# Systematic Development of Requirements Documentation for General Purpose Scientific Computing Software

Spencer Smith

*Computing and Software Department, McMaster University*
*smiths@mcmaster.ca*

## Abstract

*This paper presents a methodology for developing the requirements for general purpose scientific computing software. The first step in the methodology is to determine the general purpose scientific software of interest. The second step consists of a commonality analysis on this identified family of general purpose tools to document the terminology, commonalities and variabilities. The commonality analysis is then refined in the third step into a family of specific requirements documents. Besides fixing the variabilities and their binding times, each specific requirements document also shows the relative importance of the different nonfunctional requirements, for instance using the Analytic Hierarchy Process (AHP). The new methodology addresses the challenge of writing validatable requirements by including solution validation strategies as part of the requirements documention. To illustrate the methodology an example is shown of a solver for a linear system of equations.*

## 1. Introduction

Scientific computing is defined as the use of computer tools to analyze or simulate mathematical models of real world systems of engineering or scientific importance so that we can better understand and predict the system's behaviour. Requirements elicitation, analysis and documentation are important, but often neglected, stages in developing scientific computing software. Requirements documentation is important because scientific computing problems are complex, with many sources of potential ambiguities in terminology, and notation. Moreover, decisions on how to handle unusual situations, such as division by zero and indeterminancy, should be determined early in the process rather than left as decisions for the implementor. Another reason that requirements documentation is important is that the only way to judge the correctness and reliability of scientific software is by comparison to a specification of the requirements. In scientific computing, practitioners often argue over the relative merits of different designs based on their own opinion of what the governing Nonfunctional Requirements (NFRs) should be. Therefore, an advantage of documenting requirements is that they can specify the relative importance of the various NFRs, and thus diffuse potential criticism because NFR trade-offs are explicitly documented. Another significant benefit of appropriate and rigorous requirements documentation is that the range of the scientific computing program's applicability can be clearly specified by documenting assumptions and data constraints. Even though requirements documentation can improve the quality of scientific software with respect to such qualities as correctness, reliability, verifiability, productivity, usability, understandability, maintainability, reusability and portability, requirements in scientific computing are usually documented in an *ad hoc* manner, rather than following a systematic engineering approach.

A systematic approach to requirements documentation has been effectively used in other application areas, such as with business applications [12] and for real-time systems, such as the shutdown systems of the Darlington nuclear generating station [9] and flight guidance and weapons systems [5]. However, the characteristics of scientific software mean that methods that have been successful elsewhere will have to be modified and adapted [16]. A new requirements template has been proposed for scientific computing problems [16, 17], but this template is suited to documenting the requirements for specific physical models, not for general purpose software. The distinction between specific and general purpose scientific computing is that in the first case the emphasis is on a specific problem, such as solving for the forces in a structure, or the temperature in a heated plate. In the second case the emphasis is on building general purpose tools that can be used for many different specific problems. Some general purpose tools include mesh generators, finite element analysis programs, linear solvers, root finding software, interval arithmetic subroutines, etc. This paper modifies the previous template [16] as part of the proposal for a new methodology for documenting the requirements for general purpose scientific computing software.

14th IEEE International Requirements Engineering Conference (RE'06)
0-7695-2555-5/06 $20.00 

The first section below describes the main challenges for pre-design documentation of general purpose scientific computing software. The next section outlines the proposed methodology that addresses these challenges by refining a commonality analysis of the related general purpose tools into a family of requirements documents. After this, a specific example is presented for documenting a solver for a linear system of equations.

## 2. Challenges

Ideally software requirements should be unambiguous, validatable and abstract. Although the mathematical nature of scientific computing facilitates writing unambiguous requirements, challenges certainly exist for attaining the other two qualities. A methodology for developing requirements for general purpose scientific software must address the problem of writing validatable and abstract requirements. In addition, a useful methodology should also overcome the challenges of specifying realistic NFRs and of capturing and reusing existing knowledge. Details on all of these challenges are discussed below. The challenges listed here provide the motivation and rationale for the newly developed methodology.

### 2.1. Validatable

Any requirements specification for scientific computing will need to address the challenge of writing validatable requirements. The source of the challenge is that scientific software differs from most other software because the quantities of interest are continuous, as opposed to discrete. Many of the functional requirements will specify the output behaviour for given inputs, where both the inputs and outputs are continuously valued variables such as time, velocity, displacement, temperature, pressure, concentrations, etc. Validating the requirements is difficult because there are an infinite number of potential input values and what is even more challenging, the correct value for the output variable is unknown *a priori*. In fact, the purpose of many scientific computing tools is to solve problems that are difficult or impossible to solve without the software, so in many cases the true solution is unknown. The challenge is highlighted by the fact that even when a functional requirement is unambiguously stated, the requirement may not necessarily be validatable. For instance, the following requirement is in general extremely difficult to validate: "Given $dy/dt = f(t, y)$ and $y(t_0) = y_0$, find $y(t_n)$, where $y(t)$ is a function ($y : \mathbb{R} \rightarrow \mathbb{R}$), $f(t, y)$ is a function ($f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$), $t$ is an independent variable (often time), $t_0$ is an initial value for $t$ and $t_n$ is the final value for $t$." This requirement would typically be accompanied by an NFR that would specify the allowed accuracy, for instance by stating the following: "The relative error in $y(t_n)$ should be less than $\epsilon$, where the relative error is defined as $|y_{true} - y|/y_{true}$ with the subscript $true$ referring to the true solution." Unfortunately for an arbitrary $f(t, y)$ the true solution will in general be unknown, which is why validating the requirement is so challenging.

### 2.2. Abstract

In addition to the challenge of writing validatable requirements, general purpose scientific computing also poses a challenge for writing abstract requirements. Although writing an abstract requirement is not actually difficult, writing an abstract requirement that provides enough information, and that is realistic, is challenging because of the emphasis in scientific computing on the NFRs of accuracy and speed. For instance, the requirement, "solve the system of linear equations represented by the equation $Ax = b$ " is abstract, but without details on the possible assumptions about the structure of the equations, the problem is too general to solve efficiently. In many cases it may be possible to add assumptions about the structure of the equation to this requirement, but in these cases a realistic requirements specification will often need to accommodate later implementation decisions. For instance, in the system of equations example, it may be possible, as it is in the case of structural mechanics problems, to assume that the matrix ($A$) representing the system of equations will be positive definite ($x^T Ax > 0$ for any vector $x$). However, if the requirements do not consider implementation details and are simply written to include an input constraint that $A$ be positive definite, the implementation will suffer because testing for positive definiteness can require a similar amount of computation effort as solving the system of equations.

Ideally requirements should say "what" is required and not "how" to do it, but in the case of scientific computing many programs are written with the requirement of implementing a specific algorithm. The challenge then is to switch the emphasis away from the algorithm and instead focus on the specification of the problem to be solved, the NFRs, the tradeoffs between NFRs, the possible assumptions and the input constraints. Once the requirements have been properly specified to this level of detail, the selection of an appropriate algorithm can move to the design stage where it belongs.

### 2.3. Nonfunctional Requirements

In scientific computing it is the NFRs, like accuracy, efficiency, portability, usability, etc., that often distinguish designs. Unfortunately, as mentioned previously, requirements for scientific software are difficult to validate. For

example, proving accuracy requirements, which requires a priori error analysis, is a difficult mathematical exercise that results in an error bound that is "weaker than it might have been because of the necessity of restricting the mass of detail to a reasonable level" [21]. Validation through testing is also challenging because success in one test case does not imply success for a "nearby" test case, since that nearby test case may include a singularity that was not present in the initial problem. In addition to the validatability challenge, the context sensitive tradeoffs between the NFRs can also be difficult to specify. Simply specifying a priority for each NFR is not adequate, as many in the scientific computing community would give most of the NFRs the same high priority ranking. Another challenge for the NFRs is to specify them in a realistic way so that they are not simply stated as absolute quantitative requirements. For instance, the following requirement has arbitrary absolute bounds on two NFRs: "The solution should have less than 10% error and be calculated in less than 20 seconds." Besides the challenge of validating this requirement, there is also the problem that in most uses of scientific computing tools, the user would likely tolerate a "failed" program that provides 11% error and a computation time of less than 25 seconds. The challenge then is to specify the NFRs as relative requirements between competing algorithms and different software packages.

### 2.4. Capture and Reuse Existing Knowledge

Any methodology for developing requirements for general purpose scientific software cannot ignore the enormous wealth of knowledge, software, best practices and expertise that currently exist in the field. The challenge is to capture the current state of the art and not write the requirements documentation to imply that mostly new code will be developed, as in many cases a feasible solution will involve integrating existing software libraries. Moreover, the documentation approach should facilitate the reuse of software and of the requirements documentation. Currently reuse does not happen as frequently as it could in scientific computing, in part because practitioners do not know exactly what the existing software is designed to do, what its limitations are and how well it performs with respect to the NFRs of accuracy and performance. Furthermore, if the requirements documentation facilitates reuse of the documentation, then the effort invested in its development will certainly pay off over time. Ideally, a library of requirements documentation for scientific software will exist in the future so that people designing a large project will be able to choose the appropriate models from the library and incorporate them into their project.

## 3. Overview of Proposed Methodology

The first step in the methodology is to determine the general purpose scientific software of interest. The second step consists of a Commonality Analysis (CA) on this identified family of general purpose tools. The CA can be seen as a method for summarizing the requirements for all potential tools that are considered to be within the scope of the project. The CA includes documentation of terminology, commonalities (including goals and theoretical models) and variabilities (including assumptions, input variabilities and output variabilities). The CA is then refined in the third step into a family of specific Software Requirements Specification (SRS) documents, with different documents for different contexts. Besides fixing the variabilities and their binding times, each specific SRS also shows the relative importance of the different NFRs of importance in scientific computing (correctness, reusability etc.). The technique suggested for documenting the relative priority of the requirements is the Analytic Hierarchy Process (AHP). Another important component of the SRS is the software validation strategies, which include listing potential test cases, comparison to existing tools and suggested inspection protocols. The final step in the proposed methodology consists of using the SRS to assist in selecting between different design alternatives.

### 3.1. Commonality Analysis

In some situations it is advantageous to develop a collection of related software products as a program family. The idea is that if the software products are similar enough, then it should be possible to predict what the products have in common, what differs between them and then reuse the common aspects and thus support rapid development of the family members. The idea of program families was introduced by Dijkstra [4] and later investigated by Parnas [7, 8]. More recently, Weiss [1, 19, 20] has considered the concept of a program family in the context of what he terms Family oriented Abstraction, Specification and Translation (FAST) [3, 18]. Other approaches to developing program families, also known as software product lines, can be found in references [2, 10]

In the approach advocated by Weiss, the first step is a CA, which consists of systematically identifying and documenting the commonalities that all program family members share, the variabilities between family members and the terminology used in describing the family. A CA provides a systematic way of gaining confidence that a family is worth building and of deciding what the family members will be. In the case of general purpose scientific computing, a CA has intuitive appeal given the proliferation of scientific computing tools available. Each family of soft-

ware tools shares commonalities because all family members have the same mathematical underpinnings. As an example, the general purpose tool of a mesh generator has been shown to be a suitable candidate for development as a program family [14, 15] because mesh generators meet the three hypotheses for a program family proposed by [19]: the redevelopment hypothesis (most software development involved in producing the family is redevelopment), the oracle hypothesis (the types of changes that are likely to occur during the software's lifetime are predictable) and the organizational hypothesis (designers and developers can organize the software, as well as the development effort, in a way that predicted changes can be made independently).

A CA is proposed as an initial step in developing requirements for general purpose scientific software because it provides the following benefits [19, 20]: a starting point for the design of a domain specific languages (DSL); a basis for a common design for all family members; a historical reference; a basis for reengineering a domain; and, a basic training reference for new software developers. In addition, a CA helps address two of the challenges identified in Section 2: i) managing the abstraction level of the requirements, and ii) assisting with knowledge capture.

With respect to writing abstract requirements (Section 2.2) the CA helps by first focusing one's thoughts on the theoretical mathematical model. This mathematical model is the ideal case, which cannot usually be fully realized on a computer, but conceptually it is easy to understand and it can be stated abstractly. The CA allows the analyst to bring this theoretical model closer to reality by specifying simplifying assumptions, which form an important class of variabilities between program family members. As an example, the following theoretical model can be proposed for a root finding program: "given a function $f(x)$ and an interval $\{x|x_{lower} \leq x_{upper}\}$, return the points where $f(x) = 0$." Some potential assumptions may be that $f(x)$ is continuous on the interval, or that $f(x)$ has at least one sign change on the interval, etc.

With respect to capturing existing knowledge (Section 2.4), a CA is an ideal document. Its creation facilitates a systematic consideration of the general purpose program family, which increases the analysts confidence that their understanding of the problem is complete. Once the CA document is complete, communication between experts is improved and hopefully the document can assist in consensus building and in having stakeholders reach a mutual understanding. Furthermore, if a standard template is followed, then comparison of different tools will be possible. Over time it is possible to imagine building a library of CAs for scientific software, which will allow reuse of the knowledge in the future. The CA also assists in capturing the current state of the art because it provides a convenient framework for summarizing the existing literature and software on a given general purpose scientific computing problem. This summary of the existing work is useful because it highlights what family members have yet to be built; that is, the CA shows where the holes are in the currently available set of general purpose scientific computing tools.

A potential template for use when documenting a CA for general purpose scientific software is presented in Figure 1. The template includes a section listing potential system contexts, user characteristics and system constraints. The CA covers all the potential members of a program family, so it is not possible to know exactly what information to place in these sections; however, there will often be information that can be recorded on typical uses of the program family members. Although the information in this section cannot be presented as variabilities because it does not represent requirements, the hints provided in the CA can later be refined in the corresponding sections of the SRS.

1. Reference Material: a) Table of Contents b) Table of Symbols c) Abbreviations and Acronyms
2. Introduction: a) Purpose of the Document b) Organization of the Document
3. General System Description: a) Potential System Contexts b) Potential User Characteristics c) Potential System Constraints
4. Commonalities a) Background Overview b) Terminology Definition c) Goal Statements d) Theoretical Models
5. Variabilities a) Input Assumptions b) Calculation c) Output
6. Traceability Matrix
7. References

**Figure 1. Proposed Commonality Analysis Template**

A "Terminology" section is common place in requirements documentation. In the case of the CA its inclusion is motivated by a need to clarify the concepts presented later in the SRS and to serve as a reference aid. The contents of this section consist of a list of mathematical concepts and their exact meaning. This section should provide enough information to allow understanding of the later sections "Goal Statement," "Theoretical Model," and "Input Assumptions." The terminology section is necessary in scientific computing for an unambiguous requirements specification because terminology often has subtly different meanings, even in very similar contexts. As an example, the same symbol $\sigma$ is used to represent stress, conductivity, the Stefan-Boltzmann

constant for radiative heat transfer, the standard deviation, etc. Potential confusion exists even if $\sigma$ is known to represent stress because there is still ambiguity about whether the stress is defined with respect to the original geometry, or with respect to the deformed geometry.

The motivation of the goal statement section of the CA is to capture the goals in the requirements process. A goal, in this context, is a functional objective the system under consideration should achieve. The goal statements do not include nonfunctional objectives because the NFRs are not commonalities between family members. Goals provide criteria for sufficient completeness of a requirements specification and for requirements pertinence. Goals will be refined in the Section "Theoretical Models." The goal statements are intended to be written at a level that is easy to understand. The goal statements should briefly summarize the commonalities shared by all program family members.

The "Theoretical Models" section specifies the theory that all members of the general purpose scientific computing family share. The model is presented as it would be presented in a mathematics textbook. That is, the model is specified as the ideal mathematical case, without reference to the limitations that an actual computer implementation will have to overcome. This is done so that there is a relatively uncomplicated reference model that all stakeholders can agree on and understand. Examples of theoretical models include solving the eigenvalue problem ($Ax = \lambda x$), or integrating a function $f(x)$ over the limits $a$ to $b$ ($\int_a^b f(x)dx$).

The section on input assumptions emphasizes the importance of assumptions to scientific computing for making the theoretical model something that can be solved. In many cases if no constraints are placed on the theoretical model, then it cannot be solved numerically for all possible inputs. For instance, the eigenvalue problem cannot be solved if any of the entries in $A$ exceeds the maximum machine representable number and an integration problem can have large errors if there is a singularity nearby.

The variabilities between NFRs are not explicitly included in the CA template because it is implicitly assumed that for all scientific software variability exists in the priority of NFRs and in the degree to which the NFRs are satisfied. Rather than document the NFRs in the CA this is left to the SRS, where the tradeoffs can be made explicit for a given program family member. Moreover, the SRS is a suitable place for describing how the NFRs will be validated through the software validation strategy.

For each of the variabilities in the CA there is a parameter of variation and a binding time. The parameter of variation specifies the type of the possible values for the variability. The binding time is the time in the software lifecycle when the variability is fixed. The binding time could be during specification of the requirements (specification time), or during building of the system (compile time), or during execution of the system (run time). It is possible to have a mixture of binding times. For instance, a parameter of variation could have a binding time of "specification or build" to represent that the parameter could be set at specification time, or it could be postponed until the given family member is built. The choice of postponing the decision until the build could be associated with the presence of a domain specific language (DSL).

The traceability matrix takes the role of showing the relationship between the terminology, goals, theories and assumptions. This matrix is important so that change can be tracked through the CA document. A traceability matrix with the same purpose was introduced in [16, 17]. The matrix provides the serves the same purpose as the related commonality field that is suggested for each variability in the Weiss approach [18].

### 3.2. Requirements Analysis

Figure 2 shows the proposed requirements template for the SRS. This template is a modified version of that presented in [16, 17], which in turn was mainly based on the IEEE Standard 830 [6], except for the subsection "Nonfunctional Requirements" and the section "Other System Issues," which were inspired by the Volere Requirements Specification Template [12]. The section "General System Description" comes from both of the previously mentioned sources. The template also introduces a systematic approach to manage changes in a scientific computing SRS through a newly defined traceability matrix, which is documented in SRS Section 8.

The sections with the same names between the CA and the SRS represent sections that are refined by the SRS family members. For instance, the "General System Description" is shared between the two documents. In each of the family members of the SRS a specific entry is determined for each of the potential system descriptions given in the CA. For instance, in the CA the general description is for potential system contexts, user characteristics and constraints, while in the SRS the word "potential" has been removed because the scope and focus are now on an actual software system. The commonality section of the CA is shared by all SRS documents, while the variability section provides requirements that vary between family members. For each variability in the SRS a particular parameter of variation must be set, along with its binding time.

The NFRs section specifies system requirements that consider the quality and behaviour of the system as a whole. This section is separate from the functional requirements to facilitate the potential independent change of these two portions of the SRS. Several of the sections here are borrowed from the Volere template [12], such as sections for look and

1. Reference Material: a) Table of Contents b) Table of Symbols c) Abbreviations and Acronyms
2. Introduction: a) Purpose of the Document b) Scope of the Software Product c) Organization of the Document
3. General System Description: a) System Context b) User Characteristics c) System Constraints
4. Specific System Description: i) Background Overview, ii) Terminology Definition, iii) Physical System Description, iv) Goal Statements v) Theoretical Models, vi) Assumptions, vii) Data Constraints, viii) System Behaviour
5. Non-functional Requirements: i) Accuracy of Input Data, ii) Sensitivity of the Model, iii) Tolerance of Solution, iv) Look and Feel Requirements, v) Usability Requirements, vi) Performance Requirements, vii) Maintainability Requirements, viii) Portability Requirements, ix) Security Requirements
6. Solution Validation Strategies,
7. Other System Issues: a) Open Issues b) Off-the-Shelf Solutions c) New Problems d) Waiting Room
8. Traceability Matrix
9. List of Possible Changes in the Requirements
10. Values of Auxiliary Constants
11. References

**Figure 2. Proposed Requirements Template**

feel, usability, performance, maintainability, portability and security requirements. The new template has to address the challenge of providing validatable and useful requirements. As mentioned in Section 2.3 when NFRs are phrased as absolute quantitative measures, so as to make the requirements validatable, the problem is introduced that the requirements may be unrealistic. In many cases the absolute performance of the requirement is not important, it the relative comparison to other software products that is important. Validatable requirements can be stated as relative measures of how the general purpose tool compares to other program family members, with respect to the performance of a functionality that they share. For instance, a new software tool for solving linear system of equations can be compared to the accuracy of Matlab. This can be done by identifying benchmark test problems that will be run on the competing software. Running test problems allows for a posteriori description of the software behaviour rather than a priori specification of nonvalidatable NFRs, as discussed in Section 2.3.

The other challenge for NFRs is to capture their relative importance, given that tradeoffs typically exist such that improving one NFR means that another will suffer. For instance, it is difficult to have software that is fast, accurate, portable and maintainable, all at the same time. Depending on the software context, one or more of these NFRs may have a higher priority than the others. When reasoning about all of the NFRs at once, it can be difficult to identify the priorities. The field of decision analysis provides advice on how best to quantify the relative importance of the various NFRs because decision makers face the same challenge when defining rational criteria to judge competing goals and options. To assist with decision making the concept of utility has been introduced to allow ranking of competing alternatives so that the one with the highest utility can be selected. However, the concept of utility can sometimes be challenging to adopt because many competing attributes can contribute to the utility, but the relative importance of the different attributes may be difficult to determine as they do not always have a common basis. For instance, if one is choosing between different transportation alternatives there will be contributions to the utility from both cost and environmental impact. Although the utility of the cost can be easily measured in monetary units, the choice is unclear as to what constitutes appropriate and compatible utility units for measuring environmental impact.

One approach that has been successfully applied to addressing the challenge of comparing attributes, especially attributes that are measured in different units, is the Analytic Hierarchy Process (AHP) [13], since it does not require explicit quantification of utility. Instead of utility, AHP uses ratio scales to assess the relative priorities between various goals and criteria. AHP reduces the challenge of determining priorities to a series of pair-wise comparisons between attributes. These pair-wise comparisons are much easier to reason about than trying to tackle the entire problem all at once. Some example values that can be used for the ranking scale are as follows: 1 for equal importance, 3 for moderately strong importance, 5 for very strong importance and 9 for extreme importance. Given the success AHP has had in decision analysis, this is the approach adopted in this paper to rank the relative priority of the NFRs. An example is provided in Section 4.2 to show how a matrix, along with the associated calculations, can be used to determine the priorities.

As observed in Section 2, it is difficult to validate scientific computing software. The purpose of the SRS section on solution validation strategies is to capture the experts insight on how to validate the software. Three potential evaluation strategies are:

- Solve the problem by different techniques; for in-

stance, an ordinary differential equation can be solved using several different algorithms, such as the Runge Kutta and the predictor-corrector method, and the results compared.

- Test cases can be built where the answer is known. Although in general the answer to a scientific computing problem may be unknown, problems can be constructed where the solution is assumed and then the problem that leads to this solution is formulated. One example of this approach is the Method of Manufactured Solutions (MMS) [11].

- Partially validate the problem by validating simpler subsets of it for which the solution is known.

## 4. Example of a Linear Solver

To help illustrate the methodology described above, this section presents excerpts from a CA and SRS for a linear solver. Besides presenting the commonalities and variabilities, the example also includes a sample validation strategy and an example application of AHP.

### 4.1. Commonality Analysis

The summary of the CA document includes information on the terminology, the goal statement, the theoretical model, and the variabilities, such as the input assumptions.

#### Commonalities

The commonalities section for a linear solver includes the terminology summarized in Table 1. Even at this stage in the documentation decisions are being made, as the terminology clearly rules out the case of a non-square matrix, or of a matrix with complex, as opposed to real, entries.

For the linear solver example, there is only one goal (**G1**), as follows: "**G1**: Given a system of $n$ linear equations represented by matrix $A$ and column vector $b$, return $x$ such that $Ax = b$, if possible."

The theory section (**T1**) of the CA summarizes the ideal mathematical model for solving a system of linear equations. The theory states that for a given square matrix $A$ and column vector $b$, the possible solutions for $x$ are as follows:

1. A unique solution $x = A^{-1}b$, if $A$ is nonsingular
2. An infinite number of solutions if $A$ is singular and $b \in span(A)$
3. No solution if $A$ is singular and $b \notin span(A)$

| | |
|---|---|
| $n : \mathbb{N}$ | number of linear equations and the number of unknowns |
| $A : \mathbb{R}^{n \times n}$ | $n \times n$ real matrix |
| $x : \mathbb{R}^{n \times 1}$ | $n \times 1$ real column vector |
| $b : \mathbb{R}^{n \times 1}$ | $n \times 1$ real column vector |
| $I : \mathbb{R}^{n \times n}$ | an $n \times n$ matrix where all entries are 0, except for the diagonal entries, which are 1 |
| $A^{-1} : \mathbb{R}^{n \times n}$ | the inverse matrix, with the property that $A^{-1}A = I$ |
| $\|\|v\|\|$ | the norm (estimate of magnitude) of vector $v$ |
| residual | $\|\|b - Ax\|\|$ |
| singular | matrix $A$ is singular if $A^{-1}$ does not exist |

**Table 1. Terminology for a Linear Solver**

#### Variabilities

The variabilities and associated parameters of variation for a linear solver are best summarized in tables, such as Table 2, which shows the variabilities associated with the input assumptions. The parameters of variation column lists the valid type for the variability. Binding times are not shown in the table because for the variabilities in this example all of the binding times (specification, build or run time) are valid options. If a smaller program family is desired, then it is possible to restrict the scope to provide fewer binding time options. For instance, all of the variabilities could require specification time binding.

The presentation of the variabilities in Table 2 highlights important features of the input assumptions for the program family. For instance, the table separates the symmetry variability from the matrix structure variability to stress the importance of symmetry and to make it clear that symmetry is independent of the other structures. For instance, a matrix can be both banded and symmetric. The table also explicitly shows that the decision on the allowed size of the system of equations has to be documented. This is important because the design of software where the maximum size of the system is 10 equations is much different from the design for software where the maximum size is 100,000 equations. Table 2 assists in providing information that can potentially aid in producing accurate and efficient algorithms. For instance, if the entries in $A$ and $b$ are known to exist in a small subset of $\mathbb{R}$ then it may be possible to prove some accuracy properties that are usually too difficult to prove. When it comes to the variability of the set of possible values for the entries in $A$ and $b$ the most common choice will be $\mathbb{F}$, where $\mathbb{F}$ is the set of floating point numbers. The floating point numbers will often meet the IEEE standard for single or double precision numbers. Other choices for the subset

| Variability | Parameter of Variation |
|---|---|
| Allowed structure of $A$ | Set of { full, sparse, banded, tridiagonal, block triangular, block structured, diagonal, upper triangular, lower triangular, Hessenberg } |
| Allowed definiteness for $A$ | Set of { not definite, positive definite, positive semi-definite, negative definite, negative semi-definite } |
| Allowed class of $A$ | Set of { diagonally dominant, Toeplitz, Vandermonde } |
| Symmetry assumed? | boolean |
| Possible values for $n$ | set of $\mathbb{N}$ |
| Possible entries in $A$ | set of $\mathbb{R}$ |
| Possible entries in $b$ | set of $\mathbb{R}$ |
| Source of input | Set of { from a file, through the user interface, passed in memory } |
| Encoding of input | Set of {binary, text } |
| Format of input $A$ | Set of {arbitrary, by row, by column, by diagonal } |
| Format of input $b$ | Set of {arbitrary, ordered } |

**Table 2. Variabilities for Input Assumptions**

of $\mathbb{R}$ include fixed point numbers and multiprecision numbers.

Besides the input variabilities, there are also tables for the calculation and output variabilities, corresponding to the other variability subsections of the CA template (Figure 1). The calculation variabilities in Table 3 highlight the fact that in some cases the analyst will be confident that the input data will be correct and that errors will not occur during the calculations. In these cases the parameters of variation for checking the input and for exception generating can both be set to false. With respect to the output variabilities in Table 4 the destination of the output depends on the context. For instance, in a program like Matlab the destination for

| Variability | Parameter of Variation |
|---|---|
| Check input? | boolean (false if the input is assumed to satisfy the input assumptions) |
| Exceptions generated? | boolean (false if the goal is non-stop arithmetic) |
| Norm used for residual | Set of {1-norm, 2-norm, $\infty$-norm } |

**Table 3. Variabilities for Calculation**

| Variability | Parameter of Variation |
|---|---|
| Destination for output $x$ | Set of { to a file, to the screen, to memory } |
| Encoding of output $x$ | Set of {binary, text } |
| Format of output $x$ | Set of {arbitrary, ordered } |
| Output residual | boolean (true if the program returns the residual) |
| Possible entries in $x$ | set of $\mathbb{R}$ |

**Table 4. Variabilities for Output**

the output is memory, but the system also allows output to the screen and to a file. In an embedded system, on the other hand, this variability would likely be restricted to only placing the output in memory.

### 4.2. Requirements Analysis

The SRS needs to specify the expected characteristics of the $A$ matrix and the $b$ vector. For instance, it is difficult to write code to solve any system of equations $Ax = b$, but this job becomes easier if $A$ is known to have special characteristics, such as being symmetric positive definite. Therefore, the SRS should clearly show the assumptions restricting the input data. This is done by selecting the appropriate parameters of variation from the CA document.

The use of AHP to prioritize the NFRs can be illustrated through some examples. In the first example the requirements are for an embedded real-time system for digital signal processing. This system is assumed to have a small size for $n$, say less than 10, and the matrix is assumed to be Toeplitz. For a real time system speed is more important than accuracy and both speed and accuracy are more important than portability, since the assumption is that the hardware will not be changed. Table 5 provides example values for the pair-wise comparison between the NFRs of accuracy, speed and portability. The numbers show the relative importance of speed over accuracy and portability. The priority is calculated by adding up the values in each row and dividing this value by the sum of all the priorities in the matrix. For instance, the priority for speed is calculated as $(1+3+5)/(1+3+5+1/3+1+3+1/5+1/3+1) = 0.64$.

In a second example the relative importance of the NFRs is changed because the context is now assumed to be a linear solver for educational software to teach structural mechanics. In this type of problem the matrices will symmetric, positive definite and of medium size, say $10 \leq n \leq 1000$. In this case portability will be important because there is an assumed desire to run the software on Windows, Mac, Unix and Linux computers. Given this context the pair-wise com-

| | Speed | Accuracy | Portability | Priority |
|---|---|---|---|---|
| **Speed** | 1 | 3 | 5 | 0.64 |
| **Accuracy** | 1/3 | 1 | 3 | 0.26 |
| **Portability** | 1/5 | 1/3 | 1 | 0.11 |

**Table 5. Matrix of Pair-wise Comparisons Between NFRs**

parisons between NFRs could again be performed, but the results would now have the greatest priority value assigned to portability.

With respect to solution validation strategies, one potential approach for a linear solver is to build test cases by using known solutions. For general $A$ and $b$ the value of $x$ is unknown a priori. However, if $A$ is assumed and $x$ is assumed then $b$ can be calculated using matrix multiplication. This calculated value of $b$ can then be used with $A$ in the linear solver and a new value of $x^*$ can be calculated. The solved value of $x^*$ can be compared to the previously assumed value of $x$ to measure the error. Another solution validation strategy is to take the test suite of known solution problems and run them on Matlab and compare the speed and accuracy of these solutions to those found by other program family members. The test cases allow for a description of how the program performs with respect to the NFRs. For instance, test cases can be used to experiment to determine how the accuracy changes with changes in the condition number of the matrix.

## 5. Connection with Design

The typical next step in the software development process is to move to the design stage. The CA and the SRS can greatly facilitate the transition to design. In many cases in scientific computing several software packages will already exist within a given program family. Ideally the development process should facilitate reuse and the capture of existing knowledge and software implementations, as discussed in Section 2.4. If the existing packages are summarized by listing the values for their parameters of variation, then it should be possible to at least partly match the requirements with an existing implementation. In some cases the parameters of variation for the variabilities will directly match, or they may match except for the binding times. In the case where the desired binding time is specification time, but the implementation has a run time binding, then it may be possible to write a simple driver to obtain the desired specification time bound family member.

If several existing implementations are identified as alternatives because they all match the functional requirements documented in the CA and SRS, then the NFRs can be used to pick between the alternatives. In this case one can use methodologies from decision analysis to pick the best design. AHP can again be used to compare the existing design solutions against one another, so as to provide a score for each product. For each NFR a pair-wise comparison can be made between each of the competing alternatives so that a matrix can be constructed, in a manner similar to the approach shown previously in Table 5. The contribution of each NFR to the score for each design alternative is found by multiplying the contribution of each alternative to the given NFR with the corresponding priority of that NFR, where the priority was previously documented in the SRS. The score for each NFR is summed for each design alternative and the alternative with the highest overall score is considered to be the best design choice.

## 6. Concluding Remarks

This paper presents a new methodology for documenting the requirements for general purpose scientific computing software. The new methodology addresses the challenge of writing validatable requirements by including solution validation strategies in the SRS. These solution validation strategies can include listing system tests. Another challenge for scientific computing is writing abstract requirements that realistically consider the later implementation realities. This challenge is addressed in the new methodology by systematically refining the requirements from goal statement, to theoretical model to assumptions about the input data. NFRs can also be problematic for scientific computing because absolute quantitative requirements can be unrealistic and because it is difficult to understand the complex tradeoffs allowed between the NFRs. One idea presented in this paper for addressing these difficulties consists of phrasing the requirements as a relative comparison between program family members. A second idea suggests systematically assessing the priority of the various NFRs by using pair-wise comparisons and the AHP. The final challenge addressed in this paper is how to capture and use the existing wealth of scientific computing knowledge and software. The CA and SRS themselves provide an approach for summarizing and documenting knowledge for future reuse. Moreover, the CA allows one to summarize the currently available software to see what problems have been addressed by the existing general purpose tools and what problems still need to be solved.

In the future it is hoped that the methodology proposed in this paper will be used to improve the quality of scientific software. To succeed in this direction it will be necessary to apply to the methodology to general purpose scientific computing problems that have greater complexity than that of the linear solver example. Some candidate systems include mesh generators, ordinary differential equation solvers and finite element analysis programs. Additional work to vali-

date the proposed approach will be necessary, including empirical studies to quantify the advantages and disadvantages of the new methodology.

## Acknowledgements

The financial support of the Natural Sciences and Engineering Research Council (NSERC) is gratefully acknowledged, as is the many helpful conversations the author has had with the members of the Interval Subroutine Library (ISL) research group.

## References

[1] M. Ardis and D. M. Weiss. Defining families: The commonality analysis. In *Proceedings of the Nineteenth International Conference on Software Engineering*. ACM, Inc., 1997.

[2] P. Clements and L. M. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[3] D. A. Cuka and D. M. Weiss. Specifying executable commands: An example of fast domain engineering. *Submitted to IEEE Transactions on Software Engineering*, pages 1 – 12, 1997.

[4] E. W. Dijkstra. *Structured Programming*, chapter Notes on Structured Programming. Academic Press, London, 1972.

[5] C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2nd edition, 2002.

[6] IEEE. *Recommended Practice for Software Requirements Specifications, IEEE Std. 830*. IEEE, 1998.

[7] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[8] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, pages 128–138, March 1979.

[9] D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April-June 1991.

[10] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, 2005.

[11] P. J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico, 1998.

[12] S. Robertson and J. Robertson. *Mastering the Requirements Process*, chapter Volere Requirements Specification Template, pages 353–391. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, 1999.

[13] T. L. Saaty. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. McGraw-Hill Publishing Company, New York, New York, 1980.

[14] W. S. Smith and C.-H. Chen. Commonality analysis for mesh generating systems. Technical Report CAS-04-10-SS, McMaster University, Department of Computing and Software, 2004.

[15] W. S. Smith and C.-H. Chen. Commonality and requirements analysis for mesh generating software. In F. Maurer and G. Ruhe, editors, *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, pages 384–387, Banff, Alberta, 2004.

[16] W. S. Smith and L. Lai. A new requirements template for scientific computing. In J. Ralyté, P. Àgerfalk, and N. Kraiem, editors, *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, pages 107–121, Paris, France, 2005. In conjunction with 13th IEEE International Requirements Engineering Conference.

[17] W. S. Smith, L. Lai, and R. Khedri. Requirements analysis for engineering computation. In R. Muhanna and R. Mullen, editors, *Proceedings of the NSF Workshop on Reliable Engineering Computing*, pages 29–51, Savannah, Georgia, 2004.

[18] D. Weiss and C. Lai. *Software Product Line Engineering*. Addison-Wesley, 1999.

[19] D. M. Weiss. Defining families: The commonality analysis. *Submitted to IEEE Transactions on Software Engineering*, 1997.

[20] D. M. Weiss. Commonality analysis: A systematic process for defining families. *Lecture Notes in Computer Science*, 1429:214 – 222, 1998.

[21] J. H. Wilkinson. Modern error analysis. *SIAM Review*, 13:548–568, 1971.