RPSB-99-12
(REV 0)

OFFICE OF NUCLEAR REGULATORY RESEARCH

REACTOR PLANT SYSTEMS
BRANCH

SOFTWARE QUALITY ASSURANCE PROCEDURES FOR NRC THERMAL HYDRAULIC CODES

Frank Odar
January 1999

UNITED STATES NUCLEAR REGULATORY COMMISSION

RPSB-99-12
(REV 0)

SOFTWARE QUALITY ASSURANCE PROCEDURES FOR NRC THERMAL HYDRAULIC CODES

by

Frank Odar

Reactor Plant System Branch
Division of Systems Technology
Office of Nuclear Regulatory Research

January 6, 1999
sqa9.doc

TABLE OF CONTENTS

Acknowledgments

The author is thankful for review and comments provided by the NRC staff. The NRC staff who participated in review and comment of this document were: Leo Beltracchi, Robert Brill, Terry Jackson, Vince Mousseau and Simon Smith. Special thanks are due to William Arcieri of Scientech Inc. for his discussions of Software Quality Assurance procedures.

1 INTRODUCTION

1.1 Purpose

The purpose of this document is to provide quality assurance procedures for development and maintenance of the NRC thermal hydraulic codes. These procedures will be used by the NRC staff, its contractors and partners in the code development and maintenance programs.
It describes the methods for qualification of a computer software.

1.2 Scope

Software quality assurance (SQA) is the planned and systematic actions to provide confidence that the software product meets established technical requirements. Quality assurance procedures ensure that software correctly performs all intended functions and does not perform any unintended function. SQA activities can be categorized as follows:

1) documentation of the software or software modules as they are developed,

2) verification and validation activities and their documentation,

3) nonconformance (error) reporting and corrective actions and their documentation,

4) acceptance testing and installation of the software and upgrading of code manuals,

5) configuration management, and

6) quality assessment and improvement.

This document, (Rev.0), addresses SQA activities in the first five items. The last item will be addressed in future revisions.

The application of SQA procedures in development of codes for NRC by DOE contractors is required by Management Directive, (M.D.), 11.7 "NRC Procedures for Placement and Monitoring of Work With the Department of Energy," Reference 1. M.D. 11.7 states, "All software development, modification, or maintenance tasks shall follow general guidance provided in NUREG/BR-0167, "Software Quality Assurance Program and Guidelines"", Reference 2. NUREG/BR-0167 provides general guidelines for development of NRC codes. This document provides procedures for development of thermal hydraulic codes. It is based on NUREG/BR-0167 and ANSI/ANS-10.4-1987, "Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry," Reference 3. Both standards take reference to other ANSI, ASME, IEEE and DOD standards.

This document has also benefitted from other documents on quality assurance procedures prepared by Scientech, Inc. and Los Alamos National Laboratory. Checklists are extracted from American National Standard ANSI/ANS-10.4-1987 with permission of the publisher, the American Nuclear Society, and from Quality Assurance Manual (Draft) prepared by Scientech. These checklists are useful in executing SQA activities.

For each project, an SQA plan will be developed by the project manager of the team actually developing or maintaining the code or a model. This plan will show how software quality assurance procedures will be applied to the specific project.

1.3 Responsibilities

1.3.1 Group/Program Manager - Branch Chief

The Group/Program Manager or Branch Chief is responsible for appointing a Project Manager / Principal Investigator and a Code Custodian for each code.

1.3.2 Project Manager / Principal Investigator

Project Manager or Principal Investigator is a person responsible for directing activities of the team where a code or model development project or a corrective maintenance activity is undertaken. Project Manager or Principal Investigator is responsible for preparation and execution of "Software Project Plan" and "Software Quality Assurance Plan," (SQAP). A guideline on preparation of "Software Project Plan" is not included in this document. An example of a project plan is provided in NUREG/BR-0167, Appendix A.

1.3.3 Code Custodian

The Code Custodian is responsible for maintenance of the software and its configuration control.

2 THE SOFTWARE LIFE CYCLE AND VERIFICATION & VALIDATION

The software life cycle provides the basis for planning and implementing a software development or maintenance project. A life cycle for a model development contains following phases:

1. Initial Planning
2. Requirements Definition
3. Software Design
4. Coding
5. Software Testing
6. Installation and Acceptance

During each development phase, specific products are developed. These products are evaluated, approved and controlled. Documents are reviewed, coding is tested and approved if test results meet acceptance criteria. Reviewing and testing of a software are basically part of verification and validation activities.

Verification is a process of ensuring that products developed in a phase meet the requirements defined by the previous phase. After a development work in a phase is completed, we verify that the work has been performed correctly; i. e., requirements defined for that particular phase have been fulfilled. Requirements are defined in the previous phase. A simple example demonstrating this process is presented below.

Let us consider development of a "break flow" model in an existing code. At the "Initial Planning" phase, NRC may consider the need for such a model. If such a model is needed, NRC will establish top level requirements. The next phase is "Requirements Definition" phase which develops a physical model which would meet top level requirements established in the previous phase. The phenomena, which would occur as the flow exits through the break, are represented in terms of a mathematical model. The mathematical model is documented and requirements for the design and accuracy of the model (acceptance criteria) are specified. In the next phase, which is the "Design" phase, the program is designed to meet the design and accuracy requirements of the mathematical model. The design specifies requirements for coding. In the "Coding" phase, the design is converted into computer instructions.

In the above example, there are three verification activities to be performed at the end each phase. Products developed at each phase are 1) Software Requirements Specifications (SRS), 2) Software Design and Implementation Document (SDID) and 3) Source Coding. The first verification activity is the review of the SRS. If the review confirms that the mathematical model represents physical phenomena and that the model is expected to calculate the flow rate with desired accuracy, the model is accepted. The second verification activity is the review of the SDID. If the design meets modeling requirements, it is accepted. The third verification activity is the review or inspection of computer instructions to ensure that coding is correctly developed as required in the design document. In short, verification can be considered to be the proof that the computer instructions correctly represent the design, that the design correctly represents the mathematical model, and that the mathematical model correctly represents the phenomena. Verification consists of a detailed examination of products developed in each phase to ensure consistency with requirements imposed by the previous phase.

The basic tools of verification are review, inspect, and audit. In the above example, these tools were sufficient to perform an adequate verification. However, in some cases, requirements may be very complex and the expected coding may be very extensive. In some cases, pilot coding to test some ideas may be necessary. In these cases, review and inspection may not be sufficient for verification. Some testing of some modules or part of coding may be necessary. This testing will be called verification testing since it tests correctness of the work performed during a development phase.

Depending on the size of the source code, testing of units of software (e.g., subroutines) and testing of collection of related units may be required. These activities are called *unit* and *integration* testing. These tests are determined as the units are developed. Test problems are generally not evident and they are not formally planned. This means that some of these tests are not included in the Testing Plan used in preparation of the SRD. Following definitions apply:

> *Unit Testing-* It is defined as testing of a unit of software such as a subroutine that can be compiled or assembled. The unit is relatively small; e.g., on the order of 100 lines. A separate driver is designed and implemented in order to test the unit in the range of its applicability.
>
> *Integration Testing-* It is defined as testing of a collection of related units that performs an identifiable functional requirement. It may be necessary to design and implement a separate driver to test the collection of units.

Unit and integration testing are considered "verification" since the results of testing are compared against the software <u>design requirements</u> identified in Software Design Implementation Document (SDID). They verify that the coding is correctly developed in the coding phase.

Document reviews are conducted using checklists. In this report, checklists from ANS standards are used. Coding is inspected and tested, if necessary. Audits are performed on selected items to assure that software quality assurance procedures are followed.

Verification is performed independently by different people preferably by peers. In the example presented above, first the contractor would perform verification activities before delivering documentation and the code. Next, NRC would independently conduct verification activities. In this example, verification activities to be conducted by NRC are: 1) Review and approve SRD, 2) Review and approve SDID and 3) Test and approve the coding. Results of all verification activities are documented. The extent of verification activities will be described in the Software Quality Assurance Plan.

Validation is a process of testing a software and evaluating results to demonstrate that the software meets its requirements as defined in Software Requirements Document (SRD). This step validates coding. It shows that not only the coding has been developed correctly but also code models, which represent the phenomena, provide accurate results as defined by the acceptance criteria. Testing is the primary method for software validation. Validation testing is a combination of Qualification and Acceptance testing. Two matrices, one for qualification and the other for acceptance testing are prepared. The acceptance test matrix includes all tests in the qualification test matrix plus some other tests selected by the sponsor (NRC).

In qualification testing, results obtained from the testing are compared to results from alternative methods, such as:

1. Comparison to theoretical solutions
2. Other validated computer programs
3. Experimental results (test data)
4. Standard problems with known solutions
5. Published data and correlations

The "Qualification Testing" is also called "Developmental Assessment" in thermal hydraulics. During testing, software results are obtained using different code options and input deck nodalizations. User guidelines for the software are developed during "Qualification Testing." It is expected that if appropriate user guidelines are used, test results would meet acceptance criteria. If results do not meet the acceptance criteria, modeling of the phenomena and/or user guidelines may be deficient. NRC will be informed of the results and corrective actions will be discussed.

If results in Qualification Testing meet acceptance criteria, the code would be delivered to NRC and be installed in NRC environment. At this point, additional acceptance testing using "Acceptance Test Matrix" may be performed using the same user guidelines established in Qualification Testing. This test matrix contains the matrix of qualification testing plus some other tests chosen by NRC. Acceptance testing is performed by people different from code developers. Preferably, it is conducted by the NRC staff in its operational environment. Formal test plans for both qualification and acceptance testing are required.

3 ELEMENTS OF SOFTWARE QUALITY ASSURANCE

Software quality assurance (SQA) is planned and systematic actions to provide confidence that the software product meets established technical requirements. The elements of the SQA for thermal hydraulic codes are listed below:

TABLE 1 - ELEMENTS OF SQA

Life Cycle	Development Product	Verification & Validation Activities	Checklist
Initial Planning	SOW, Project Plan SQA Plan (SQAP)	Management Review	
Requirements Definition	Software Requirements Specifications (SRS)	Verification of Requirements Verification of test plan and acceptance criteria	QA Form 03 QA Form 5A
Software Design	Software Design and Implementation Document (SDID)	Verification of Design	QA Form 04
Coding	Source Code Verification Testing Report	Verification of Source Code Verification of Program Integration Verification of Test Results	QA Form 06 QA Form 07 QA Form 05
Software Testing	Validation Testing Report	Validation of Program	QA Form 05 QA Form 08 QA Form 11
Installation and Acceptance	Installation Package Upgrading Program Documentation	Verification of Installation Package Verification of Program Documentation	QA Form 12 QA Form 09 QA Form 10

The first column shows different phases of the software development. Associated SQA elements are shown in the second, third and fourth columns. The second column shows development products produced in different phases. The third column identifies various verification and validation activities associated with different development products. The last column identifies the checklists which can be used to perform verification and validation activities. Appendix B contains all checklists to be used in the verification and validation process.

The first phase in the life cycle is "Initial Planning." One of the products of this phase is a set of initial requirements. These requirements are top level requirements and they are set by NRC. They are contained in the Statement of Work (SOW). Management reviews of these requirements, constitute verification activities. These initial requirements state "What the software will do" and not "How the software will do."

The next phase in the life cycle is the "Requirements Definition." In this phase, initial requirements are analyzed and the question of how initial requirements will be satisfied is answered. The next level of requirements; i. e., requirements' specifications, are established. These requirements state "What the specifications will be." They are used in the next phase where the software is designed.

As we progress from one phase to another, each phase will produce requirements for the following phase. The level of requirements becomes lower. Requirements at one level are a subset of requirements at the previous level. Each requirement shall be defined such that it is capable of being verified and validated by a prescribed method (e.g., review, inspection, analysis, or testing). In addition, each requirement shall be traceable throughout the software quality assurance process. Requirements shall have following characteristics:

- **Necessity**—Ensure that the requirement is necessary.
- **Attainable**—Ensure that the requirement is technically feasible and there is sufficient funding to perform necessary work.
- **Completeness**—Ensure that all necessary requirements are included.
- **Unambiguity**—Ensure that requirements are interpreted the same way by all readers.
- **Consistency**—Ensure that requirements do not conflict.
- **Verifiability**—Determine that a practical method exists to verify that each requirement is satisfied.
- **Modifiability**—Ensure that requirements are easy to modify correctly.
- **Traceability**—Determine that software requirements trace to the initial requirements.
- **Readability**—Ensure that readers can easily read and understand all requirements.

A discussion on writing good requirements is presented in Reference 5. For convenience, Reference 5 is reproduced in Appendix C.

3.1 Initial Planning

During the initial planning phase, the technical and managerial requirements for a particular program or modification are defined. Initial requirements' definition is the responsibility of the staff in the Office of Nuclear Regulatory Research, although contractor support may be used to aid in the planning process. There are two products produced at this stage:

1. Project Plan

2. SQA Plan (SQAP)

3.1.1 Project Plan

A project plan describes required software activities and contractual commitments. It is a baseline management plan. The plan contains following:

a. Project Background and Objectives
b. Description of Tasks, Responsibilities and Organization
c. Scheduling and Resources
e. Implementing SQA Plan

If the work is to be conducted by a contractor, information on the first three items is generally contained in the Statement of Work (SOW) or Request for Proposal (RFP). NRC prepares these three items. Initial requirements are contained in these three items. The plan for implementation of SQAP at the contractor site is developed by the contractor. It describes how SQAP is implemented. NRC also performs required reviews which is also a part of SQAP. If the entire work is to be performed by the NRC staff, a project plan will be prepared by the appropriate NRC staff.

A sample Project Plan is described in NUREG/BR-0167, Appendix A, Reference 2. This plan can be used as a guidance in preparing the Project Plan.

3.1.2 SQA Plan

At the start of each project, a Software QA Plan (SQAP) will be completed by the Project Manager or Principal Investigator. It will show the scope of the quality assurance activities to be performed in the project and will be consistent with the Project Plan. The plan may combine some of the development products of the SQA shown in Table 1. It may emphasize or de-emphasize some of the verification and validation activities. If a certain verification or validation activity or development of some documentation is to be de-emphasized, justification should be provided in the plan. The plan will basically address project needs and it will be designed specifically for the project. If the budget is not sufficient to perform all of the necessary quality assurance work, this will be stated in the plan and items which will not be covered by the quality assurance activities, will be identified. The management will be informed of the lack of coverage. Further guidance on preparation of an SQAP is provided in Appendix A.

3.2 Requirements Definition

3.2.1 Software Requirements Specifications (SRS)

This document will be based on initial overall requirements developed by the NRC. The SRS is a technical document that shall focus on technical specifications of the software. It will clearly describe analysis of each requirement and develop details and specifications showing how NRC requirements will be met. Following requirements will be specified.

I. <u>Functional Requirements:</u>

If the initial requirement is development of a new capability, first analysis of this requirement will be made. The theoretical basis and mathematical model consistent with the phenomena to be modeled are described. The range of parameters over which the model is applicable is specified. Relation of phenomena to code models is described. The SRS provides a detailed description of the selected model, including its range of applicability, scalableness to reactor plant applications, assessment base, and accuracy. It should include all figures, equations, and references necessary to specify the functional requirements for the design of the software.

If the initial requirement from NRC is a modification of an existing model or coding, the SRS will describe how it was done in the past and what will be done now including new functional requirements. It will discuss options considered in developing these requirements and reasons for rejection of some of the options. It should describe the new approach and how it will be implemented. It should include all figures, equations, and references necessary to specify the functional requirements for the design of the software.

II. <u>Performance Requirements:</u>

These requirements specify performance characteristics of the software or a modification. They address following items:

1) time-related issues of software operation, such as speed, etc.,
2) accuracy issues and acceptance criteria,
3) scalability

Resolution of speed, accuracy and scalableness issues require development of a test plan and acceptance criteria. In general, SRS should contain a requirement on accuracy of code predictions relative to the phenomena to be modeled. The code should be exercised using a test plan and results should meet acceptance criteria. The test plan will include a list of test problems that should provide complete coverage of all of the functional requirements. It will also discuss applicability of models to reactor systems since these models are to be tested in different scales. An example of how scaling issues are addressed, is provided in NUREG/CR-5249, "Quantifying Reactor Safety Margins," Reference 4. Note that discussion of scalableness may not be applicable to some type of code work; e.g., modernization of data structures.

The test plan is also called "Qualification Test Plan." The following information should be provided in the test plan:

a. The number and types of qualification problems to be completed,
b. The rationale for their choice, why was this problem chosen, which functional requirement does it test?
c. The specific range of parameters and boundary conditions for which successful execution of the problem set will qualify the code to meet specific functional requirements,
d. Descriptions of the code input test problems,
e. A description of what code results will be compared against (analytical solution, experimental data or other code calculation)
f. Significant features not to be tested and the reasons (for example, for complex codes, absolute qualification of every combination of options over every usable range of parameters is not practical)
g. Acceptance criteria for each item to be tested. Number and types of sensitivity calculations to be performed in order to develop user guidelines.
h. Discussion of scalableness, if applicable.

The Test Plan will address following items if applicable:

a. Compliance with software requirements
b. Performance at hardware, software, user, and operator interfaces
c. Assessment of run time, user guidelines, and acceptance criteria
d. Measures of test coverage and software maintainability
e. Hardware and software used in the testing.

III. Design Constrains

These requirements are constraints imposed on the source code that will restrict design options.

IV. Attributes

These requirements specify operation of the software, such as portability, excess control, maintainability, user-friendliness, etc.

V. **External Interfaces**

These are interfaces with people, hardware, and other software, including a description of the operational environment,

VI **Input and output requirements.**

These specify requirements for input and output of the software.

3.2.2 Verification and Validation Process

SRS reviews shall be performed by the individual designated on the SQAP in the organization where SRS has been prepared. NRC staff will perform a separate review. QA Form 03 Requirements Review Checklist and QA Form 5A Software Test Plan Review Checklist may be used as a basis for these reviews. Reviews should ensure that the requirements are complete, verifiable, consistent and technically feasible. Review should also assure that the requirements will result in a feasible and usable code. As reviews are completed, reviewers will sign an appropriate box in QAForm 03.

3.3 Software Design

3.3.1 Software Design and Implementation Document (SDID)

The design of the software is the foundation for implementing the requirements and constrains specified in the SRS. During this phase, the software design is developed. This phase is needed for all software development projects. The areas which can compromise the integrity and the robustness of the software design are identified below:

- **Modular Design**—Enhance the quality of software by dividing it into manageable, more understandable sets of interrelated components with clearly defined interfaces. Modular design contributes to maintainability by minimizing the ripple effect of design changes.

- **Interface Integrity**—Ensure the correctness of the interfaces between software components so that errors, such as invalid protocol and data, do not occur.

- **Data Integrity**—Ensure that the software operates in defined states by minimizing the occurrence of errors that could transition the software to an undefined state or allow an unintended function to be performed.

- **Error Handling**—Ensure that the software is robust and able to recover from an error by following a well-defined strategy.

Development of the software design is performed by the software development group, either at NRC or at the contractor's office. If necessary, a pilot computer code may be developed in order to facilitate the design. The pilot code should be tested using a verification test matrix designed for this purpose. Development of a pilot code will require NRC approval. After enough knowledge is gained from testing with the pilot code, SDID is prepared. After NRC approval of the SDID, the coding begins.

The SDID shall describe the logical structure, information flow, data structures, the subroutine and function calling hierarchy, variable definitions, identification of inputs and outputs, and other relevant parameters. The design document shall include a tree showing the relationship among modules and a database describing each module, array, variables, and other parameters used among code modules. The level and quality of the design documentation shall allow future modifications and improvements without having to re-engineer the program. The following shall be included in the SDID, as a minimum:

a. Descriptions of major design components related to specified requirements.

b. Technical description of a program (i.e., control flow, data flow, control logic, data structures, the routine and calling hierarchy, variable definitions, identification of inputs and outputs etc.).

c. Allowable / prescribed input and output ranges.

d. Design Implementation - Model implementation or integration brings together the source code with individual models to form an operational package to obtain desired results.

Any tools, techniques, or methodologies which must be employed during the V&V testing, shall be noted in the SDID. All this information is used to provide requirements for the coding phase.

3.3.2 Verification and Validation Process

SDID reviews shall be performed upon completion of the SDID. Reviews shall evaluate the technical adequacy of the design approach; assure internal completeness, consistency, clarity and correctness of the software design; and verify that the design is traceable to the SRS. QAForm 04, Software Design Review Checklist, may be used as a basis for this review. At the discretion of the Project Manager a Design Review Meeting may also be held. Results of verification test problems used during pilot code programming are reviewed. As reviews are completed, appropriate boxes in QAForm 04 are signed by reviewers.

3.4 Coding

3.4.1 Development of Source Code

Software coding is implementation of design requirements of the SDID. Work on Source Code will start after SRS and SDID have been prepared, reviewed and all comments are resolved. For software developed for the NRC the program status, data input and results, the code version, date and time of execution, and output parameter units, will be included in the program output. If printouts are not usually generated, the status will be clearly noted on the computer screen.

Coding standards in Reference 6 will be used. For convenience of the reader, Reference 6 is reproduced in Appendix D. All coding in Fortran 90 language should use standards in presented in Reference 7.

A verification test plan with defined acceptance criteria shall be developed by the code developer. Verification testing (defined in Section 2 as *unit* and *integration* testing) checks that each code module and groups of modules meet a program design requirement. Test planning can be done in parallel with the software design. The developer shall specify, as applicable, in Computer Software Testing Cover Sheet (QAForm 05):

a. The number and types of verification problems to be completed,
b. The rationale for their choice,
c. The specific portions/options of the program and range of parameters and boundary conditions for which successful execution of the problem set will verify the code related to specific design or specification requirements
d. Significant features not to be tested and the reasons (for example, absolute verification of every combination of options over every usable range of parameters may be not practical)
e. Acceptance criteria for each item to be tested.

The Test Plan will address, as applicable:

a. Compliance with the Software Requirement Specification
b. Performance at hardware, software, user, and operator interfaces
c. Assessment of run time, and accuracy
d. Measures of test coverage and software maintainability
e. Hardware and software used in the testing.

Note that these tests are verification tests. They are not validation tests. The purpose of verification tests at this point is to verify that the coding developed meets the requirements specified in the SDID and that the coding is done correctly. Description of the test plan, test matrix, input decks and testing results will be documented in "Verification Testing Report."

3.4.2 Verification and Validation Process

The Source Code Listing or update listing shall be reviewed for the following attributes. There will be sufficient explanations in comment cards in the listing which will permit review of these attributes:

a. Traceability between the source code and the corresponding design specification - analyze coding for correctness, consistency, completeness, and accuracy

b. Functionality - evaluate coding for correctness, consistency, completeness, accuracy, and testability. Also, evaluate design specifications for compliance with established standards, practices, and conventions. Assess source code quality.

c. Interfaces - evaluate coding with hardware, operator, and software interface design documentation for correctness, consistency, completeness, and accuracy. At a minimum, analyze data items at each interface.

QAForm 06, the Code Review Checklist, may be used as the basis for this review. Reviewers will sign an appropriate box in QAForm 06.

The update listing will be line by line inspected if these updates were created manually. If updates were created using Perl Scripts, or equivalent type of script, then these scripts will be documented and, inspected line by line. Inspection is detailed examination of lines of coding by an independent reviewer. The results of inspection will be reported in an inspection report. This inspection report shall contain the SQAP number, the date of updates, full listing of the updates, identifications for these updates, Perl Scripts and statements on the results of the inspection. Inspection shall address the question of whether or not updates would perform intended function and would not perform unintended function.

Reviewers will review the verification test plan and results of verification tests and signify approval by signing spaces in Computer Software Testing Cover Sheet, QA Form 05 and Verification Test Report Review Checklist, QA Form 07. If results of verification testing do not meet acceptance criteria and/or the test plan is not adequate, the coding will not be approved and it will be returned to the developer for corrections or further testing.

3.5 Validation Testing

3.5.1 Performing Testing and Preparation of Testing Report

Requirements (Acceptance criteria) for validation testing have been prepared in "Requirements Definition" phase. They are documented in SRS. Execution of the Validation (Qualification) test plan may be performed by individuals who are involved with the software development group. After code development is completed, testing shall begin. As a minimum all testing described in the SRS Test Plan will be done, however additional testing may be required by NRC. The input for the test problems shall be constructed from the description in the test plan. The results shall be plotted against the data for comparison. All testing activities shall be documented and shall include information on the date of the test, code version tested, test executed, discussion of the test results, and whether the software meets the acceptance test criteria. Results from the testing report will be incorporated into the developmental assessment (DA) manual by the code maintenance personnel. Test reports shall contain sufficient information as described below:

1. Test Reports shall describe the testing outlined in the Test Plan in sufficient detail

to allow an engineer of comparable qualifications to understand and, if necessary, reproduce the results. The report shall include nodalization diagrams, listing of the input deck, options used in constructing the deck and justification for their selection.

2. For validation (qualification) tests, code calculated results will be compared to results obtained by alternative means (exact solution, experimental data, other code calculations). A short description of these alternative methods should be included. For example, if experimental test data are used for comparison, the report will include a short description of the test facility, phenomena observed and availability and accuracy of measurements. It will include graphical display of selected parameters and calculated values. It will discuss the acceptance criteria and conclude whether or not the code will meet these criteria. It will discuss any user guidelines which are essential in meeting the acceptance criteria.

3. In validation (qualification) testing, when calculated code results are compared to results obtained from alternative methods such as test data, the reasons for differences should be understood and discussed in light of acceptance criteria. User guidelines will be developed, revised or confirmed based on analysis of results.

4. If limitations of the code are found during the testing process and acceptance criteria are not met, a list of those limitations will be presented and limitations will be explained in the test report. Test failures will be reviewed to ascertain adequacy of user guidelines and the soundness of the theory and design. If user guidelines are not adequate, sensitivity studies to improve user guidelines may be needed. Calculations should be done with NRC approval. If theory or design is faulty, modifications of requirements, design and implementation may be needed. Modifications to design may be performed with NRC approval. If acceptance criteria are unrealistic, they can be changed with NRC approval.

5. Reports shall have conclusions and recommendations if necessary. All conclusions shall be supported by analyses.

3.5.2 Verification and Validation Activities

Review of the Validation Test Report shall be made by an independent reviewer. QAForm 08 is the Validation Test Report Review Checklist. After the review is completed, the reviewer will sign the appropriate box in the form. The reviewer may provide an additional report discussing acceptability of the product.

3.6 Installation and Acceptance

3.6.1 Installation Package

The program installation package consisted of program installation procedures, files of the program, selected test cases for use in verifying installation, and expected output from these test cases.

3.6.2 Acceptance Testing

NRC will conduct acceptance testing in accordance with the accepted test plan. This plan will be developed by NRC. It may include following items: 1) Test cases in the installation package, 2) Selected test cases used by the contractor in Validation or Verification testing, and 3) Additional test cases as determined by NRC. See also Section 5.2.

3.6.3 Upgrading Program Documentation

The program documentation is revised and enhanced to provide a complete description of the program. Code manuals will be produced and updated concurrently with the code development process. A set of code manuals will cover following subjects:

- Theory & Numerical Methods Manual - This manual describes the theoretical basis, derivation, averaging, discretization, and solution of the conservation equations.

- Models & Correlations Manual - This manual describes the theoretical basis, physical models and correlations used to represent physical phenomena and includes discussions on their ranges of applicability, scaling considerations, assessment base, and accuracy.

- User's Manual - This manual shows the user how the code should be used. It provides information from code installation to post -processing including guidelines on input model preparation. It provides guidelines for selection of nodalization and code options.

- Programmer's Manual - This manual describes the code architecture from routine calling hierarchies to variable definitions. It should show control logic, the data flow, and data structure. It should provide enough detail such that an organization external to the development team can modify the code.

- Qualification Testing & Developmental Assessment Manual - This manual demonstrates the accuracy of the code predictions of phenomena observed in separate and integral effect tests. It provides a qualitative statement or a quantitative measure describing how the software meets the acceptance criteria provided that the code is used with a given set of user guidelines. It provides guidance on resolution of scaling issues so that code calculations are applicable to the reactor plant system. It also demonstrates how the models work together through integral facility tests.

The manuals will be developed in two sets: one "external" set to describe the released code version and the second "internal" set to describe updates reflecting recent code development and error correction activities. Two manual sets will reflect "released" and "developmental" code versions.

Code manuals will be updated after following documentation including their review and resolution of comments are completed:

1. Software requirements' specification - This document will provide information to update the Theory or Models & Correlations Manual.
2. Software Design and Implementation Document - This document will provide information to update the Programmer's Manual.
3. Listing of the Source Code Modifications - This listing will provide information to update the Programmer's Manual.
4. Verification and Validation Test Reports - Theses reports will provide information to update Qualification Testing & Developmental Assessment Manual.

3.6.4 Verification of the Installation Package

Verification of the installation package ensures that all elements to install the program are available and when the program is installed, it reproduces expected results. The program is installed following the procedures. Test cases which are supplied with the installation package are run. The output is checked against the output supplied with the installation package. This ensures that the program will produce the same results as the program executing in the development environment. QA Form 12 will be used for verification of the program installation package.

3.6.5 Verification of Upgrading of the Code Manuals

Program documentation is revised and enhanced to reflect upgrades in the code. Programmer and User Manuals are the primary sources of information about the computer program when new upgrades are to be made. They will be used by users, maintenance programmers, and V&V personnel to understand the program's objectives, characteristics and operation. Verification of these documents is performed to ensure that program documentation is completed, that the documentation conforms established standards, and that it provides a clear and correct description of the program. Checklists for verification of Programmer and User manuals are QA Form 09 and QA Form 10. The reviewer will review the manuals, provide answers to questions and sign these forms. The reviewer may also provide and document additional comments as necessary.

4 ERROR CORRECTIONS AND CODE MAINTENANCE

An error is a failure of the code or its documentation to meet its requirements. All errors will be reported in QA Form 13 by the users. NRC or NRC contractor evaluation and disposition of errors will be reported in QA Form 14. Error corrections are performed by code maintenance personnel, not code developers. Most error corrections are small (e.g., less than five subroutines affected, and no subroutines added or deleted). They require small effort; e.g., less than one staff month. They may originate from users or they may relate to problems discovered during code development. Because of their size and cost, error corrections do not require the same level of documentation as code development tasks.

All errors are evaluated for their criticality and level of importance. After the evaluation, resources required for corrective actions are identified and the impact of these corrective actions are discussed. See QA Form 14 for the documentation.

Tracking of errors and reporting their correction status, the number of errors found in the code, and criticality of open problems will be kept current.

Error corrections require only a single SQA package to be delivered to NRC. This package includes QA Forms 13 and 14. QA Form 13 requires following information:

(a) A description of the symptom of the problem (i.e., code failure method, or parameter plot). Plots will be attached to QA Form 13.

(b) A description of the root cause of the bug and a demonstration that all similar bugs have been caught.

QA Form 14 requires following information:

(a) A description of how the code was changed to correct the error, including data dictionary for major code variables added, deleted, or modified, and for each modified subroutine, a statement of the deficiency being corrected, or the functionality being added or deleted.

(b) The input deck(s) for the test problem(s). One of the test problems must address the symptom from the original user problem. Plots will be attached to QA Form 14.

(C) A documented patch files that fixes the error.

The NRC personnel should duplicate plots and verify that they are complete and reasonable. If the error correction is acceptable, the NRC personnel should approve the correction for inclusion into the configuration control system. If the correction is not acceptable, then the NRC will reject the coding for correction and send it back to the code developer.

5 CONFIGURATION CONTROL

Software configuration control involves updating, testing, storing, distributing and final dispositioning of the software. Configuration control is primarily performed by the code custodian.

5.1 Configuration Control File and Software Configuration Plan

Team members or contractors who developed a model or performed a corrective maintanance, are responsible for transmitting complete project software packages which include all updates and documentation required in the SQA work. The code custodian or project manager will ensure that all QA requirements have been met.

The code custodian is responsible for establishing and maintaining the software Configuration Control file. Access to this file shall be limited. Files and supporting documentation for each code revision shall be traceable. The code custodian shall prepare a Software Configuration Plan which shows how the files are maintained and how the changes to the code files will be performed. The plan will also show how revisions to documentation are maintained. The plan will document the configuration control file.

The configuration control file shall contain following items:

1. SQAP Number (Note that this number identifies individual updates and dates), computer hardware and operating systems for which it has been tested and any special limitation on the software.

2. Listing of all documentation with identification numbers. Electronic copies of all documentation produced as SQA procedures were applied.

3. Complete software source and update files with proper identifications, including auxiliary and library files necessary to update, compile and operate the software. Suggested naming convention for these files is presented in Table 2.

4. Directory which lists and describes the contents of all files contained in the Configuration Control file. Description of the directory structure that will be used to organize files.

5. Software commands necessary to update, compile, install, test, and operate the software. Instructions on using these commands. These changes will be performed using the CVS package. The approach is to uniquely identify code changes in the source code using CVS by a unique identification label. The naming convention for the identification label should allow a reviewer to trace code changes to revisions in the supporting documentation. The naming convention for the identification label shall be defined in the Software Configuration Plan.

6. Listing of the Acceptance test problems; i.e., Qualification test problems plus special test problems for the changes in the current version. Qualification test matrix and special test problems for the current version comprise acceptance test matrix.

7. A list of complete set of manuals describing the code and its use. Electronic files of these manuals.

TABLE 2

NAMING OF CONFIGURATION CONTROL FILES

File Name	Naming Convention
Platform Independent Source in CVS Format	.s
Code Library	.lb
Code Fixes	.fx
Fortran Source File in ASCII Format	.f
Header Files Containing Common Blocks and Other Global Definitions	.h
Object Files	.o
Executable Files	.x
Library Archive Files	.a

5.2 Software Acceptance

The Project Manager / Principle Investigator will determine which models and changes will be included in a new version of the software. The code custodian will create the new version. Acceptance testing will be performed by the Project Manager and acceptability of the new version will be determined after resolution of comments on acceptance testing report. After the acceptance, the code version will be archived and distributed.

The new version shall contain a block of information that includes the version name, creation date, and contents (e.g., names of updates). Acceptability is determined by successfully performing test runs in the acceptance test matrix and demonstrating that results meet acceptance criteria. The acceptance test matrix contains the qualification assessment test matrix and a list of special test problems. Special test problems are two kinds: a) Test Problems for Software Configuration Testing and b) Test Problems for Performance Testing.

Software Configuration Testing is performed to check that new or modified software is compatible with prior software versions; i.e., no interference with other updates and compiles correctly on intended platforms. This is particularly important when several updates developed by different contractors are to be combined to make a new version of the code. The matrix for configuration tests contains test cases from different *unit* and *integration* testing for different updates.

Performance test matrix contains some additional testing not included in the Qualification test matrix. The testing is performed using the user guidelines developed during qualification testing. Meeting the acceptance criteria ensures that new or modified software performs correctly with the established user guidelines. Testing of the qualification test matrix will be an automatic procedure while the other tests may not be automatic since they will be developed for the specific models and updates that the new version will contain. Testing will be performed on a set of computer platforms and operating systems as needed.

The Software Acceptance Testing Report shall be prepared by the Project Manager / Principle Investigator. The report shall be reviewed and the reviewer may use Checklist QA Form 11. Upon completion of comment resolution the Project Manager / Principle Investigator will sign appropriate location in SQA Form 01.

6 RECORDS

All documentation produced in applying the SQA procedures described in this document shall be maintained in both electronic (CD-ROM) and hardcopy form in NRC Headquarters and appropriate contractor or partner sites for code versions. In addition to the SQAP number, documentation shall include Job Code Number, Task number, Task Identification (Title) and a Search Code. This will permit expeditious retrieval of any documentation or a group of documentation on any part of code development including related correspondence.

Code versions which are not used shall be retired and archived. The archive shall consist of the source code and all documentation produced in support of the code including all SQA documents in a retrievable format such as CD-ROM for future reference.

7 REFERENCES

1. "NRC Procedures for Placement and Monitoring of Work with the Department of Energy," Management Directive, (M.D.), 11.7, May 1993.

2. "Software Quality Assurance Program and Guidelines," NUREG/BR-0167, February 1993.

3. "Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry," ANSI/ANS-10.4-1987

4. "Quantifying Reactor Safety Margins," NUREG/CR-5249, December 1989.

5. "Writing good Requirements," Proceedings of the Third International Symposium of the NCOSE, Volume 2, 1993. Also, http://www.incose.org/workgrps/rwg/writing.html

6. "Safer Subsets of Fortran 77", extracted from Appendix A of Hatton, L. (1992), "Fortran, C or C++ geophysical software development," Journal of Seismic Exploration, 1, p77-92, http://www.oakcamp.demon.co.ok/Lang.F77.

7. "Fortran 90 Handbook, Complete ANSI/ISO Reference," J. Adams, W. Brainerd, J. Martin, B. Smith, J. Wagener, published by McGraw Hill, 1992, ISBN 0-07-000406-4.

APPENDIX - A

GUIDANCE ON PREPARATION OF AN SQA PLAN

A-1 Software Quality Assurance Plan (SQAP)

At the start of each project, a Software QA Plan (SQAP) will be completed by the Project Manager or Principle Investigator. It will show the scope of the quality assurance activities to be performed in the project and will be consistent with the Project Plan. The plan may combine some of the development products of the SQA shown in Table 1. It may emphasize or deemphasize some of the Verification and Validation activities. If a certain verification or validation activity or a development of a certain documentation is to be deemphasized, a justification should be provided in the plan. The plan will basically address project needs and it will be designed specifically for the project. If the budget is not sufficient to perform necessary quality assurance items, this will be stated in the plan and items which will not be covered by the quality assurance activities, will be identified. If non-coverage of any item cannot be justified in the plan , the management will be informed of the non-coverage.

QAForm 01 is a SQAP Table. It shows a summary of implementation of the plan. The items in the table are discussed below.

A-2 SQAP Number

The SQAP will be assigned a number by the Project Manager / Principle Investigator. This number shall be in form of SQAPxxxxxx.W.VRN.YYMMUP, where:

xxxxxx is the code name such as TRAC-P, TRAC-B and RELAP5.

W represents an indicator for the status of the code. There are three different statuses for the code: Production, Released, Developmental. Definitions of these statuses are given below:

Production- The code is fully qualified in accordance with this procedure and uncertainties of predictions are quantified. W replaced by P.

Released- The code is fully qualified in accordance with this procedure. All verification and validation activities are completed; all documentation is completed and the code is ready for release. The code is released for further assessment by independent parties. W is replaced by R.

Developmental- The code is partially qualified, but it is in the process of complying with this procedure. This status is used for new development of the code undergoing many updates. W is replaced by D.

VRN represents the version of the code that the work will start; such as V25. After the work is completed the version number will be updated; such as V26.

YYMMUP represents the update number(s). The format is that the first two numbers indicate the year that updates started and the next two numbers indicate the month that updates started. The last two characters indicate identification of updates that this SQAP is to cover.

For example, if a contractor is going to improve the break flow model in the TRAC-P code and the work is going to start using version 11 in March 1998, the SQAP number would be SQAP.TRAC-P.D.V11.9803BR. In this case, BR indicates all updates related to the break flow model improvement.

A-3 Review and Schedules

Reviewers' names and completion dates of reviews will be indicated in QAForm 01 (SQAP Table). If a checklist is used for the review, the number of the checklist will be entered in QAForm 01. All checklists, if used, contain reviewer's name and signature and the date of completion of the review. If there are comments that need to be resolved, QAForm 02 should be used. The date of resolution of comments is entered both in QAForm 01 and QAForm 02. All documents shall have an ID number which will permit easy retrieval.

APPENDIX - B

CHECKLISTS

QA Form 01 SQAP TABLE SQAP#___________

Item	Preparer	Reviewer	Reviewer (NRC)	Checklist	Comments Resolved?	Completion Date	Document ID#
1. Project Plan		▨	▨	▨	▨		
2. Software Requirements Specification (SRS) inc. Qualification Assessment Test Plan							
3. Software Design Implement. Doc. (SDID)							
4. Verification Testing Plan and Test Report							
5. Source Code Listing							
6. Validation Test Report							
7. Installation Package							
8. Code Manuals	▨						

QA FORM 02

Computer Software V&V Review Comments SQAP #_______

Software Name:	Version:
Documentation I.D. Reviewed:	
Reviewer:	
Review based on: _ checklist (attach) _ other (explain method of review)	
Review Comments:	
Reviewed By: ____________________ Independent Reviewer	Date ____________
Response to Review Comments:	
Response By: ____________________	Date ____________
Response Accepted: ____________________ Independent Reviewer Accepted: ____________________ Project Manager (required for testing only	Date ____________ Date ____________

QA Form 03

Requirements Review Checklist - SQAP #_______________ Doc. ID______

Reviewer: **Date:**	Yes	No	NA
1. Does the SRS conform to the requirements specified by the sponsor?			
2. Are the requirements appropriate for the problem to be solved?	▨	▨	▨
a. Are the specified models, numerical techniques and algorithms appropriate for the problem to be solved?			
b. Will the program as specified solve the problem?			
3. Are the requirements clear and unambiguous?	▨	▨	▨
a. Can each requirement be interpreted in only one way?			
b. Are the requirements clearly organized and presented?			
c. Are program requirements clearly distinguished from other information that may be contained in the SRS?			
4. Are the requirements complete?	▨	▨	▨
a. Do requirements include all functions called for or implied by the Project Plan?			
b. Is the operational environment (hardware, operation system) of the program specified? If applicable, are timing and sizing constraints identified?			
c. Are design constraints specified?			
d. Does the specification include desired quality requirements, such as portability, maintainability, user friendliness?			
e. If the program is required to interface with other programs, is its behavior with respect to each defined?			
f. Are input and output requirements identified and described to the extent needed to design the program?			
5. Are the requirements internally consistent?	▨	▨	▨
a. Is the SRS free of internal contradictions?			
b. Are the specified models, algorithms and numerical techniques mathematically compatible?			
c. Are input and output formats consistent to the extent possible?			
d. Are the requirements for similar or related functions consistent?			
e. Are input data, computations, output, etc. required accuracies compatible?			
6. Are the requirements correct?	▨	▨	▨
a. Are all requirements consistent with the Project Plan?			
b. Are the requirements consistent with the properties of the specified operating environment, and any other programs with which the program must interface?			
c. Are descriptions of inputs and outputs correct?			
d. Do the requirements for models, algorithms and numerical techniques agree with standard references, where applicable?			
7. Are the requirements feasible?	▨	▨	▨
a. Are the specified models, algorithms and numerical techniques practical? Can they be implemented within system and development effort constraints?			
b. Are the required functions attainable within the available resources?			
c. Can the desired attributes be achieved individually and as a group? (ie, generally not possible to maximize both efficiency and maintainability.)			
8. Do the requirements make adequate provision for program V&V testing?	▨	▨	▨
a. Is each requirement testable?			
b. Are acceptance criteria specified?			
c. Are the acceptance criteria consistent with at least one of the following (circle all appropriate): Results obtained from similar computer programs; Solutions of classical problems; Accepted experimental results; Analytical results published in technical literature; Solutions of benchmark problems			
9.Do requirements avoid placing undue constraints on code design and implementation?			

QA Form 04

Software Design Review Checklist - SQAP#________________Doc. ID__________

Reviewer: **Date:**	Yes	No	NA
1. Does the SDID conform to the requirements specified in SRS?			
2. Is the SDID traceable to the SRS?	▨	▨	▨
a. Are all requirements implemented in the design?			
b. Are all design features consistent with the requirements?			
c. Are the specified numerical techniques appropriate for the problem to be solved?			
d. Are the specified algorithms appropriate for the problem to be solved?			
e. Is the structure of the design appropriate for the problem to be solved?			
f. Will the program as designed meet the requirements?			
3. Is the design clear and unambiguous?	▨	▨	▨
a. Can all design information be interpreted in only one way?			
b. Is the design information clearly organized and presented?			
c. Is the design sufficiently detailed to prevent misinterpretation?			
4. Is the design complete?	▨	▨	▨
a. Are all program inputs, outputs, and database elements identified and described tothe extent needed to code the program?			
b. Does the program design conform to its required operational environment?			
c. Are all required processing steps included?			
d. Are all possible outcomes of each decision point designed?			
e. Does the design account for all expected situations and conditions?			
f. Does the design specify appropriate behavior in the face of unexpected or improper inputs and other anomalous conditions?			
g. Are coding standards specified or referenced as applicable?			
h. If interface is required with other programs, is this provided for in the design? If applicable, does the design provide for reading and writing of external files?			
5. Is the design internally consistent?	▨	▨	▨
a. Is the SDID free of internal contradictions?			
b. Are the specified models, algorithms and numerical techniques mathematically compatible?			
c. Are input and output formats consistent to the extent possible?			
d. Are the designs for similar or related functions consistent?			
e. Are the accuracy and units of inputs, database elements and outputs that are used together in computations or logical decisions compatible?			
f. Are the style of presentation and level of detail consistent throughout the SDID?			
6. Is the design correct?	▨	▨	▨
a. Is the design logic sound, such that the program will do what is intended?			
b. Is the design consistent with the properties of the specified operation environment, and with any other programs with which the program must interface?			
c. Does the design correctly accommodate all required inputs, outputs and database elements?			
d. Do the models, algorithms and numerical techniques used in the design agree with standard references, where applicable?			
7. Is the design feasible?	▨	▨	▨

a. Are the specified models, algorithms and numerical techniques practical? Can they be implemented within system and development effort constraints?			
b. Can functions, as designed, be implemented within the available resources?			

QA Form 05

Computer Software Testing Cover Sheet - SQAP#__________Doc. ID_____

I.	
Software Name:	Version:
Code Author and Affiliation:	
Computer Type:	Program Language:
Applicable Testing: _ Verification Testing	Code Verifier:
_ Validation Testing	Code Validator:
II. Testing Plan Scope:	
III. Approved:	Date:
IV. Summary and Conclusion of Testing Activity:	

QA Form 05A

Software Test Plan Review Checklist - SQAP#______________Doc. ID_______

Reviewer: **Date:**	Yes	No	NA
1. Do the SRS and SDID contain information needed as a bases for testing?	▨	▨	▨
a. Are the requirements testable?			
b. Are the acceptance criteria specified?			
c. Are the acceptance criteria consistent with at least one of the following (circle all that apply): Results obtained from similar computer programs; Solutions of classical problems; Accepted experimental results; Analytical results published in technical literature; Solutions of benchmark problems			
2. Does the test plan include planning for Verification Testing and Validation Testing, as specified on the SQA Plan?			
3. Do documented test plans include reference to documents containing the requirements to be tested?			
4. Are requirements to be tested identified, with acceptance criteria ?			
5. Are the planned test cases adequate?	▨	▨	▨
a. Is the basis for selection of test cases documented? Is the rationale clear and valid?			
b. Does each test case have known and accepted results?			
c. Are dependencies between test cases identified?			
d. Is the application range of the software product, as defined by the requirements, adequately covered by the set of test problems?			
6. Is each testable requirement adequately covered?	▨	▨	▨
a. Is at least one test case provided for each requirement?			
b. If the requirement covers a range of values or capabilities, are test cases identified to cover the range adequately?			
c. Does documentation include demonstration of test to requirements, as in a traceability matrix?			
d. Do the tests include cases that are representative of the conditions under which the program will be used?			
7. Are the test case specifications complete?	▨	▨	▨
a. Are the test cases consistent with the planned cases that are listed?			
b. Is the specification for each test case complete?	▨	▨	▨
unique identification			
function(s) tested/objective(s)			
input, including modeling assumptions			
expected results			
test setup instructions			
hardware and software environment			
8. Is the specification for each test case adequate?	▨	▨	▨
a. Is input detail sufficient?			
b. Are expected results explicit, and specified with sufficient accuracy?			
c. Do evaluation criteria provide clear acceptance criteria for each test?			
d. Are all relevant databases, data files or libraries identified?			
9. Does the test planning provide for test databases or data files?			
10. Are test cases specified in sufficient detail for future reproducibility?			
11. Are instructions provided for disposition of test files and test results?			
12. Is configuration management provided for test databases, data files, and external programs?			
13. Can the planned testing be performed within the available resources?			

QA Form 06

Code Review Checklist - SQAP#______________Doc. ID________

Reviewer: **Date:**	Yes	No	NA
1. Does the Source Code conform to applicable standards? (As when, eg, the use of ANSI standard FORTRAN is required?			
2. Are sufficient comments provided to give an adequate description of each routine?			
3. Is the Source Code clearly understandable?	▨	▨	▨
a. Is ambiguous or unnecessarily complex coding avoided?			
b. Is the code formatted to enhance readability?			
4. Is the Source Code logically consistent with the Software Design Description?	▨	▨	▨
a. Are all featured of the design fully and correctly implemented in the code?			
b. Do all features of the coded program have their basis in the Software Design Description?			
5. Are all variables properly specified and used?	▨	▨	▨
a. Is the program free of unused variables?			
b. Are all variables initialized?			
c. Are array subscripts consistent?			
d. Are loop variables within bounds?			
e. Are constants correctly specified?			
f. Are proper units used with each variable?			
6. Is there satisfactory error checking?	▨	▨	▨
a. Are input data checked for applicable range?			
b. Are external data files checked to assure that the correct data file is being read and the data are in proper format?			
c. Are results of calculations checked for reasonable values?			
7. Do all subroutine calls transfer data variables correctly?	▨	▨	▨
a. Are the number of variables and the types of each variable the same in both the calling and called routine?			
b. Are labeled common variable names, type, location, and size of arrays consistent throughout the program?			
c. Are the data read from each file consistent with the data written to that file? (Are the number and type of variable and the unit numbers consistent?)			
8. Were draft code-related documents reviewed with the source code for completemess, correctness, and consistency? (ref. Exhibit P7-7 if desired)			
9. Is the code status (PRODUCTION, VERIFIED, RELEASE or DEVELOPMENTAL indicated on the program output?			

QA Form 07

Verification Test Report Review Checklist -SQAP#____________Doc. ID______

Reviewer: **Date:**	Yes	No	NA
1. Do unit test results show that:	▨	▨	▨
a. Each major logical path within the routine was tested?			
b. Each routine was checked for appropriate minimum, maximum, and average sets of variables?			
c. The routine reproduces identical results given identical input data?			
1. Does the integrated program conform with the resource requirements on the operating system?	▨	▨	▨
a. Does the program meet storage requirements for memory and external devices?			
b. Does the program meet timing and sizing requirements?			
2. Does the integrated program interface properly with external files?			
a. Does the program properly read and write external files?			
b. Does the program properly read user-specifies input data?			
3. Are all elements of the integrated program properly identified?	▨	▨	▨
a. Has the source code been verified?			
b. Has the compiler been identified?			
c. Have special user libraries been verified?			
d. Have system libraries been identified?			
e. Has the loader been identified?			
4. Does the program link correctly?	▨	▨	▨
a. Are all required subroutines present?			
b. Does the module linkage specification create a properly linked program?			
c. Are all routines loaded into proper segments?			

Reviewer: **Date:**	Yes	No	NA
d. Are global and local labeled common blocks properly specified for each segment?			
5. Are the interfaces between functional units correct?	▨	▨	▨
a. Are labeled COMMON blocks consistent (FORTRAN)?			
b. Are argument lists passed consistently?			
c. Are I/O data file names consistent?			
d. Are I/O structures consistent and correct?			
6. Is the control language for building the integrated program correct?	▨	▨	▨
a. Are proper compiler options used?			
b. Are proper libraries specified?			
c. Are loading options consistent for initialization of variables and obtaining load maps?			
7. Is the control language used for execution proper?	▨	▨	▨
a. Are all files properly specified?			
b. Are execution time limits correct?			
c. Are external data files properly attached and saved?			
8. Are the data libraries that are used for execution appropriate?	▨	▨	▨
a. Do the data libraries conform to the Design Specification in structure and format?			
b. Is the data in the libraries adequate for proper execution?			
9. Is there sufficient evidence to verify that the processing of data and transmission of data between modules is correct?	▨	▨	▨
a. Have all I/O files been checked?			
b. Have labeled COMMON blocks been checked?			
c. Have variables passed to routines been checked?			

QA Form 08

Validation Test Report Review Checklist - SQAP#__________Doc. ID______

Reviewer: Date:	Yes	No	N/A
1. Does the Software V&V Report meet the requirements of SRS?			
2. Do test results adequately identify the software and hardware under test, including support software such as operation system, test drivers, test data?			
3. If test cases were supplied with the installation package, did they produce results identical to the output supplied with that package?	▨	▨	▨
a. Could all the test cases be performed?			
b. Were all results identical to previous results?			
c. Were differences in results clearly understood and justified?			
4. Do the reported test results comply with the test planning documentation?	▨	▨	▨
a. Is the program tested completely identified?			
b. Are test planning documents, and any other relevant documents, referenced?			
c. Is the test environment (location, hardware configuration, support software) completely and accurately described?			
d. Are deviations from test plans, if any, described and justified?			
e. Is a summary of test results provided?			
f. Is program performance with respect to requirements evaluated?			
g. Are recommendations provided for retesting, program acceptance, conditional acceptance, as appropriate?			
h. Are results of each test case reported in detail?			
i. Is a test log included to provide a narrative record of the progress of the testing?			
j. Were unexpected occurrences documented, as well as expected pass/fail results?			
k. Are problems and their resolution documented?			
5. Does the documentation of the test results accurately reflect the testing performed?	▨	▨	▨
a. Does the summary of test results accurately reflect the test output produced?			
b. Is the report conclusion a realistic and accurate reflection of the test results?			
c. Are recommendations for retesting and/or acceptance sound and based on the test results?			
d. Do the descriptions of the test case results accurately reflect actual test outputs?			
e. Is the test log, if any, complete and consistent with actual test output?			
f. Are problem reports complete and consistent with actual test output?			
6. Were all the test cases executed correctly?	▨	▨	▨
a. Do reported test results indicate performance of each test case in the specified environment, using the documented specifications?			
b. Is there an explanation for any deviation from the specified test environment or procedures?			
c. Is there a problem report for each deviation from expected results?			
d. Were correct input data used for each test case?			
e. Is the output of each test case accurately reported or attached?			
7. Has each requirement been tested adequately?	▨	▨	▨
a. Does the set of test results corresponding to each requirement adequately cover the range?			
b. Has each test result for this requirement satisfied its acceptance criteria?			
c. Does the combination of test case results for this requirement meet the acceptance criteria?			
8. Is the total set of requirements met?	▨	▨	▨
a. Is every acceptance criterion met satisfactorily?			
b. Are there any test results that indicate unrepeatable, unreliable or unexpected program behavior?			
c. Are the test results consistent with the initial Task Proposal for the program?			

QA Form 09

User's Manual Review Checklist - SQAP# _______

Reviewer: **Date:**	Yes	No	NA
1. Is the level of detail in the manual appropriate for its intended users?			
2. Does the user section of the manual describe the program's functions, options, limitations and accuracy?			
3. Is the description of user input adequate?			
a. Are all input data requirements specified?			
b. Are formats fully specified?			
c. Are valid ranges of input values specified?			
d. Are theoretic limitations specified?			
e. Are units specified?			
4. Are the necessary run instructions provided?			
5. Does the user section of the manual provide guidance for interpreting output?			
6. Are error messages adequately explained?			
7. Are hardware and operation system requirements specified?			

QA Form 10

Programmer's Manual Review Checklist - SQAP# _______

Reviewer: **Date:**	Yes	No	NA
1. Is the information in the manual internally consistent?			
a. Is the user information consistent with the programmer's information?			
b. Is the user information an accurate reflection of the coded program?			
c. Is the information in the manual clear, unambiguous and well organized?			
d. Is the manual free of internal contradictions?			
2. Are all the elements of the program (e.g., source, library, compiler, loader) properly identified?			
3. Are instructions (including control language) provided for compiling, loading and running the program?			
4. Is the design correct?			
a. Is the design logic sound, such that the program will do what is intended?			
b. Is the design consistent with the properties of the specified operating environment, and with any other programs with which the program must interface?			
c. Does the design correctly accommodate all required inputs, outputs and database elements?			
d. Do the models, algorithms and numerical techniques used in the design agree with standard references, where applicable?			
5. Are the specified models, algorithms and numerical techniques practical? Are they implemented properly within the system constraints?			
6. Are the specified models, algorithms and numerical techniques mathematically compatible?			
7. Is the design free of internal contradictions?			

QA Form 11

Program Validation Checklist - SQAP#____________

Reviewer: **Date:**	Yes	No	NA
1. Has each requirement been tested adequately?	▨	▨	▨
a. Does the set of test results corresponding to each requirement adequately cover the range?			
b. Has each test result for this requirement satisfied its evaluation criteria?			
c. Does the combination of test case results for this requirement meet the acceptance criteria?			
2. Is the total set of requirements met?	▨	▨	▨
a. Is every acceptance criterion met satisfactorily?			
b. Are the test results consistent with the initial Statement of Problem for the program?			
c. Are there any test results that indicate unrepeatable, unreliable or unexpected program behavior?			

QA Form 12

Verification of the Installation Package SQAP#____________

Reviewer: **Date:**	Yes	No	NA
1. Are sufficient materials available on the program installation tape to permit rebuilding and testing of the installed program?	▨	▨	▨
a. Are the necessary elements from the following list available?			
Source Code			
User-supplied library routines			
Module linkage specifications			
External file structure definitions			
Control Language for Installation			
External Data Libraries to be used by the program			
Test Cases			
Control Language for Execution			
Output Produced by the Test Cases.			
b. Are the format and content of the tape properly identified in the installation procedures for easy reading of the files?			
c. Are the installation procedures clearly understandable to allow installation and checkout?			
2. Can the program be rebuilt from the installation package?	▨	▨	▨
a. Can the program source be recompiled and reloaded in the same manner as before?			
b. If there are changes in rebuilding, do these changes affect the functional operation of the program?			
3. Do the test cases produce results identical to output supplied with the installation package?	▨	▨	▨
a. Can all test cases be performed?			
b. Are all results identical to previous results?			
c. Are differences in results clearly understood and justified? (such as new date and time on printed output)			

QA Form 13 **Trouble Report -Reporting**

Trouble Report No.____________
(to be entered by NRC)

To report a problem, error, or code deficiency, enter all of the following information.

Code Name:

Version:

Date:

Submitted by (name):

Submitted by (organization):

Address:

Phone Number:

E-mail Address:

Classification of the Problem or Deficiency: (Check one or more)

___ Input Processing Failure
___ Code Execution Failure
___ Restart/Renodalization Failure
___ Unphysical Result
___ Installation Problem
___ Other

Provide following items:

1. Input deck(s)
2. Description of the symptom of the bug
3. Plots, if available

Computer Hardware Type / Computer Operating System (include version):

User's Determination of the Criticality of the Problem:

QA Form 14 **Trouble Report - Disposition**

Trouble Report No.__________

NRC's or Contractor's Determination of the Criticality of the Problem:

Organization Assigned for Corrective Action:

Provide following items:

1. A description of the root cause of the bug and a demonstration that all similar bugs have been caught.

2. A description of how the code was changed to correct the bug, including data dictionary for major code variables added, deleted, or modified, and for each modified subroutine, a statement of the deficiency being corrected or the funtionality being added or deleted. Following format should be used:

```
Subroutine Report

Subroutines Deleted:

subroutine1.f:  statement of subroutine's function and why that
functionality is no longer necessary, or what subroutines now
perform the function.  Refer to other subroutines with their full
name (e.g. TempM.f).

subroutine2.f:  similar statement.

Subroutines Added:

subroutine5.f: statement of subroutine's function.

subroutine8.f: statement of subroutine's function.

Subroutines Modified:

subroutine23.f:  statement of what changes are being made, the
deficiency being corrected, or the functionality being added or
deleted.
```

3. Input deck(s) for test problem(s). One of the test problems must address the symptom from

the original user problem.

4. A documented patch file that fixes the bug.

Date on which Nonconformance is Closed:

APPENDIX - C

GOOD REQUIREMENTS

Published in the *Proceedings of the Third International Symposium of the NCOSE - Volume 2, 1993*. Prepared by the Requirements Working Group of the International Council on Systems Engineering, for information purposes only. Not an official position of INCOSE.

Writing Good Requirements

(A Requirements Working Group Information Report)

Ivy Hooks
Compliance Automation, Inc.
17629 Camino Real, Suite 207
Houston, Texas 77058

Abstract. The primary reason that people write poor requirements is that they have had no training or experience in writing good requirements. This paper will address what makes a good requirement. It will cover some of the most common problems that are encountered in writing requirements and then describe how to avoid them. It also includes examples of problem requirements and how to correct them.

INTRODUCTION

If you or your staff are having problems with writing good requirements, you may benefit from guidance in how to write good requirements. The college courses you took probably never mentioned the subject of requirements. Even if you have taken classes in system engineering or program management, you may have had only an introduction to the subject of writing requirements. If you are using existing specifications for guidance, you may be using poor examples. The information provided in this paper is used in training people to write good requirements and generally results in a step function improvement in the quality of requirements.

An important aspect of system engineering is converting user "needs" into clear, concise, and verifiable system requirements. While this paper primarily discusses system level requirements, it is equally applicable at all lower levels.

The first section discusses what is a good requirement. This is followed by a discussion of common problems and how to avoid them. It also includes examples of problem requirements and how to correct them.

GOOD REQUIREMENTS

A good requirement states something that is **necessary**, **verifiable**, and **attainable**. Even if it is verifiable and attainable, and eloquently written, if it is not necessary, it is a good requirement. To be verifiable, the requirement must state something that can be verified by examination, analysis, test, or demonstration. Statements that are subjective, or that contain subjective words, such as "easy", are not verifiable. If a requirement is not attainable, there is little point in writing it. A god requirement should be clearly stated.

Need. If there is a doubt about the necessity of a requirement, then ask: *What is the worst thing that could happen if this requirement were not included?* If you do not find an answer of any consequence, then you probably do not need the requirement.

Verification. As you write a requirement, determine how you will verify it. Determine the criteria for acceptance. This step will help insure that the requirement is verifiable.

Attainable. To be attainable, the requirement must be technically feasible and fit within budget, schedule, and other constraints. If you are uncertain about whether a requirement is technically feasible, then you will need to conduct the research or studies to determine its feasibility. If still uncertain, then you may need to state what you want as a goal, not as a requirement. Even if a requirement is technically feasible, it may not be attainable due to budget, schedule, or other, e.g., weight, constraints. There is no point in writing a requirement for something you cannot afford -- be reasonable.

Clarity. Each requirement should express a single thought, be concise, and simple. It is important that the requirement not be misunderstood -- it must be unambiguous. Simple sentences will most often suffice for a good requirement.

COMMON PROBLEMS

The following is a list of the most common problems in writing requirements:

- **Making bad assumptions**
- **Writing implementation (HOW) instead of requirements (WHAT)**
- **Describing operations instead of writing requirements**
- **Using incorrect terms**
- **Using incorrect sentence structure or bad grammar**
- **Missing requirements**
- **Over-specifying**

Each of these are discussed in detail below.

BAD ASSUMPTIONS

Bad assumptions occur either because requirement authors do not have access to sufficient information or the information does not exist. You can eliminate the first problem by documenting the information critical to the program or project, including:

- Needs
- Goals
- Objectives
- Constraints
- Missions
- Operations Concept
- Budget
- Schedule
- Management/Organization

This information then must be made available to all authors. You can create and maintain a list of other relevant documents and make these easily accessible to each author. If you have automated the process, you can offer documents on-line and you can filter the information within the documents so that individual authors can get copies of only the data that they need.

In the second case where information does not exist, the requirement author should document all assumptions with the requirement. When the requirement is reviewed, the assumptions can also be reviewed and problems quickly identified. It is also useful to document the assumptions even if the authors were provided the correct information. You cannot ensure that all authors have read all the information or interpreted it correctly. If they document their assumptions, you will avoid surprises later.

IMPLEMENTATION

A specification was released for the development of a requirements management tool. The first requirement was to "provide a data base". The statement is one of implementation and not of need, and it is common to find such statements in requirement specifications. Specifications should state *WHAT* is needed, not *HOW* it is to be provided. Yet this is a common mistake made by requirement writers. Most authors do not intend to state implementation, they simply do not know how to state the need correctly.

To avoid stating implementation, ask yourself *WHY* you need the requirement. In the example cited, it can be seen that by asking *WHY*, the author can define all of the needs that the system must meet and will then state the real requirements, e.g.:

- *provide the capability for traceability between requirements*
- *provide the capability to add attributes to requirements*
- *provide the ability to sort requirements*

These requirements state *WHAT* is needed, not *HOW* to accomplish it. Each of the above listed requirements might result in a data base type of system, but the requirement for the data base was not needed.

There are two major dangers in stating implementation. The one most often cited is that of forcing a design when not intended. If all the needs can be met without a data base, then why state the need for a data base. If they cannot be met, another, or better way, then a data base will be the solution -- whether or not there was a requirement for a data base.

The second danger is more subtle and potentially much more detrimental. By stating implementation, the author may be lulled into believing that all requirements are covered. In fact, very important requirements may be missing, and the provider can deliver what is asked for and still not deliver what is wanted. Providing a data base will not be sufficient for someone needing a requirements management tool that need to be stated as requirements.

At each level of requirements development the problem of needs versus implementation will occur. At the system level the requirements must state WHAT is needed. The system designer will determine HOW this can be accomplished and then must define *WHAT* is needed at the subsystem level. The subsystem designer will determine *HOW* the need can be met and then must define *WHAT* is needed at the component level.

To ensure that you have not stated implementation, ask yourself *WHY* you need the requirement. If this does not take you back to a real need statement, then you are probably stating a need and not implementation.

The Implementation Trap. If you have been doing design studies at a low level, you may begin to document these results as high level requirements -- this is the implementation trap. You will be defining implementation instead of requirements.

An example of this occurred during the definition of the Assured Crew Return Vehicle (ACRV) requirements. An individual submitted a requirement like this:

- *The ACRV System shall enter when sea state is at TBD conditions.*

The ACRV had no requirement for a water landing -- that was a design option. The individual had been working with that design option, and from previous Apollo experience known that crew rescue was possible only in certain sea sates.

When asked WHY the requirement was needed, the individual stated that the crew could not be left in the module for a lengthy period of time, thus the landing needed to be where and when sea states could accommodate crew rescue. He had a valid requirement -- but not the one he had written. Whether the ACRV landed on water or land, removing the crew within a limited time period was essential. Thus the real requirement was:

- *The ACRV System shall provide for crew removal within TBD time of landing.*

The question *WHY* will resolve most implementation requirement errors. Always ask *WHY* a requirement is needed to insure that you have not fallen into this lower level requirement trap.

OPERATIONS VS. REQUIREMENTS

This problem is somewhat similar to the implementation problem. The Simplified Aid for EVA Rescue (SAFER) project provides examples of this problem.

The author intended to write an environment requirement in the SAFER Flight Test Article (FTA) Specification. The statement was written as a description of operations, not a requirement about the environment.

- *The SAFER FTA shall be stowed in the Orbiter Airlock Stowage Bag for launch landing, and on-orbit stowage.*

The real requirement is:

- *The SAFER FTA shall be designed for the stowage environment of the Airlock Storage Bag for launch, entry, landing, and on-orbit, as defined in TBD.*

The next requirement again describes the operations and is confusing.

- *The SAFER FTA shall be operated by an EVA Crew member wearing EMU sizes small through extra large without limiting suit mobility.*

The statement was rewritten and resulted in a requirement and a design goal. The design goal is needed because no quantifiable requirement can be written regarding suit mobility.

- *The SAFER FTA shall be designed for use with EMU sizes small through extra large.*
- *The SAFER FTA should not limit EVA crew member mobility.*

The danger in stating operations, instead of a requirement is (1) the intent may be misunderstood and (2) determining how to verify can be a problem.

USE OF TERMS

In a specification, there are terms to be avoided and terms that must be used in a very specific manner. Authors need to understand the use of ***shall***, ***will***, and ***should***:

- Requirements use *shall*.
- Statements of fact use *will*.
- Goals use *should*.

These are standard usage of these terms in government agencies and in industry. You will confuse everyone if you deviate from them. All *shall* statement (requirements) must be verifiable, otherwise, compliance cannot be demonstrated.

Terms such as are, *is, was,* and *must* do not belong in a requirement. They may be used in a descriptive section or in the lead-in to a requirements section of the specification.

There are a number of terms to be avoided in writing requirements, because they confuse the issue and can cost you money, e.g.

- Support

- But not limited to
- Etc.
- And/Or

The word support is often used incorrectly in requirements. Support is a proper term if you want a structure to support 50 pounds weight. It is incorrect if you are stating that the system will support certain activities.

- *WRONG: The system shall support the training coordinator in defining training scenarios.*

- *RIGHT: The system shall provide input screens for defining training scenarios. The system shall provide automated training scenario processes.*

The terms ***but not limited to***, and ***Etc.*** are put in place because the person writing the requirements suspects that more may be needed than is currently listed. Using these terms will not accomplish what the author wants and can backfire.

The reason the terms are used is to cover the unknown. The contractor will not increase the cost in the proposal because you added these terms. The only way to get the work added is to place an analysis task in the Statement of Work (SOW) to determine if more items need to be added to the list. In the SOW you can control what effort the contractor will expend to address these unknowns. If more items are found, you may have to increase the scope of the contract to cover the additions.

If you have these terms in your requirements specification, the contractor may use them as an excuse for doing unnecessary work for which you must pay. You cannot win by using the terms in the specification.

The term ***and/or*** is not appropriate in a specification. If you use and/or and the contractor does the or he has met the terms of the contract. Either you want item 1 and item 2 or you will be satisfied with item 1 or item 2. Again, if you use the term or, then the contractor has met the terms of the contract if he does either item.

REQUIREMENT STRUCTURE/GRAMMAR

Requirements should be easy to read and understand. The requirements in a system specification are either for the system or its next level, e.g. subsystem. Each requirement can usually be written in the format:

- *The System shall provide ....*
- *The System shall be capable of ....*
- *The System shall weigh ....*
- *The Subsystem #1 shall provide ....*
- *The Subsystem #2 shall interface with .....*

Note: The name of your system and the name of each subsystem appears in these locations. If you have a complex name, please use the acronym, or your document will have many unneeded pages just because you have typed out a long name many times.

Each of these beginnings is followed by *WHAT* the System or Subsystem shall do. Each should generally be followed by a single predicate, not by a list. There are situations where a list is appropriate, but lists are over-used. Since each item in the list must be verified, unless all items will be verified by the same method and at the same time, it is generally not appropriate to put items in a list.

Requirement statements should not be complicated by explanations of operations, design, or other related information. This non-requirement information should be provided in an introduction to a set of requirements or in rationale.

You can accomplish two things by rigorously sticking to this format. First, you avoid the Subject Trap. Second, you will avoid the Subject Trap. Second, you will avoid bad grammar that creeps into requirements when authors get creative in their writing.

Subject Trap. A system specification contains requirements for the system, e.g., you may want to require that the system provide control. A set of requirements might be written that reads as follows:

- *The guidance and control subsystem shall provide control in six degrees of freedom.*
- *The guidance and control subsystem shall control attitude to 2+/- 0.2 degrees.*
- *The guidance and control subsystem shall control rates to 0.5 +/- 0.05 degrees/second.*

The subject trap is created because the author has defined a guidance and control subsystem. Controlling attitude and rate is a system problem, it requires not only a guidance and control subsystem but also a propulsion subsystem to achieve these attitudes and rates. Determining what subsystems will be required to accomplish the requirements is part of the design process The requirements should be written from the system perspective, as follows:

- *The system shall provide six degrees of freedom control.*
- *The system shall control attitude to 2 +/- 0.2 degrees.*
- *The system shall control rates to 0.5 +/- 0.05 degrees/second.*

The author of the original requirements was not trying to define the lower level breakout. He probably comes from a control background and sees the system from that perspective and hence writes requirements that way. The flow down of requirements, to all affected segments, elements, and subsystem, will be badly affected if these requirements are not written correctly.

Bad Grammar. If you use bad grammar you risk that the reader will misinterpret what is stated. If you use the requirements structure suggested above, you will eliminate the bad grammar problems that occur when authors try to write complex sentences and use too many clauses.

Another solution is to write requirements as bullet charts. When the content is agreed upon a good writer can convert the information into a sentence for the specification.

Authors will also try to put all that they know in a single sentence. This results in a long complex sentence that probably contains more than one requirement. Bullet charts or one good editor can alleviate this problem. The following requirement contains multiple requirements.

- *The ACRV System shall provide special medical life-support accommodations for one ill or injured crew member consisting of medical life-support and monitoring equipment and the capability of limiting impact accelerations on that crew member to be not greater than.... for a total impulse not to exceed ....*

The requirement needs to be broken into at least four requirements and it could use a lead-in such as:

- *The ACRV will be used as an ambulance for an ill or injured crew member. Only one crew member will accommodate at a time. The following define the unique requirements for this capability.*
  - *..provide medical life-support accommo-dations for one crew member*
  - *..provide monitoring equipment for one crew member*
  - *..limit impact accelerations to the ill or injured crew member to no greater than...*
  - *..limit total impulse to the ill or injured crew member to ...*

UNVERIFIABLE

- ***Every requirement must be verified.***

Because every requirement must be verified, it is important to address verification when writing the requirements. Requirements may be unverifiable for a number of reasons. The following discusses the most common reason -- use of ambiguous terms.

Ambiguous Term. A major cause of unverifiable requirements is the use of ambiguous terms. The terms are ambiguous because they are subjective -- they mean something different to everyone who reads them. This can be avoided by giving people words to avoid. The following lists ambiguous words that we have encountered.

- Minimize
- Maximize
- Rapid
- User-friendly
- Easy
- Sufficient
- Adequate
- Quick

The words ***maximize*** and ***minimize*** cannot be verified, you cannot ever tell if you got there. The words minimum and maximum may be used is the context in which they are used can be verified.

What is ***user-friendly*** and what is ***rapid***? These may mean one thing to the user or customer and something entirely different to a designer. When you first begin writing your requirements, this may be what you are thinking, but you must write the requirements in terms that can be verified. If you must use an ambiguous term in first draft documents, put asterisks on either side of the term to remind yourself that you are going to have to put something concrete in the requirement before you baseline the document.

There may be cases where you cannot define, at your level, exactly what is needed. If this is the case, then you should probably be writing a design goal, not a requirement. You can do this by clearly indicating that your statement is a goal, not a requirement. Use of the word should, instead of the shall, will denote a design goal.

MISSING

Missing items can be avoided by using a standard outline for your specification, such as those shown in Mil-Std-490 or IEEE P1233, and expanding the outline for your program.

Many requirements are missed because the team writing the requirements is focused on only one part of the system. If the project is to develop a payload, the writers will focus on the payload's functional and performance requirements and perhaps skip other important, but less obvious, requirements. The following is a checklist of requirement drivers you need to consider:

- Functional
- Performance
- Interface
- Environment
- Facility
- Transportation
- Deployment
- Training
- Personnel
- Reliability
- Maintainability
- Operability
- Safety
- Regulatory
- Security
- Privacy
- Design constraints

You will need to develop detailed outlines for your specification for the functional and performance requirements, and in perhaps other areas.

You may also have a number of requirements that you must include by reference. In particular, those standards that define quality in different disciplines (materials and processes) or for different projects.

Detailed requirements analysis is necessary to assure that all requirements are covered. There are a number of approaches to performing requirements analysis and a number of tools for doing this work. Detailed requirements analysis is beyond the scope of this paper.

OVER-SPECIFYING

The DoD has stated that over-specification is the primary cause of cost overruns on their programs. Over-specifying is most often from stating something that is unnecessary or from stating overly stringent requirements.

Unnecessary Items. Unnecessary requirements creep into a specification in a number of ways. The only cure is careful management review and control.

People asked to write requirements will write down everything they can think of. If you do not carefully review each requirement and why it is needed before baselining the specification, the result will be a number of unneeded requirements.

Example: The Space Station Training Facility (SSTF) had a requirement for a high-fidelity star field. The author knew that a new high-fidelity star field was being developed for the Shuttle Mission Simulator (SMS) and assumed they might as well put the same thing in the SSTF. The crew needs a background to view outside the Space Station, but there is no need for a high-fidelity star field, since they do not use the stars for navigation.

The requirement needs to be written for a visual background for crew orientation. The design process will determine if using the SMS star field is a cost effective solution or it something simpler is adequate and more cost effective.

Example: A number of SSTF requirements were deleted when their authors were queried as to the need. The need they stated was that "it would be nice to have". Most program do not have budgets for *nice to have items.*

Unnecessary requirements can also appear after baseline if you let down your review and control process. In ACRV a number of requirements were added after the initial baseline that were not needed. One such instance occurred because of an error in the baseline document.

Example: The baseline document had two requirements:

- *The ACRV System shall be capable of operating over a planned operational life of thirty (30) years.*
- *The Flight Segment shall provide an operational life of 30-years for the flight elements.*

The second requirement, for the Flight Segment, was not required since the System requirement was adequate. The action that should have been taken was to delete the Flight Segment requirement. Instead, two more requirements were added to require a 30-year operational life of the other two segments.

At least one other requirement was added to ACRV that was a duplicate of an existing requirement. The wording of the two differed only slightly and their rationale was the same. It requires careful attention to detail to avoid this type of problem.

Over Stringent. Most requirements that are too stringent are that way accidentally, not intentionally. A common cause is when an author writes down a number but does not consider the tolerances that are allowable.

Thus, you should not state that something must be a certain size, e.g., 100 ft.2, if it could just as easily be 100 +/- 10 ft.2. You do not need to ask that something deliver a payload to exactly 200 n.m. if greater than or equal to 200 n.m. is acceptable.

Some of the major horror stories of the aerospace industry deal with overly-stringent requirements. One contractor was severely criticized for charging $25,000 per coffee pot in airplanes built for the government. But the requirements for the coffee pot were so stringent, that the plane could have crashed and yet no coffee could spill. It cost a great deal to develop the coffee pot and to verify that it met its requirements. Each copy had to be built to a stringent design.

The solution to this problem is to discuss the tolerances allowable for any value and then to write the requirement to take into consideration those tolerances. Each requirement's cost should also be considered.

AUTHOR'S BIOGRAPHY

Ivy F. Hooks is President and CEO of Compliance Automation, Inc., a woman-owned small business. She has over thirty years experience, 20 with NASA, where she was the separation integration manager for the Space Shuttle and also manager of Shuttle flight software verification, and 10 in private industry. For the past eight years, she has specialized in requirements definition and management, marketing the CAI product Vital Link, as well as conducting training seminars and performing consulting work for government and industry.

APPENDIX - D

A FORTRAN 77 STANDARD

Safer Subsets of Fortran 77

The following is extracted from Appendix A of Hatton, L. (1992) "Fortran, C or C++ for geophysical software development", Journal of Seismic Exploration, 1, p77-92.

1.0 Features which should be avoided

1.1 Unreachable code.
Reduces the readability and therefore maintainability.

1.2 Unreferenced labels.
Confuses readability.

1.3 The EQUIVALENCE statement except with the project manager's permission.
This statement is responsible for many questionable practices in Fortran giving both reliability and readability problems. Permission should not be given lightly. A really brave manager will unequivocally forbid its use. Some programming standards do precisely this.

1.4 Implicit reliance on SAVE. (This prejudices re-usability).
A particular nasty problem to debug. Some compilers give you SAVE whether you specify it or not. Moving to any machine which implements the ANSI definition from one which SAVE's by default may lead to particularly nasty run and environment sensitive problems. This is an example of a statically detectable error which is almost impossible to find in a source debugger at run-time.

1.5 The computed GOTO except with the project manager's permission.
Often used for efficiency reasons when not justified. Efficiency should never precede clarity as a programming goal. The motto is "tune it when you can read it".

1.6 Any Hollerith.
This is non-ANSI, error-prone and difficult to manipulate.

1.7 Non-generic intrinsics.
Use generic intrinsics only on safety grounds. For example, use REAL() instead of FLOAT().

1.8 Use of the ENTRY statement.
This statement is responsible for unpredictable behaviour in a number of compilers. For example, the relationship between dummy arguments specified in the SUBROUTINE or FUNCTION statement and in the ENTRY statements leads to a number of dangerous practices which often defeat even symbolic debuggers.

1.9 BN and BZ descriptors in FORMAT statements.
These reduce the reliability of user input.

1.10 Mixing the number of array dimensions in calling sequences.
Although commonly done, it is poor practice to mix array dimensions and can easily lead to improper access of n-dimensional arrays. It also inhibits any possibility of array-bound checking which may be part of the machine's environment. Unfortunately this practice is very widespread in Fortran code.

1.11 Use of BLANK='ZERO' in I/O.
This degrades the reliability of user input. See also 1.9.

1.12 Putting DO loop variables in COMMON.
Forbidden because they can be inadvertently changed or even lead to bugs in some optimising compilers.

1.13 Declarations like REAL R(1)
An old-fashioned practice which is frequently abused and leads almost immediately to array-bound violations whether planned or not. Array-bound violations are responsible for a significant number of bugs in Fortran.

1.14 Passing an actual argument more than once in a calling sequence.
Causes reliability problems in some compilers especially if one of the arguments is an output argument.

1.15 A main program without a PROGRAM statement.
Use of the program statement allows a programmer to give a module a name avoiding system defaults such as main and potential link clashes.

1.16 Undeclared variables.
Variables must be explicitly declared and their function described with suitable comment. Not declaring variables is actually forbidden in C and C++.

1.17 The IMPLICIT statement.
Implicit declaration is too sweeping unless it is one of the non-standard versions such as IMPLICIT NONE or IMPLICIT UNDEFINED.

1.18 Labelling any other statement but FORMAT or CONTINUE.
Stylistically it is poor practice to label executable statements as inserting code may change the logic, for example, if the target of a DO loop is an executable statement. This latter practice is also obsolescent in Fortran 90.

1.19 The DIMENSION statement.
It is redundant and on some machines improperly implemented. Use REAL etc. instead.

1.20 READ or WRITE statements without an IOSTAT clause.
All READ and WRITE statements should have an error status requested and tested for error-occurrence.

1.21 SAVE in a main program.
It merely clutters and achieves nothing.

1.22 All referenced subroutines or functions must be declared as EXTERNAL. All EXTERNALS must be used.
Unless EXTERNAL is used, names can collide surprisingly often with compiler supplied non-standard intrinsics with strange results which are difficult to detect. Unused EXTERNALS cause link problems with some machines, leading to spurious unresolved external references.

1.23 Blank COMMON.
Use of blank COMMON can conflict with 3rd. party packages which also use it in many strange ways. Also the rules are different for blank COMMON than for named COMMON.

1.24 Named COMMON except with the project manager's permission.
COMMON is a dangerous statement. It is contrary to modern information hiding techniques and used freely, can rapidly destroy the maintainability of a package. The author has bitter, personal experience of this ! Some company's safety-critical standards for Fortran explicitly forbid its use.

1.25 Use of BACKSPACE, ENDFILE, REWIND, OPEN, INQUIRE and CLOSE.
Existing routines for each of these actions should be designed and must always be used. Many portability problems arise from their explicit use, for example, the position of the file after an OPEN is not defined. It could be at the beginning or the end of the file. The OPEN should always therefore be followed by a REWIND, which has no effect if the file is already positioned at the beginning. OPEN and INQUIRE cause many portability problems, especially with sequential files.

1.26 DO loops using non-INTEGER variables.
The loop may execute a different number of times on some machines due to round-off differences. This practice is obsolsecent in Fortran 90.

1.27 Logical comparison of non-INTEGERS.
Existing routines for this should be designed which understand the granularity of the floating point arithmetic on each machine to which they are ported and must always be used. Many portability problems arise from it's explicit use. The author has personal experience whereby a single comparison of two reals for inequality executed occasionally in a 70,000 line program caused a very expensive portability problem.

1.28 Any initialisation of COMMON variables or dummy arguments is forbidden inside a FUNCTION, (possibility of side-effects).
Expression evaluation order is not defined for Fortran. If an expression contains a function which affects other variables in the expression, the answer may be different on different machines. Such problems are exceedingly difficult to debug.

1.29 Use of explicit unit numbers in I/O statements.
Existing routines to manipulate these should be designed and must always be used. Many portability problems arise from their explicit use. The ANSI standard only requires them to be non-negative. What they are connected to differs wildly from machine to machine. Don't be surprised if your output comes out on a FAX machine !

1.30 CHARACTER*(N) where N>255.
A number of compilers do not support character elements longer than 255 characters.

1.31 FORMAT repeat counts > 255.
A number of compilers do not support FORMAT repeat counts of more than 255.

1.32 COMMON blocks called EXIT.
On one or two machines, this can cause a program to halt unexpectedly.

1.33 Comparison of strings by other than the LLE functions.
Only a restricted collating sequence is defined by the ANSI standard. The above functions guarantee portability of comparison.

1.34 Using the same character variable on both sides of an assignment.
If character positions overlap, this is actually forbidden by the standard but some compilers allow it and others don't. It should simply be avoided. The restriction has been removed in Fortran 90.

1.35 Tab to a continuation line.
Tabs are not part of the ANSI Fortran definition. They are however easily removable if used only to code lines and for indentation. If they are also used for continuation (like the VAX for example), it means they become syntactic and if your compiler does not support them, removing them is non-trivial.

1.36 Use of PAUSE
An obsolescent feature with essentially undefined behaviour.

1.37 Use of '/' or '!' in a string initialised by DATA.
Some compilers have actually complained at this !

1.38 Using variables in PARAMETER, COMMON or array dimensions without typing them explicitly before such use.
e.g.
PARAMETER (R=3)
INTEGER R
Some compilers get it wrong.

1.39 Use of CHAR or ICHAR.
These depend on the character set of the host. Best to map onto ASCII using wrapper functions, but almost always safe today.

1.40 Use of ASSIGN or assigned GOTO.
An obsolescent feature legendary for producing unreadable code.

1.41 Use of Arithmetic IF.
An obsolescent feature legendary for producing unreadable code.

1.42 Non-CONTINUE DO termination.
An obsolescent feature which makes enhancement more difficult.

1.43 Shared DO termination for nested DO loops.
An obsolescent feature which makes enhancement more difficult.

1.44 Alternate RETURN.
An obsolescent feature which can easily produce unreadable code.

1.45 Use of Fortran keywords or intrinsic names as identifier names.
Keywords may be reserved in future Fortran standards. The practice also confuses readability, for example,
IF (IF(CALL)) STOP=2
Some people delight in this sort of thing. Such people do not take programming seriously.

1.46 Use of the INTRINSIC statement.
The ANSI standard is particularly complex for this statement with many exceptions. Avoid.

1.47 Use of END= or ERR= in I/O statements. (IOSTAT should be used instead).
Using END= and ERR= with associated jumps leads to unstructured and therefore less readable code.

1.48 Declaring and not using variables.

This just confuses readability and therefore maintainability.

1.49 Using COMMON block names as general identifiers, where use of COMMON has been approved.
This practice confuses readability and unfortunately, compilers from time to time.

1.50 Using variables without initialising them.
Reliance on the machine to zero memory for you before running is not portable. It also produces unreliable effects if character strings are initialised to zero, (rather than blank). Always initialise variables explicitly.

1.51 Use of manufacturer specific utilities unless specifically approved by the project manager.
This simply reduces portability, in some cases pathologically.

1.52 Use of non-significant blanks or continuation lines within user-supplied identifiers.
This leads both to poor readability and to a certain class of error when lists are parsed, (it may have been a missing comma).

1.53 Use of continuation lines in strings.
It is not clear if blank-padding to the end of each partial line is required or not.

1.54 Passing COMMON block variables through COMMON and through a calling sequence.
This practice is both illegal and unsafe as it may confuse optimising compilers and in some compilers simply not work. It is a very common error.

2.0 Other undesirable features

2.1 An IF..ELSEIF block IF with no ELSE.
This produces a logically incomplete structure whose behaviour may change if the external environment changes. A frequent source of "unexpected software functionality".

2.2 DATA statements within subroutines or functions.
These can lead to non-reusability and therefore higher maintenance and development costs. If constants are to be initialised, use PARAMETER.

2.3 DO loop variables passed as dummy arguments.
See also section 1.12.

2.4 Equivalencing any arrays other than at their base, even if use of EQUIVALENCE has been approved.
Some machines still have alignment problems and also modern RISC platforms rely on good alignment for efficiency. So at best, it will be slow and at worst, it will be wrong.

2.5 Equivalencing any variable with COMMON, even if use of EQUIVALENCE and COMMON has been approved.
This rapidly leads to unreadable code.

2.6 Type conversions using the default rules, either in DATA or assignment statements.
Type conversions should be performed by the programmer - state what you mean. For example:
R = I Wrong
R = REAL(I) Right