Document Driven Certification of Computational Science and Engineering Software

Spencer Smith
McMaster University
Computing and Software Department
Hamilton, Ontario, Canada
smiths@mcmaster.ca

Nirmitha Koothoor
McMaster University
Computing and Software Department
Hamilton, Ontario, Canada
koothon@mcmaster.ca

Nedialko Nedialkov
McMaster University
Computing and Software Department
Hamilton, Ontario, Canada
nedialk@mcmaster.ca

ABSTRACT

This paper presents a documentation and development methodology to facilitate the certification of Computational Science and Engineering (CSE) software that is produced by professional end user developers to solve mathematical models of physical systems. To study the problems faced during quality assurance and certification activities, a case study was performed on legacy software used by a nuclear power generating company for safety analysis in a nuclear reactor. Although no errors were uncovered in the code, the documentation still needed significant updating for certification, since its was incomplete and inconsistent. During the case study, 27 issues were found with the documentation. This work proposes improvements to the case study software and other CSE software via a new template for the Software Requirements Specification (SRS) that clearly and sufficiently states the requirements, while satisfying the desired qualities for a good SRS. For developing the design and implementation, this paper suggests Literate Programming (LP) as an alternative to traditional structured programming. Literate Programming documents the numerical algorithms and the logic behind the development and the code together in the same document, the Literate Programmer's Manual (LPM). The LPM is developed in connection with the SRS. The explicit traceability between the theory, numerical algorithms and implementation (code), facilitates completeness and consistency, and simplifies the process of verification and the associated certification.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/ Specifications; G.4 [**Mathematical Software**]: Certification and testing

Keywords

computational science and engineering, software engineering, literate programming, software requirements specification, document driven design

SE-HPCCSE November 17 - 21 2013, Denver, CO USA

http://dx.doi.org/10.1145/2532352.2532353.

1. INTRODUCTION

The goal of the present work is to facilitate the certification of the Computational Science and Engineering (CSE) software that is produced by professional end user developers to solve mathematical models of physical systems. Professional end user developers [18] work within a highly technical, knowledge-rich domain; they create software as a means to further their work. The software may be developed by an individual, or a group. Its lifetime may be short, or it may span decades. The mathematical models that the professional end user developers build include such components as governing ordinary or partial differential equations, boundary conditions, initial conditions, and constitutive equations. The theory behind the models could come from domains such as physics, chemistry, biology, or engineering. The developed software will use numerical techniques to solve or simulate the mathematical models. Typically, numerical techniques will be used to solve systems of linear equations, to solve ordinary differential, or partial differential equations, to find the zeros of a function, to calculate numerical integrations, etc. The algorithms could be designed for serial processing or for high performance parallel computing. The numerical algorithm is an important choice for the professional end user developers, but the scope of the current work is software projects that encompass both the numerical algorithm and the previously mentioned mathematical models of physical systems. The discussion and recommended methodology would be different in the case of software implementing a generic numerical algorithm.

Certification implies an official recognition by an authority or regulatory body that a certain standard has been met. This step is important for industries where the work has a significant impact on safety, such as in the areas of nuclear power, aerospace and medicine. The point of certification is to ascertain whether the product for which a certificate is being sought has appropriate characteristics [11, p. 91]. This implies asserting facts that determine whether the characteristics of a given product are appropriate. Hence, a certification process should be a measurement-based activity, and the facts should be measurable entities. The goal for software certification should be to: "...systematically determine, based on the principles of science, engineering and measurement theory, whether a software product satisfies accepted, well-defined and measurable criteria" [3, p. 12]. Regulators

will not typically determine whether the software satisfies the stated criteria themselves, rather they will look for proof that proper Quality Assurance (QA) activities have been completed by the software development team. This necessitates a systematic and rigorous methodology that includes documentation that will both improve the software quality and that can later be used to convince regulators that the software is fit for its intended use.

The task of providing adequate documentation for QA and certification efforts for CSE software is not an easy one. Several potential challenges must be dealt with, as described below.

- The underlying mathematical model of the physical system will continue to change as the model is refined via modifications of the underlying assumptions. The development of CSE software often starts with a vague understanding of the model, with the final model evolving through a series of trial runs, modifications and extensions [19].

- Even when the mathematical model is stable, the numerical techniques used to simulate/solve the model will continue to change. The change will be driven both by a need to correct errors and to improve performance. As the emphasis on performance increases, which is the case for high performance computing, the connection between the original model and the developed computer code can become more difficult to see.

- With the potential demands of recertification, future modification of the original theory and code could be delayed or cancelled. A proper certification exercise can consume significant time and resources. If any future changes require recertification, then the recertification process must be significantly easier and cheaper than the first certification exercise, or the recertification is unlikely to happen.

To develop, test and justify the documentation and development methodology proposed in this paper, we conducted a case study with existing software where QA, and the associated documentation, were important considerations. The case study uses legacy nuclear safety analysis software provided by a power generation company. For the purpose of this paper, the software under study is referred to by the name FP, where FP performs thermal analysis of a single fuelpin in a nuclear reactor. FP was used to simulate simplified reactor physics, fuel management calculations and simulations for reliability and safety studies at nuclear power generating stations. The FP software was provided to us with source code and a theory manual, which includes the requirements, numerical algorithms, assumptions, constraints and the mathematical model.

Our approach was to redo the thermal analysis portion of the original FP code using modern software engineering techniques. By redoing the previous work, we are able to judge whether there is room for improvement and then propose a new and improved process. The design and development of the new documentation was done to be consistent with clause 11.2 of N286.7, where N286.7 forms the standard set for the quality assurance of analytical, scientific, and design computer programs for nuclear power plants [1]. Although the conclusions from this paper are given based on the case study, the case study is considered representative of other CSE software.

2. BACKGROUND

This section is divided into three subsections. The first subsection outlines the desirable qualities for documentation that facilitates certification of CSE software. The following two subsections detail the background information on proposed approaches used to achieve the desired qualities: Software Requirements Specifications (SRS) and Literate Programming (LP).

2.1 Desirable Qualities for Documentation

To address the challenges for adequate documentation for QA and certification efforts for CSE software, as described in Section 1, the qualities described below need to be a priority. All but the final quality listed (abstraction), are adapted from the IEEE recommended practise for producing good software requirements [4].

- *Completeness*: Documentation is said to be complete when all the requirements of the software are detailed. That is, each goal, functionality, attribute, design constraint, value, data, model, symbol, term (with its unit of measurement if applicable), abbreviation, acronym, assumption and performance requirement of the software is defined. The software's response to all classes of inputs, both valid and invalid and for both desired and undesired events, also needs to be specified.

- *Consistency*: Documentation is said to be consistent when no subset of individual statements are in conflict with each other. That is, a specification of an item made at one place in the document should not contradict the specification of the same item at another location.

- *Modifiability*: The documentation should be developed in such a way that it is easily modifiable so that likely future changes do not destroy the structure of the document. Also it should be easy to reflect the change, wherever needed in the document to maintain consistency, traceability and completeness. For documentation to be modifiable, its format must be structured in a way that repetition is avoided and cross-referencing is employed.

- *Traceability*: Documentation should be traceable, as this facilitates maintenance and review. If a change is made to the design or code of the software, then all the documentation relating to those segments have to be modified. This property is also important for recertification.

- *Unambiguity*: Documentation is said to be unambiguous only when every requirement's specification has a unique interpretation. The documentation should be unambiguous to all audiences, including developers, users and reviewers.

- *Correctness*: There is no direct tool or method for measuring correctness. One way of building confidence in correctness is by reviewing to ensure that each requirement stated is one that the stakeholders and experts desire. By maintaining traceability, consistency

and unambiguity, we can reduce the occurrence of errors and make the goal of reviewing for correctness easier.

- *Verifiability*: Every requirement in the documentation must be the one fulfilled by the implemented software. Therefore all the requirements should be clear, unambiguous and testable, so that a person or a machine can verify whether the software product meets the requirements.

- *Abstract*: Documented requirements are said to be abstract if they state what the software must do and the properties it must possess, but do not speak about how these are to be achieved. For example, a requirement can specify that an Ordinary Differential Equation (ODE) must be solved, but it should not mention that Euler's method should be used to solve the ODE. How to accomplish the requirement is a design decision, which is documented during the design phase.

2.2 Software Requirements Specification

The first portion of the proposed documentation specifies the requirements. Requirements record all the expected characteristics and behaviour of the system. The document that records the requirements is called the Software Requirements Specification (SRS). This document describes the functionalities, expected performance, goals, context, design constraints, external interfaces and other quality attributes of the software [4].

An SRS provides many advantages during software development [15, 22]. For instance, an SRS acts as an official statement of the system requirements for the developers, stakeholders and the end-users, and creating the SRS allows for earlier identification of errors and omissions. Fixing errors at the early stages of development is much cheaper than finding and fixing them later. In addition, the quality of software cannot be properly assessed without a standard against which it should be judged. A further advantage of an SRS is that it aids in making decisions regarding design and coding of the software by serving as a starting point for the software design phase. Moreover, the SRS aids the software lifecycle by facilitating incremental development. That is, a new version of the software can inherit features of the previous version to upgrade the system by improving the features. This last advantage is important for CSE software, where the changes are frequent as developers explore the problem domain.

To write an SRS, a common approach that is adopted is the use of a requirements template, which provides guidelines for documenting the requirements. It uses a framework that suggests an order for filling in the details. There are many advantages of using a template in writing an SRS [21]. One advantage is that a template increases the adequacy of an SRS by providing a predefined organization that aids in achieving completeness and modifiability. Moreover, a template with a well organized format acts like a checklist for the writer, thus reducing the chances of missing information. Another benefit is that an SRS facilitates the communication between the stakeholders, developers and maintenance staff. A template aids in achieving information hiding through specific guidelines on the appropriate level of abstraction and makes the document more understandable by showing the connections between different sections.

There are several existing templates that have been designed for business and real time applications. These templates contain good suggestions on how to avoid complications and how to develop an SRS to achieve qualities of good documentation [2, 4, 13]. There is no universally accepted template for an SRS. The current research adapts the SRS template developed for CSE software in [21], since this template most closely matches the needs of the FP software.

2.3 Literate Programming

Literate Programming (LP) was introduced as a programming methodology by D. Knuth [6]. Its essence can be captured as in [7, pg. 99]: "...instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do," and introducing concepts "...in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other."

When developing a literate program, we break down an algorithm into smaller, easy-to-understand parts, and explain, document, and implement each of them in an order that is more natural for human comprehension, versus an order that is suitable for compilation. In a literate program, documentation and code are in one source. The program is an interconnected "web" of pieces of code, referred to as *sections* [6, 7] or *chunks* [5, 17], which can be presented in any sequence. They are assembled into a compilable program in a *tangle* process, which extracts the source code from the LP source. Extracting the documentation so that it may be properly typeset [7, 8] is called a *weaving* process. Developing a literate program becomes a task much closer to writing an article or a book: we present the program in an order that follows our thought process and strive to explain our ideas clearly in a document that should be of publishable quality. This also ends up producing much higher quality code, which is furthermore impeccably documented.

A comprehensive collection of resources on LP, including extensive bibliography is [10]; an annotated bibliography of LP, until 1991, is [20]. The two most significant examples of LP applied to CSE are [14, 16].

In [12], LP is used to facilitate the verification of a network security device. The authors propose in [12] that LP techniques are used to "document the entire assurance argument." According to their experience, rigorous arguments, including machine-generated proofs of theory and implementation, "did not significantly improve the certifier's confidence" in their validity. One of the main reasons is that specifications and proofs were documented in a manner to facilitate acceptance by mechanical tools rather than humans. Essentially, the authors conclude that LP greatly facilitates the development of assurance arguments that would be more naturally understood by (human) certifiers than descriptions of machine-generated proofs. This idea is also what motivates the current work that proposes LP to improve QA and facilitate certification.

3. CASE STUDY

In this section, we give a brief overview of the FP software and then present excerpts from the SRS and LP documents. The full SRS and LPM (Literate Programmer's Manual) for this case study are presented in [9]. The purpose of FP is to perform thermal analysis of a single fuelpin in the reactor. Each fuelpin includes the following elements, as presented

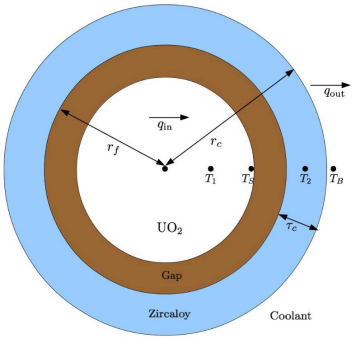


Figure 1: Fuel pellet representation

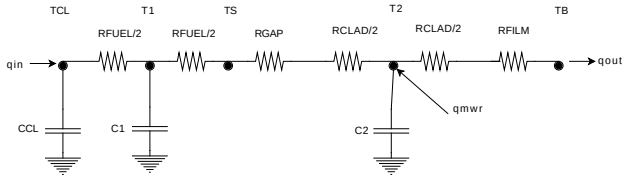


Figure 2: Electrical Circuit Analogue

in Figure 1: a fuel pellet made of uranium dioxide (UO_2), the clad material zircaloy covering the pellet, and coolant surrounding the clad material.

The software is used for running safety analysis cases. The analysis of one fuelpin by FP is used to give insight into the use of multiple pins. The goals of FP are as follows:

G1: Given fuel power versus time as input, predict transient reactor fuel and clad temperatures.

G2: Given the neutron flux versus time as input, predict transient reactor fuel and clad temperatures.

G3: Given the reactivity transient as input, predict transient reactor fuel and clad temperatures.

G4: Given the trip set points, number of trips to initiate shutdown, shutdown reactivity transient as inputs, simulate reactor trip and shutdown.

FP uses point neutron kinetics, decay heat equations, lumped parameter fuel modelling techniques, temperature dependent thermodynamic properties, a metal water reaction model, fuel stored energy, integrated fuel power calculations and trip parameter modelling to do the thermal analysis. The electrical circuit analogue of fuelpin representation is given by Figure 2. A summary of the variables for Figure 1 and Figure 2 is given below, with their interpretation.

T_S	surface temperature (°C)
T_1	average fuel temperature (°C)
T_2	average clad temperature (°C)
T_B	coolant temperature (°C)
T_{CL}	centreline temperature (°C)
r_c	clad radius (m)
r_f	fuel radius (m)
τ_c	clad thickness (m)
q_{in}	input heat ($\frac{\text{kW}}{\text{m}^2\text{°C}}$)
q_{out}	output heat ($\frac{\text{kW}}{\text{m}^2\text{°C}}$)
q'_{MWR}	metal water reaction heat ($\frac{\text{kW}}{\text{m}}$)
R_{FUEL}	thermal resistance of fuel ($\frac{\text{m°C}}{\text{kW}}$)
R_{CLAD}	clad resistance ($\frac{\text{m°C}}{\text{kW}}$)
R_{GAP}	gap resistance ($\frac{\text{m°C}}{\text{kW}}$)
R_{FILM}	coolant film resistance ($\frac{\text{m°C}}{\text{kW}}$)
C_1	thermal capacitance of the fuel ($\frac{\text{kWs}}{\text{m°C}}$)
C_2	thermal capacitance of the clad ($\frac{\text{kWs}}{\text{m°C}}$)
C_{CL}	thermal capacitance at the centerline ($\frac{\text{kWs}}{\text{m°C}}$)

3.1 Software Requirements Specification

We borrowed the template developed by [21] for engineering mechanics and adapted it to suit the nuclear physics domain by adding a few new sections. Following [21], the proposed requirements template is systematically developed by decomposing the problem into smaller tasks. That is, we try to achieve the problem goals by considering the theoretical models and then developing the instance models from them to solve the problem. During this refinement from goals to theory to mathematical models, we apply different assumptions, build general definitions and create data definitions. The proposed template aids in documenting all the necessary information, as each section has to be considered, even if it is inappropriate for the given problem. This facilitates the achievement of completeness by providing a checklist for the questions that are to be asked and for the information that is to be filled in. Besides filling in section headings, the template also requires that every equation be either cited, or derived. Furthermore, every symbol, general definition, data definition, and assumption needs to be used at least once.

The most important sections of this template that are used to improve the desired qualities of an SRS are presented below with an introduction to their motivation and content [21].

3.1.1 Goals

Motivation

To collect and document the objectives of a system in the requirements process.

Content

A goal statement should specify the target of the system. The goal must be abstract. That is, it should be a specification indicating what the system is expected to perform, but not the ways of achieving the objective. The goals for FP (G1–G4) are listed at the beginning of Section 3.

3.1.2 Assumptions

Motivation

To record the assumptions that have to be made, or have been made, while developing the software.

Content

An assumption is a specification showing the approximation to be made while solving a problem. We suggest that assumptions are documented with the forward references made

to the data using them. An example assumption is given below:

A9: The spacial effects are neglected in the reactor kinetics formulations.

3.1.3 Theoretical Models

Motivation

To develop an understanding of the theory or principles relevant to the problem [21].

Content

The theoretical models are sets of governing equations or axioms that are used to model the problem described in the problem definition section. Theoretical models give an introduction of the theory. In the context of nuclear physics, the theoretical models can be physical laws (include relevant equations), constitutive equations, etc. Given below is an example of a theoretical model from our case study.

Number	**T1**
Label	**Conservation of energy**
Equation	$-\nabla \mathbf{q} + q''' = \rho C \frac{\partial T}{\partial t}$
Description	The above equation gives the conservation of energy for a time varying heat transfer in a material of specific heat capacity C and density ρ, where $\mathbf{q}$ is the thermal flux vector, q''' is the volumetric heat generation, T is the temperature, ∇ is the gradient operator and t is the time.

The conservation of energy equation is the most important theoretical model in our case study, as it forms the foundation for the derivation of the mathematical models of FP. How the symbolic equation of conservation of energy is used in deriving the instance models for FP is shown in [9]. The theory must be given as abstractly as possible to make it reusable for other problems. The theory will be later refined to instance models by applying assumptions and definitions. For example, in the theoretical model given above, the coordinate system is not assumed. In the current case study, a cylindrical coordinate system is used, but the general notation means that T1 can be used in a different context, say with a Cartesian coordinate system.

3.1.4 General Definitions

Motivation

This is the new section included in this template to gather and document all the necessary data that will be repetitively used in deriving different data definitions.

Content

General definitions constitute the laws and equations that will be used indirectly in developing the mathematical models. That is, general definitions are those that will not directly model the problem, but will be used in deriving the data definitions, which in turn are used to build the instance models. The general definitions are documented using tabular and textual descriptions. An example of a general definition is given below.

Number	**GD1**
Label	**Cylindrical coordinate system**
Equation	$\nabla = \hat{r}\frac{\partial}{\partial r} + \hat{\theta}\frac{1}{r}(\frac{\partial}{\partial \theta}) + \hat{z}\frac{\partial}{\partial z}$ where $\hat{r}$, $\hat{\theta}$ and $\hat{z}$ are unit vectors. In matrix notation, this appears as: $\nabla = \begin{bmatrix} \frac{\partial}{\partial r} \\ \frac{1}{r}\frac{\partial}{\partial \theta} \\ \frac{\partial}{\partial z} \end{bmatrix}$ The divergence ∇A is calculated as: $\nabla \mathrm{A} = \frac{\partial (A_r)}{\partial r} + \frac{1}{r}\frac{\partial A_\theta}{\partial \theta} + \frac{\partial A_z}{\partial z}$
Description	The spatial location in a cylindrical coordinate system is expressed in terms of $\hat{r}$, $\hat{\theta}$, $\hat{z}$. The gradient operator is defined as shown above.
Sources	FP Theory Manual

3.1.5 Data Definitions

Motivation

To collect and organize all physical data needed to solve the problem [21].

Content

All the symbols that are used in developing the mathematical models of the system are defined using a tabular representation. The symbol should be defined with the meaning of the physical data they represent and needs to be given a unique label to support traceability. If any equation is defined in this section, then the derivation of that equation is given under the table. An example of data definition is given below.

Number	**DD3**
Label	**Integrated fuel power**
Symbol	$P_{\mathrm{F,SUM}}$
Units	FPS
SI equiv.	-
Equation	$P_{\mathrm{F,SUM}}(t_i) = \int_0^{t_i} q'_{\mathrm{NFRAC}}(t)dt$
Description	The above equation gives the integrated fuel power at t_i, where q'_{NFRAC} is the relative fuel power and $P_{\mathrm{F,SUM}}(t_i)$ is the integrated fuel power at t_i
Sources	FP Theory Manual

3.1.6 Instanced Model

Motivation

The mathematical model needs to become more concrete before it can be solved using a numerical algorithm implemented in code.

Content

The theoretical model is refined to an instanced model, using the general definitions, data definitions and assumptions. An example instanced model follows.

Number	IM1
Label	Rate of change of average fuel temperature
Equation	$C_1 \frac{dT_1}{dt} = q'_N - \frac{T_1 - T_2}{R_1}$
Description	T_1 - average fuel temperature T_2 - clad temperature R_1 - effective resistance between fuel and clad C_1 - thermal capacitance of the fuel q'_N - linear element power t - time
Sources	FP Manual

3.1.7 Evaluation of SRS

The documentation accompanying the case study software was done by highly qualified domain experts with the goal of fully explaining the theory behind FP for the purpose of quality assurance. As is usual for this kind of documentation, the presentation of the theory was similar to what one would see in a scientific or engineering journal or technical report. Even with an understanding of the importance of the documentation, our development of the new SRS uncovered 27 issues in the previous documentation, ranging from trivial to substantive. Some qualities that were impacted and the corresponding examples are described below.

1. *Incompleteness*: As per the definition of completeness in Section 2.1, every term in the document should be defined. However, the term R_{GAP}, as shown in Figure 2, and used in several equations in the original documentation, was not defined.

2. *Inconsistency*: In some cases the same concept was referenced with different symbols, such as fuel radius, which at times was symbolized by r and at other times by r_0. In other cases, the same symbol was used for different concepts, such as the term h_g being used to represent gap conductance in one instance and the effective heat transfer coefficient between clad and fuel surface in another.

3. *Verifiability problem*: As per the definition of verifiability in Section 2.1, every specification in the document must be the one fulfilled by the software. However, in the original source code, the effective thermal resistance R_1 was solved by an equation that was seemingly different from the equation in the original documentation. This problem was eventually tracked to an inconsistency in the documentation, and not a fault in the code.

4. *Lack of traceability*: One problem regarding traceability was no traceability between the figure representing the electrical circuit analogue (Figure 2) of the fuel pellet and the derivation of R_1. The figure is required at the time of verifying R_1, as it gives an insight into the derivation of the equation, but the figure, although present was not referenced. This problem was compounded by a lack of an explanation of the electric circuit analogy for thermal "circuits." The omission was likely because the domain experts understood the analogy so well. However, for completeness this needs to be explained, so that QA activities can properly judge whether the analogy holds. A lack of traceability leads to a modifiability problem as well, since managing changes requires knowledge of the impact of the changes.

The SRS proposed in this paper is intended to avoid the above problems. To achieve the qualities of a good SRS, the template applies the principle of "separation of concerns" by including different sections so that focus can be on one thing at a time. By dividing the problem into smaller steps and considering each section, it is easier to document all the necessary information completely and correctly. The sections like Theoretical Models, General Definitions, Data Definitions are included before the Instance Models section for the purpose of systematically solving the problem in a hierarchical way. This way of developing the concrete models from abstract ones helps in achieving completeness, consistency, traceability and verifiability in the documentation. The purpose of including these sections is to document all the background information, physical laws, constitutive equations, rules, principles and physical data required to solve the problem. This documentation means that the QA activities can involve asking other domain experts if the stated assumptions, and derivations, are realistic and correct.

To tackle the inconsistency problem, the template includes a section called "Table of Symbols," where all the symbols used in the document are summarized along with their units. To deal with the problem of traceability, and modifiability, we use cross referencing between the components. The template requires the use of a unique label to each component and the development of models in a hierarchical manner.

To solve the problems with completeness and correctness, the template uses the Assumptions, Theoretical Models, General Definitions and Data Definitions sections. These sections collect all the necessary data needed to build the equations, mathematical models and define them using tabular representation, as well as textual description. The mathematical models are built from the theoretical models considering the approximations and using the data definitions. The data definitions are derived from the General Definitions. This way of developing the concrete models from the abstract ones, while maintaining traceability between them, aids in achieving correctness. As the derivations of the equations are also included in these sections, it makes checking correctness easier. Hence the completeness, as well as the correctness, can be achieved at the same time by documenting every equation, assumption, definition and model in the respective sections. Once the requirement is specified completely and correctly, with traceability to its components, the task of verifiability becomes easier.

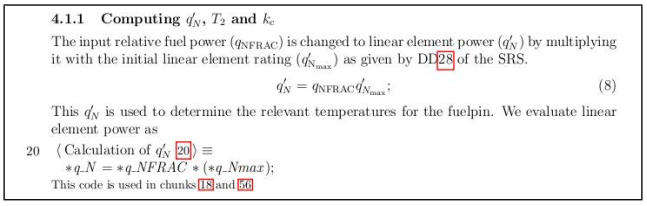
3.2 Literate Programming

Although in standard engineering practice, design and implementation are done in two different phases, we can document both the phases together using Literate Programming (LP). This can be achieved because the source code is developed along with the logic behind it in LP. The document recording the logic behind the numerical algorithm is termed the Literate Programmer's Manual (LPM). The first step in the LPM is the design of each instance model by developing the numerical algorithms required to solve it. Second, the information behind the implementation of the instance model is given. Third, the data definitions required to develop the instance model are implemented as small pieces

of code called chunks. Finally, all the chunks are integrated into one main chunk. This way, all the instance models are designed and implemented as an interconnected web. As for the SRS, the presentation begins with an abstract view that is later refined into a concrete implementation.

3.2.1 Example

While developing the numerical algorithms, the logic and concepts behind the development are given in an order that is best for human understanding. When a concept is introduced, the data definition equation used in it is given with the chunk of implementation, to make verification easier. When the equations of the data definitions and the code implementing them are presented together, the lines of code can be compared to the definition and the correctness of the implementation can be easily checked. An example, showing how the verification of the code with the design is achieved, is given by Figure 3.



4.1.1 Computing q'_N, T_2 and k_c

The input relative fuel power (q_{NFRAC}) is changed to linear element power (q'_N) by multiplying it with the initial linear element rating ($q'_{N_{\max}}$) as given by DD28 of the SRS.

$$q'_N = q_{\text{NFRAC}} q'_{N_{\max}}; \quad (8)$$

This q'_N is used to determine the relevant temperatures for the fuelpin. We evaluate linear element power as

20 ⟨ Calculation of q'_N 20 ⟩ ≡
$*q\_N = *q\_NFRAC * (*q\_Nmax);$
This code is used in chunks 18 and 56

Figure 3: Excerpt from LPM showing the implementation of q'_{NFRAC} in the steady state

While introducing the concepts, references are made to the relevant data definitions, assumptions and auxiliary constants defined in the SRS. Thus the traceability between the concepts being introduced during design and the SRS can be achieved.

3.2.2 Evaluation of LP

Even though LP is a programming technique, the main idea is to make the design and logic behind the code understandable to the human reader. Documenting the design and program flow using LP achieves most of the qualities of good documentation (mentioned in Section 2.1) and at the same time meets the expectations of the N286.7 standard [1] for the design and programmer's manuals. In this section, we discuss the advantages of LP, by showing the steps we have taken to achieve the desired qualities of a good document, as follows:

1. *Completeness:* While developing the LP source file, the main algorithm of the program was divided into smaller parts, which contain explanation, definitions and implementation. As all the theory and numerical algorithms necessary for the implementation are presented before the coding is done, the quality of completeness can be achieved.

2. *Correctness:* As LP is developed in connection with the SRS, checking for correctness becomes easier. Before implementing each term, the definition of the term is taken from the SRS and given again in the LP document, as shown in Figure 3. This way of implementing each term as a chunk in connection with SRS aids in checking whether the program is implementing the right requirement. Confidence in correctness is built by verifying that every line of code either traces back to a description of the numerical algorithm (in the LPM), or to a data definition, or to an instance model, or to an assumption, or to a value from the auxiliary constants table in the SRS.

3. *Consistency:* To improve consistency, the naming conventions of the variables have been given during the design of the algorithm. As the code was developed following the naming conventions, the probability of inconsistencies has been reduced. Consistency is also improved since each term was developed as an individual chunk only once and has been reused wherever necessary, as shown in Figure 3, where chunk 20 is shown to be used in chunks 18 and 56.

4. *Traceability:* For traceability between the components of LP, each equation, definition and table has been labelled and referenced, wherever necessary in the document. In LP, the program is developed as a web of interconnected chunks. LP automatically assigns a number to each chunk. When a chunk is used somewhere in the development of an algorithm, or in the implementation of an instance model, LP automatically generates a hyperlink to the place where the chunk is being used. Also, at the place where the chunk is being called, LP mentions the number of the chunk that gives the code of the definition, so that the reader will know where the chunk is being used, and also where it has come from during the implementation. For traceability between the LPM and the SRS, cross referencing was done between the two documents.

5. *Verifiability:* The quality of verifiability is improved since all the necessary information behind the implementation, like development of numerical algorithms, solution techniques, assumptions and the program flow is given. Moreover, as traceability between the SRS and LP has been maintained, compliance of the design and implementation with requirements can be checked. As documentation of the design and code is made together in the same document, it is sufficient for the verifier to have the SRS and LP to confirm the correctness of the software.

6. *Unambiguity:* As each term or model is developed only once as a chunk and reused wherever necessary, there is no chance of having two different implementations for the same definition. Also, as everything behind the implementation is provided in detail, the chances of having two different interpretations can be reduced through QA reviews.

7. *Modifiability:* As each term is being implemented only once, the task of modification becomes easier. If in the future, the implementation has to be changed, only that chunk consisting of the code has to be modified. As repetition is avoided, consistency will not be affected by the modification. Also, as traceability and consistency are maintained, the modifiability becomes easier. This quality is helpful for recertification efforts.

4. CONCLUDING REMARKS

In many instances the documentation of requirements and design are not being given the importance they deserve in

CSE. The introduction of software engineering methodologies into CSE should improve the situation. The goal of this research is to make the scientists understand the gravity of the problems that arise when software is poorly documented. By the introduction of a new template for the SRS, we hope that people from the scientific field will adopt the practise of documenting requirements, while avoiding problems of inconsistency and incompleteness. Introducing LP for documenting the design and implementation will facilitate maintaining the traceability between the theory, design and code, thus making the verification part easier during QA and certification activities.

Although some of the problems in the original documentation for the case study would likely have been found with any effort spent redoing the documentation, the systematic and rigorous process proposed here is intended to build confidence that the methodology itself improves quality. The proposed SRS template assists in systematically developing the requirements document. The template helps in achieving completeness, as sections of it act as a checklist to the developer and force him or her to fill in the necessary information. As the template is developed following the principle of separation of concerns, each section can be dealt with individually, and the document can be developed in detail by refining from goals to instanced models.

In all software projects, there is a danger of the code and documentation getting out of sync, which seems to have been a problem in the given case study. LP, together with a rigorous change management policy, mitigates this danger. LP develops the code and design in the same document, while maintaining traceability between them, and back to the SRS. As changes are proposed their impact can be determined and assessed. This implies that the cost of recertification can be made reasonable when LP is employed.

5. ACKNOWLEDGMENTS

The financial contributions from the McMaster Centre for Software Certification (McSCert) are gratefully acknowledged, as is the advice provided by Dr. Alan Wassyng and Dr. Jacques Carette, from McMaster University.

6. REFERENCES

[1] CSA. Quality assurance of analytical, scientific, and design computer programs for nuclear power plants. Technical Report N286.7-99, Canadian Standards Association, March 1999.

[2] ESA. ESA software engineering standards, PSS-05-0 issue 2. Technical report, European Space Agency, February 1991.

[3] John Hatcliff, Mats Heimdahl, Mark Lawford, Tom Maibaum, Alan Wassyng, and Fred Wurden. A Software Certification Consortium and its Top 9 Hurdles. *Electronic Notes in Theoretical Computer Science*, 238(4):11–17, 2009.

[4] IEEE. *Recommended practice for Software Requirements Specifications*. IEEE, June 1998.

[5] Andrew Johnson and Brad Johnson. Literate programming using `noweb`. *Linux J., Article No 1*, 1997.

[6] Donald E. Knuth. The WEB system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[7] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. 1992.

[8] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*. Addison-Wesley, Reading, Massachusetts, 1993.

[9] Nirmitha Koothoor. A document driven approach to certifying scientific computing software. Master's thesis, McMaster University, Hamilton, Ontario, Canada, 2013.

[10] Literate programming web site. `http://www.literateprogramming.com`.

[11] Thomas Maibaum and Alan Wassyng. A Product-Focused Approach to Software Certification. *IEEE Computer*, 41(2):91–93, 2008.

[12] Andrew P. Moore, Charles N. Payne, and Jr. Increasing assurance with literate programming techniques. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, pages 187–198, 1996.

[13] NASA. Software requirements DID, SMAP-DID-P200-SW, release 4.3. Technical report, National Aeronautics and Space Agency, 1989.

[14] Nedialko S. Nedialkov. VNODE-LP — a validated solver for initial value problems in ordinary differential equations. Technical Report CAS-06-06-NN, Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4K1, 2006. VNODE-LP is available at `http://www.cas.mcmaster.ca/~nedialk/vnodelp`.

[15] David L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[16] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[17] Joachim Schrod. Typesetting CWEAVE output. CTAN, the Comprehensive TEX Archive Network, 1995.

[18] Judith Segal. End-user software engineering and professional end-user developers. In *Dagstuhl Seminar Proceedings 07081, End-User Software Engineering*, 2007.

[19] Judith Segal and Chris Morris. Developing scientific software. *IEEE Software*, 25(4):18–20, July/August 2008.

[20] Lisa M. C. Smith and Mansur H. Samadzadeh. An annotated bibliography of literate programming. *ACM SIGPLAN Notices*, 26(1):14–20, 1991.

[21] W. Spencer Smith, Lei Lai, and Ridha Khedri. Requirements analysis for engineering computation: A systematic approach for improving software reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation*, 13, 2007.

[22] I. Sommerville and P. Sawyer. *Requirement Engineering: A Good Practice Guide*. John Wiley & Sons Ltd., 1997.