (b) and (c) include "(2) stack overflow errors; (3) CPU-level exceptions such as address and bus errors, dividing by zero, and the like;" and "(1) pointer variable errors; (2) dynamic memory allocation and management errors." These are usually not recoverable errors and often indicate large-scale memory corruption. The only safe response is to terminate program execution.