U.S. ELECTION ASSISTANCE COMMISSION
633 3rd St. NW, Suite 200
Washington, DC 20001

EAC Decision on Request for Interpretation

2023-03 2.1-A.5 Block-Structured Exception Handling

Sections of Standards or Guidelines:

2.1-A – Acceptable programming languages

Application logic must be produced in a high-level programming language that has all of the following control constructs:

5. block-structured exception handling (for example, try/throw/catch).

Discussion (excerpt)

…Previous versions of VVSG required voting systems to handle such errors by some means, preferably using programming language exceptions ([VVSG2005] I.5.2.3.e), but there was no unambiguous requirement for the programming language to support exception handling. These guidelines require programming language exceptions because without them, the programmer must check for every possible error condition in every possible location, which both obfuscates the application logic and creates a high likelihood that some or many possible errors will not be checked. Additionally, these guidelines require block-structured exception handling because, like all unstructured programming, unstructured exception handling obfuscates logic and makes its verification by the test lab 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." [Moulding89] …

Date:

September 29, 2023

Question(s):

Request for clarification for using the Rust programing language which does not explicitly use block-structured error handling. Would Rust qualify as using border-logic, meeting the VVSG 2.0 code requirement 2.1-A?

Discussion:

Per the discussion for requirement 2.1-A.5 in VVSG 2.0, the intent is to require exception handling in a structured manner. Without the use of exceptions, programmers would be required to check for every possible error, which is an infeasible task with the potential to miss some or many errors. Additionally, this calls for organization in a structured form. Unstructured programming creates a difficult scenario where the testing lab must track down the logic in

multiple locations within the code to verify exception handling is appropriately addressed. Therefore block-structured exception handling is specifically identified to address these issues.

Block-structured exception handling functions by using a command to **try** a section of code and if something goes wrong for ANY reason, either it will **catch** with another attempt of different sections of code or **throw** a custom error and give the details. These three functions make it possible to address all errors and program for the handling of specific errors without aborting the code. Also, by having the code and its error handling blocked on top of one another makes this logic easy to verify. This is a common capability of most coding languages.

Rust does not explicitly use block-structured exception handling. Instead, it has built in mechanisms that avoid splitting the handling of the application logic. It also groups errors into two major categories: *recoverable* and *unrecoverable.* For a *recoverable* error, the programmer can propagate error states succinctly and clearly with **Result<T, E>**, the **?** operator, **unwrap()**, **expect()**, and related constructs. The programmer does not have to "check for every possible error condition in every possible location." Rust's type checking requires errors to be handled or propagated back to the caller. The goal is to report the problem to the user and retry the operation. This functions similarly to **try** and **catch** without stopping the logic from running. For an *unrecoverable* error, which is a symptom of a bug in the code, the program should be aborted and the **panic!** macro stops all execution, which functions similarly to **throw** logic in other programming languages.

The second concern is that "unstructured exception handling obfuscates logic and makes its verification by the test lab more difficult." Rust does not have unstructured exceptions. All recoverable and unrecoverable error handling is organized in a structured manner like try/catch/throw but not as separate blocks. Exception handling can be in a linear or a nested fashion, but an error cannot cause an arbitrary control transfer to another section of code.

Conclusion:

The discussion of the requirement lays out two major concerns without block-structured handling and that is 1) the programmer must check for every possible error condition in every possible location, and 2) unstructured exception handling obfuscate logic and makes its verification more difficult. Though the Rust programming language does not use block-structure exception handling, it addresses both issues by handling errors in a structured and succinct way. Therefore, the Rust programming language is considered acceptable to satisfy requirement **2.1-A Acceptable programming languages**, and its sub requirements.

Effective Date:

As of the date this document is published.