Presented at 25th Water Reactor Safety Meeting, Bethesda, MD, October 22, 1997

Review Guidelines for Software Written in High Level Programming Language Used in Safety Systems

Myron Hecht, SoHaR Incorporated
and Robert Brill, U.S. Nuclear Regulatory Commission

Abstract: This paper provides an overview of the results of NRC-sponsored research on guidelines for review of software written in high level languages for use in safety systems. These guidelines were developed using a 3-level hierarchical framework consisting of top level, intermediate, and base attributes. The top level attributes of reliability, robustness, traceability, and maintainability were developed in order to define general qualities of software related to safety. Intermediate attributes were then developed to describe the top level attributes in greater detail. At the lowest level are the base attributes which were defined to be sufficiently specific to derive language specific guidelines. These attributes were then used to develop specific guidelines for a total of 9 languages. The resulting guidelines are available in the form of both a NUREG report and HTML files.

Certain programming practices can affect the safety of digital systems, and hence, guidelines which should be followed or avoided can be developed to enhance their dependability. This paper provides an overview of the results of An NRC-sponsored project which identifies such guidelines for safety related software written in the following nine high level languages: Ada, Ada95, C/C++, Pascal, PL/M IEC 1131-3 Ladder Logic, Sequential Function Charts, Structured Text, and Function Block Diagrams. The complete guidelines are described in NUREG CR/6463 Rev. 1 (Hecht, 1997) and in a set of hypertext markup language (HTML) files . The first section describes how the guidelines were developed, the second discusses the generic attributes, and the third discusses language-specific issues for selected languages covered by the guidelines.

1 Methodology

In order to develop a technical basis for the guidelines, attributes of software safety were compiled and classified from a review of the large body of literature in software safety. An initial framework was established and iteratively refined resulting in the 3-level hierarchy described shown in Figure 1. The two highest levels called *top level* and *intermediate level* were used to define broader and narrower categories of attributes. The lowest level, called the *base level*, contains qualities or characteristics of safe programming practices that can then be used to develop language specific guidelines.

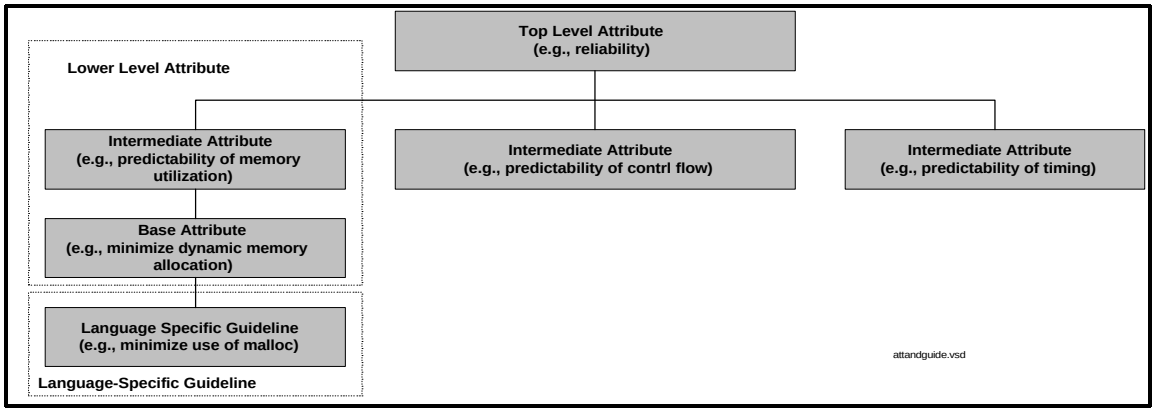

Figure 1 Conceptual Framework for Attributes and Guidelines

These attributes, which are called *generic attributes*, were used by individuals with extensive programming backgrounds in one or more of the 9 languages covered by this project to develop language specific guidelines. The generic attributes and language specific guidelines were then submitted to an independent set of SMEs who served as reviewers. These reviewers provided an initial round of comments, after which the guidelines were revised. The guidelines were then verified by a resubmission to the reviewers for a final round of evaluations. The classification was validated by comparing the attributes with the causes and descriptions of failures in two major air traffic control projects (the Federal Aviation Administration Advanced Automation System and Voice Control Switching System) as well as incident reports from the Eagle 21 reactor protection system upgrades at the Tennessee Valley Authority (TVA) Sequoyah Nuclear Plant. For the C and Ada languages, a total of 150 specific failure reports were associated with specific guidelines. Additional validation came from other published large scale studies of software failures.

2 Generic Attributes

This section describes the generic attributes shown in figure 1. A complete description can be found in Chapter 2 of NUREG/CR 6463 Rev. 1. The top level generic attributes, which define a general quality of software related to safety, are

- *Reliability*. The predictable and consistent performance of the software under conditions specified in the design basis. This top level attribute is important to safety because it decreases the likelihood that faults causing unsuccessful operation will be introduced into the source code during implementation.

- *Robustness*. Robustness is the capability of the safety system software to operate in an acceptable manner under abnormal conditions or events. This top level attribute is important to safety because it enhances the capability of the software to handle exception conditions, recover from internal failures, and prevent propagation of errors arising from unusual circumstances.

- *Traceability*. Traceability relates to the feasibility of reviewing and identifying the source code and library component origin and development processes, i.e., that the delivered code can be shown to be the product of a disciplined implementation process. Traceability also includes being able to associate source code with higher level design documents. This top level attribute is important to safety because it facilitates verification and validation, and other aspects of software quality assurance.

- *Maintainability*. The means by which the source code reduces the likelihood that faults will be introduced during changes made after delivery. This top level attribute is important to safety because it decreases the likelihood of unsuccessful operation resulting from faults during adaptive, corrective, or perfective software maintenance.

2.1 Reliability

In the software context, reliability is either (1) the probability of successful execution over a defined interval of time and under defined conditions, or (2) the probability of successful operation upon demand (IEEE, 1977). That the software executes to completion is a result of its proper behavior with respect to system memory and program logic. That the software produces timely output is a function of the programmer's understanding of the language constructs and run-time environment characteristics. Thus, the intermediate attributes for reliability are:

- *Predictability of memory utilization.* There is a high likelihood that the software will not cause the processor to access unintended or unallowed memory locations.

- *Predictability of control flow*. There is a high probability that the processor will execute instructions in sequences intended by the programmer.

- *Predictability of timing.* There is a high probability that the software executing within the defined run-time environment will meet its response time and capacity constraints.

- *Predictability of mathematical or logical result.* There is a high probability that the software executing within the defined run-time environment will yield the programmer-intended mathematical or logical result.

Figure 2 shows the reliability hierarchy including base attributes.

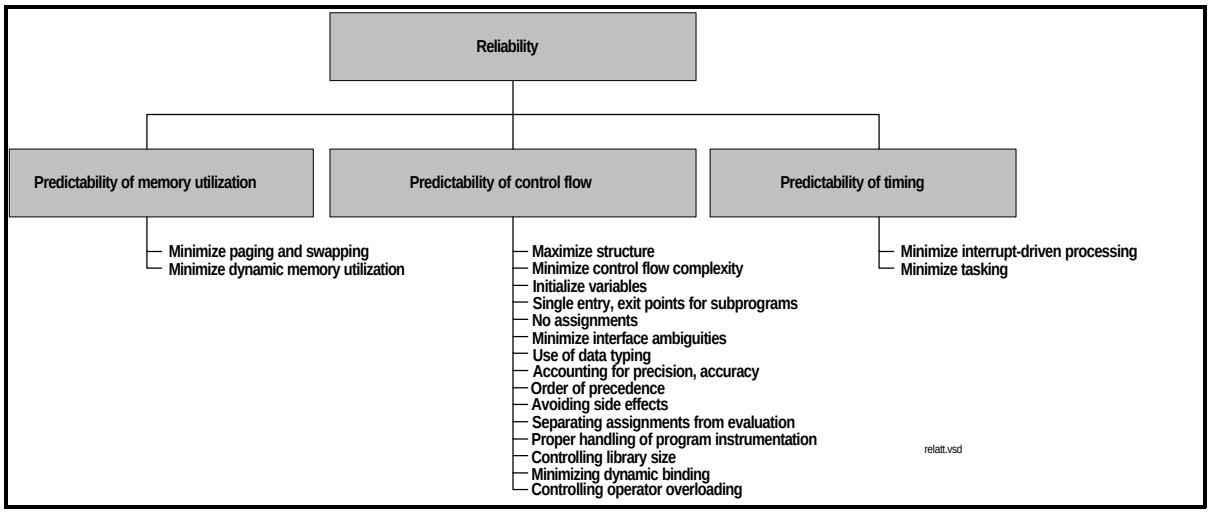

Figure 2 Reliability Attributes

2.2 Robustness

Robustness refers to the capability of the software to continue execution during off-nominal or other unanticipated conditions. A synonym for robustness is survivability (Bowen, 1985; Wigle, 1985). Robustness is an important attribute for a safety system because unanticipated events can happen during an accident or excursion, and the capability of the software to continue monitoring and controlling a system in such circumstances is vital. Figure 3 shows the intermediate and base attributes for robustness.

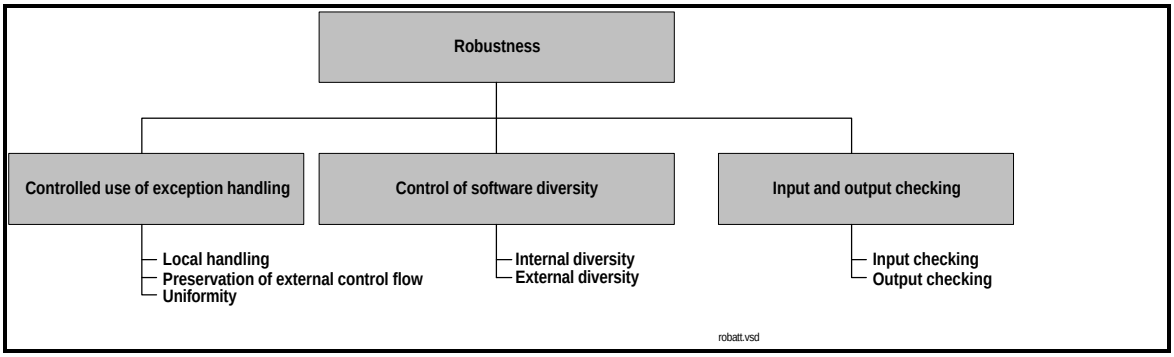

Figure 3. Robustness Attributes

2.3 Traceability

Traceability refers to attributes of safety software which support verification of correctness and completeness compared with the software design. The base attributes for traceability are:

- Readability (also an attribute of maintainability)
- Controlling use of built-in functions
- Controlling use of compiled libraries.

2.4 Maintainability

Software maintainability reduces the likelihood that errors will be introduced while making changes. The intermediate attributes related to maintainability that affect safety include:

- *Readability*: those attributes of the software that facilitate the understanding of the software by project personnel

- *Data abstraction:* the extent to which the code is partitioned and modularized so that the collateral impact and probability of unintended side effects due to software changes are minimized

- *Functional cohesiveness:* the appropriate allocation of design level functions to software elements in the code (one procedure; one function)

- *Portability:* the major safety impact of which is the avoidance of non-standard functions of a language.

- *Malleability:* the extent to which areas of potential change are isolated from the rest of the code

Figure 4 shows these intermediate level and associated base attributes.

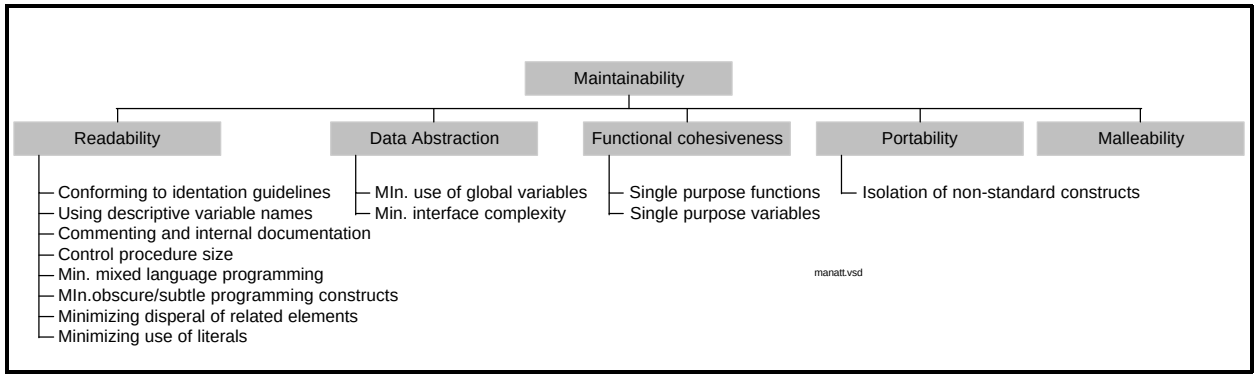

Figure 4. Maintainability Attributes

3 Examples of Language Specific Guidelines

This section provides examples of language-specific guidelines to acquaint readers with what can be found in the full version of NUREG/CR 6463 Rev. 1. The first subsection discusses guidelines on pointer initialization in C; the second discusses several issues on timing, health monitoring, and fault handling in IEC 1131-3 programming languages used in programmable logic controllers (PLCs).

3.1 C and C++ Guidelines on Pointer Initialization

The C/C++[1] language-specific guideline for pointer initialization was derived from the top level attribute of reliability, the intermediate attribute of the predictability of control flow, and the base attribute of initialization of variables before use.

All variables and pointers should be initialized before use (Porter, 1993; Kernighan, 1978). C supports initialization through the facility of specifying initial values along with declarations. However, it does not require that all objects[2] be initialized (Eckel, 1995). In some cases, the initialization of an object entails not only assigning a specific bit-pattern value to the object location, but also taking special actions to facilitate smooth initialization of the object's life (e.g., allocating corresponding resources to the objects). In C++ it is possible to consider any correlated data set as an object and provide facilities for constructing an instance of the data set and destroying the current instance of the data set in a systematic way. The following are specific guidelines.

- *Do not use pointers to automatic variables outside of their scope*. Pointers to automatic variables should not be used outside of their declared scope. The value stored in a pointer to an automatic variable will contain garbage outside the function scope.

- *Initialize pointers*. Initialization problems can also occur in pointers. In safety systems, all pointer variables in C should be initialized to **`NULL`**, and all pointer variables in C++ language should be initialized to 0 (Plum, 1991). The pointer should then be tested for a valid value before being used. In C and C++, when a pointer is defined, it does not have a memory location associated with it. Using an uninitialized pointer will overwrite an unintended portion of memory. Incorrectly overwriting memory can cause serious problems, including system crashes.

- *Ensure that the indirection operator is present for each pointer declaration*. Each pointer should have an indirect operator (*) when it is declared (Porter, 1993). The following example shows how the C syntax facilitates omitting the indirection operator:

- *Use the ~ operator when initializing to all 1's*. When initializing all bits of an integer type to all 1's, use bitwise **`not 0`**.

[1]Guidelines for C and C++ were developed together because of the close relationship between the two languages. In addition, programs written in C++ are also likely to contain C code as well. For C, the guidelines address the problems in memory allocation and deallocation, pointers, control flow, and software interfaces. For C++, the guidelines address additional issues associated with multiple inheritance, late binding, and large class libraries.

[2]That is, variable, structures, or arrays.

3.2 Guidelines for Fault Handling and Timing in IEC 1131 Languages

This subsection summarizes guidelines on fault handling routines, system health monitoring, and watchdog timers and which are relevant across all the IEC 1131-3 languages[3]. These language-specific guidelines are derived from the generic attribute of robustness and the intermediate attribute of exception handling

3.2.1 Fault Routines

Programs should properly account for PLC behavior at shutdown. Generally, all outputs turn off, but this is not always true. The PLC system is designed so that such a shutdown places the system in a fail-safe condition. Some PLCs have the capability to run a subroutine which the processor automatically executes when it encounters a condition that will cause execution of the main Ladder Logic routine to stop. This subroutine is called a "fault routine". It allows the designer to decide on the appropriate action, including shutting down the system in a safe manner. The IEC 1131-3 Programmable Controller Language Specification does not mandate that a compliant PLC system contain a designated hardware fault routine that will execute upon detection of one of a range of fault conditions, similar in concept to the Fault Routine available in current Allen Bradley PLCs (Allen Bradley, 1991). However, a non-periodic task could be set up to trigger appropriate internal diagnostic information to handle particular types of errors (fall-back modes on I/O failures, warnings to operator interfaces, etc). Usage of such a construct should be considered in the design phases of programming for a system of this kind.

The following specific guidelines apply to fault routines:

- *Completeness*. The fault routine cannot be relied on to detect all instances of program crashes. Additional provisions that may be required by the specific safety requirements of the application for PLC major faults must be specified.

- *Observability*. The fault routine should annunciate and log the condition. The execution of the fault routine should not be masked.

- *Validity checking*. The conditions under which the fault routine is running may have corrupted program memory, data files, or I/O. The fault routine must ensure the validity of its environment before proceeding to execute.

- *Fail safe properties in the absence of the fault routine*. The fault routine cannot be relied upon to operate under every major failure condition. The PLC may be so disabled that this is not possible. Thus, the system design should ensure a safe state in the absence of the successful execution of the fault routine.

3.2.2 System Health Monitoring

PLC systems provide System Health Monitoring information to the application program, usually in the form of designated internal bits and words. Information may include execution times for particular tasks, I/O update information, I/O operating status information, battery state information, and other status information used to

[3]The International Electrotechnical Commission (IEC) standard 1131-3 has defined standards for a total of five languages for Programmable Logic Controllers (PLCs). Four of the languages were covered by the guidelines (the fifth, Instruction Lists, more closely resembles assembly language)

determine the proper operating status of a PLC system. Unfortunately, the nature, extent, and type of this information is not covered by the IEC 1131-3 specification. However, the safety program should make use of such information (as available) as part of its health checking and exception processing.

3.2.3 Watchdog Timers

A note in section 2.7.2 (Tasks) of the IEC 1131-3 specification states that "The manufacturer shall provide information to enable the user to determine that all (task) deadlines will be met in a proposed configuration". However, the specification does not mention any form of checks that these deadlines are met at runtime, since this falls into the error-checking category of constructs not covered by the document. Many PLC and PLC-like systems with 1131-3 compliant multitasking capability implement a watchdog timer for each periodic task. Typically, these systems will ignore a single task overrun and flag that an overrun has occurred, but will shut down the system upon *N* consecutive task overruns, where the value of *N* will vary from system to system. Where available, programs should monitor the task overrun flag, and react appropriately. It is not appropriate for a safety-critical system of this nature to exhibit regular task overruns — this should be looked for specifically during the check-out phases of development.

4 Conclusions

The guidelines in NUREC/CR 6463 Rev. 1 address a gap in the literature on software safety. Much has been written about the importance of understanding the application, disciplined development, design methodologies, and testing. However, less has been written about language-specific programming guidelines (some examples include Hatton, 1995; Plum, 1993; and Saaltink, 1996) , and even less has on a uniform framework that can be applied across multiple heterogeneous languages. The guidelines developed in this study address programming issues specific to safety. They are not intended as general programming style guidelines; excellent sources already exist for this purpose.

The guidelines are available as both a printed document (available for purchase from the Government Printing Office, 202-512-1800) and in hypertext markup language (HTML) formatted files available at www.nrc.gov. As HTML files, they can be particularly valuable because they can be viewed and modified locally, within restricted network domains (such as a development group or organization), or across the entire Internet. For development organizations or groups, they can be customized and enhanced for individual projects based on the experience of the development organization.

References

Allen Bradley, *PLC-5 Programming Software – Programming*, Publication 6200-6.4.7 November 1991.

Bowen, T.P. and G.B. Wigle and J.T. Tsai, "Specification of Software Quality Attributes" Report, 3 Vols. RADC-TR-85-37, available from NTIS, 1985.

Eckel, B., "Exception Handling in C++", *Embedded Systems Programming*, Vol.8, No.1, January, 1995.

Hatton, L. *Safer C*, McGraw-Hill International Series in Software Engineering, Maidenhead, Berkshire, England, 1995

Hecht, H., M. Hecht, S. Graff, *et. al.*, *Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems*, NUREG/CR-6463 Rev. 1, U.S. Nuclear Regulatory Commission, September, 1997

International Electrotechnical Commission (IEC), "Software for Computers in the Safety Systems of Nuclear Power Stations," Standard 880, 1986.

International Electrotechnical Commission (IEC), *Programmable Controllers Programming Languages*, IEC Standard 1131, Part 3, 1993. (Available in the U.S. from the American National Standards Institute, New York.)

Kernighan, B.J and P. J. Plauger, *The Elements of Programming Style*, *Second Edition*, McGraw-Hill, New York, 1978.

Porter, A., *The Best C/C++ Tips Ever*. Osborne McGraw-Hill, New York, 1993.

Saaltink, M. And Mitchell, S., Ada95 Trustworthiness Study: Guidance on the Use of Ada95 in the Development of High Integrity Systems Version 1.0, (Canada) Department of National Defence contract W2207-5-RC02/01-SV, Document No. TR-96-5499-04, September, 1996.

Spuler, D.A., C++ and C Debugging, Testing, and Reliability, Prentice Hall, Englewood Cliffs, New Jersey, 1994.