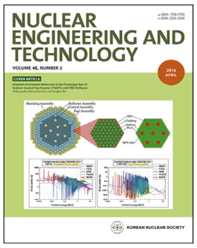
Available online at ScienceDirect

Nuclear Engineering and Technology

journal homepage: www.elsevier.com/locate/net



Original Article

A Document-Driven Method for Certifying Scientific Computing Software for Use in Nuclear Safety Analysis

W. Spencer Smith[*],[1] *and Nirmitha Koothoor*

Computing and Software Department, McMaster University, Hamilton, Ontario L8S 4L7, Canada

ARTICLE INFO

Article history:
Received 12 June 2015
Received in revised form
16 November 2015
Accepted 29 November 2015
Available online 18 December 2015

Keywords:
Literate Programming
Nuclear Safety Analysis
Numerical Simulation
Requirements Specification
Software Engineering
Software Quality

ABSTRACT

This paper presents a documentation and development method to facilitate the certification of scientific computing software used in the safety analysis of nuclear facilities. To study the problems faced during quality assurance and certification activities, a case study was performed on legacy software used for thermal analysis of a fuelpin in a nuclear reactor. Although no errors were uncovered in the code, 27 issues of incompleteness and inconsistency were found with the documentation. This work proposes that software documentation follow a rational process, which includes a software requirements specification following a template that is reusable, maintainable, and understandable. To develop the design and implementation, this paper suggests literate programming as an alternative to traditional structured programming. Literate programming allows for documenting of numerical algorithms and code together in what is termed the literate programmer's manual. This manual is developed with explicit traceability to the software requirements specification. The traceability between the theory, numerical algorithms, and implementation facilitates achieving completeness and consistency, as well as simplifies the process of verification and the associated certification.

1. Introduction

This paper focuses on the certification of scientific computing (SC) software used for safety analysis in the design of nuclear facilities. Although this class of software is not considered as safety-critical, since it does not control the operation of a nuclear reactor or the associated safety systems, high-quality SC software is necessary for designing efficient and safe power plants. Standards and guidelines exist for producing SC software in a nuclear context, such as the Canadian requirements for quality assurance (QA) of scientific computer programs [1–3] and the US Department of Energy (DOE) guidelines for determining the adequacy of software used in safety analysis and design [4]. These publications list documentation that is expected for QA activities, including software requirements, design specification and verification, and

* Corresponding author.
E-mail address: smiths@mcmaster.ca (W.S. Smith).
1 URL: http://www.cas.mcmaster.ca/%7esmiths/

http://dx.doi.org/10.1016/j.net.2015.11.008

validation reports. The standards and guidelines lay out at a high (abstract) level of what needs to be achieved by documentation, but at times they give limited concrete information on how to achieve these requirements. This paper fills in the missing details by proposing a systematic method for writing complete, consistent, and verifiable documentation for SC software used in nuclear safety analysis.

For certification to be successful, the documentation and code should have the qualities of verifiability, validatability, reliability, usability, maintainability, reusability, and reproducibility. With the exception of reproducibility and validatability, these qualities for general software are defined in a report by Ghezzi et al. [5, pp. 18–28]. In a SC context, verification means "solving the equations right" and validatability is "solving the right equations" [6, p. 23]. Reproducibility means being able to rerun the code in the future, possibly through an independent third party, and obtaining identical results [7].

Maintainability is necessary in SC, because change through iteration, experimentation and exploration is inevitable. Models of physical phenomena necessarily evolve over time [8,9], as do the numerical techniques used to simulate the models. QA activities need to take this need for creativity into account without smothering it [6, p. 352]. Maintainability is of practical importance because when changes occur after the initial certification, the recertification process must be significantly easier and cheaper than the first certification exercise, or recertification is unlikely to happen. Similarly, reusability is important for certification, because reuse can save time and money spent on the certification of similar products by reusing trusted components [10]. Fortunately, SC software is well suited for reuse, as program families (sets of programs where there are nontrivial commonalities and predictable variabilities) are frequently encountered in SC [11].

Documentation for nuclear safety software [2,4] follows the typical stages of the waterfall model of software development, as shown in Fig. 1. Given the exploratory nature of SC, developers do not follow this waterfall model [12,13], but this is not a problem for the documentation. As Parnas and Clements [14] point out, the most logical way to present the documentation is to "fake" a rational design process. "Software manufacturers can define their own internal process as long as they can effectively map their products onto the ones that the much simpler, faked process requires" [15]. To keep the scope of the current work manageable, we focus on the three middle stages from Fig. 1: requirements, design, and code implementation.

1. Problem statement.
2. Development plan.
3. Software requirements specification.
4. Design documentation.
5. Code implementation.
6. Verification and validation report.

Fig. 1 – Software documentation following a rational process.

To develop, test, and justify the documentation and development method proposed, we conducted a case study with existing SC software, for which QA and the associated documentation are important considerations. The case study uses legacy nuclear safety analysis software provided by a power generation company. The software under study performs thermal analysis of a single fuelpin in a nuclear reactor by simulating simplified reactor physics and fuel management calculations. In the discussion that follows, the software will be referred to as FP. Along with the source code for FP we also received a theory manual, which includes the requirements, numerical algorithms, assumptions, constraints, and the mathematical model.

Our approach, which was described by Koothoor [16] and Smith et al. [17], was to redo the thermal analysis portion of the original FP code using modern software engineering techniques. By redoing the previous work, we were 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 the Canadian standard for quality assurance of analytical, scientific, and design computer programs for nuclear power plants, N286.7, clause 11.2 [2]. Although the conclusions from this paper are based on the case study, the case study is considered representative of many other SC programs.

Section 2 provides background on the software engineering methods that are employed in the documentation, namely, software requirements specification (SRS) and literate programming (LP). This background section also gives an overview of the FP case study. Section 3 provides examples for the SRS for FP, along with an evaluation of the improvements of the new documentation compared with the old. Section 4 presents the LP excerpts from FP and explains how the literate programmer's manual (LPM) contributes to the goal of producing certifiable documentation. The final section, Section 5, consists of concluding remarks.

2. Background

How do we create documentation that facilitates achieving qualities such as verifiability and maintainability? Unfortunately, these qualities are examples of ones that cannot be measured directly. They must be measured indirectly, since their measurement depends on interactions with the environment [18, p. 109]. Moreover, many of the qualities being considered, such as reliability and usability, are external qualities, which means that they are measured by their impact on the user, as opposed to the software developer [5, p. 16]. Although the user and the developer in SC are often the same person, here we are making the distinction based on the role of the individual. With their connection to the end user, external qualities can only be measured when the software is complete. We need internal measures that can be assessed as the software is being built, so that we have confidence that we are on the right track for success. We also need measures that have a smaller scope, so that their measurement is not so daunting. The first section below provides a list of the lower-level qualities that should be targeted for documentation and code to achieve the higher-level qualities mentioned in Section 1. Following the overview of the desirable qualities is background information on the topics of the SRS, LP, and the FP case study.

2.1. *Desirable qualities for documentation*

To address the challenges for adequate documentation for certification of SC software, the qualities described below need to be a priority. All but the final quality listed (abstraction) are adapted from the IEEE recommended practice for producing software requirements [19]. The IEEE guidelines are for the SRS, but we extend the discussion here when the quality can also be applied to the code in the LPM document. From the list below, the DOE guidelines [4] explicitly mention the qualities of completeness, consistency, correctness, traceability, and verifiability. The Canadian Standards Association (CSA) standard [2] mentions completeness and verifiability. Neither document explicitly shows how to achieve these qualities.

Complete: Documentation of the requirements is complete when 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 documentation of the code is complete when sufficient information is given, including traceability to requirements, design decisions and proofs for the code to be verified. The code is complete when every requirement has been addressed.

Consistent: Documentation is 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 a document should not contradict the specification of the same item at another location, either in the same document, a different document, or in the code.

Modifiable: The documentation and code should be developed in such a way that they are easily modifiable so that likely future changes do not destroy the structure of the document. The document structure and tool support should ensure retention of the consistency, traceability and completeness when the changes are made. This is done in part by using cross-referencing for traceability and avoiding manual repetition, as opposed to automatically generated repetition.

Traceable: Documentation should be traceable, as this facilitates maintenance and review. If a change is made to the design or code, then all documentation relating to those segments has to be modified. This property is also important to minimizing the costs of recertification. Additional advantages of traceability include program comprehension, requirement tracing, impact analysis, and reuse [20].

Unambiguous: Documentation of the requirements and design decisions are said to be unambiguous only when every specification has a unique interpretation. The documentation should be unambiguous to all audiences, including developers, users, and reviewers.

Correct: Each requirement should accurately capture the scientific model the stakeholders and experts desire, and every decision for the numerical algorithm and code should be appropriate for the model. To build confidence in correctness, reviewers should be able to inspect and investigate every component of the requirements, design, and implementation. Success on this quality requires maintaining traceability, consistency, and unambiguity.

Verifiable: This quality is repeated from the list of qualities in Section 1, but where the term has been previously applied at a high level to all of the documentation in the current context, the term refers to each individual requirement, design decision, and line of code. All of these must be clear, unambiguous, and testable, so that a person or a machine can verify whether the software product meets the requirements. Correct and verifiable are different qualities. A requirement can be correct and yet not verifiable. For instance, the requirement for solving any linear system of equations ($Ax = b$) is mathematically correct, but it is only verifiable if an error tolerance is allowed and some limits are placed on the range of acceptable inputs.

Abstract: Documented requirements are 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, for example, that Euler's method should be used to solve it. How to accomplish the requirement is a design decision, which is documented during the design phase. Abstraction is also used in code development as a technique to deal with complexity [5, p. 40].

2.2. *Software requirements specification*

Upon determination of the problem to be solved, the first significant document in a rational design process (Fig. 1) is the requirements document. Requirements record all of the expected characteristics and behavior of the system. The document that records the requirements is called the SRS. This document describes the functionalities, expected performance, goals, context, design constraints, external interfaces, and other quality attributes of the software [19].

An SRS provides many advantages for software development [14,21]. For instance, an SRS improves the understandability of the problem to be solved by acting as an official statement of the system requirements for the developers, stakeholders, and end users. Creating an SRS allows for earlier identification of errors and omissions. In SC, the requirements document contributes to improved usability by providing explicit statements of the expected user characteristics, modeling assumptions, definitions, and the range of applicability of the code. It improves the maintainability in the early stages of development, as fixing errors at the beginning is much cheaper than finding and fixing them later. As an additional quality benefit, the reliability and performance of the software cannot be properly verified without a standard against which they 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 (reuse) features of the previous version to upgrade the system by improving the features. This last advantage is important for SC software, where the changes are frequent, as developers explore the problem domain. Also important in SC is reproducibility, which requires a clear statement of the problem being solved if the solution is to be reproduced in the future.

Validation of the scientific model via comparison of the software output against empirical data has to wait for complete implementation, but SC practitioners sometimes appear to forget that verification is possible early in the process. Remmel et al. [22] surveyed the literature on SC software development and concluded that the steps for verification and validation are code verification, algorithm verification, and scientific validation. The important step of model verification seems to be neglected in the characterization of the SC development process. The SRS specifies the model [23]; therefore, verification of the underlying science can begin as soon as the first draft of the SRS is complete. This is especially true if the SRS, like the example shown in Section 3, is designed for verifiability. An SRS can also help with verification via testing, since it manages the cognitive complexity of SC software [24], so that people other than domain experts have hope of designing black-box test cases.

To write an SRS, a common approach is to use a requirements template, which gives guidelines for documenting the requirements. The template provides a framework that suggests an order for filling in details. There are many advantages of using a template in writing an SRS [21,23]. One advantage is that a template increases the adequacy of an SRS by providing a predefined organization that aids in achieving completeness, consistency, and modifiability. A template with a well-organized format acts like a checklist for the writer, thus improving completeness by reducing the chances of missing information. Breaking the information into manageable units and using cross-referencing facilitates traceability and verifiability. Following the template provides structure and rigor, which improves communication between 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 suggestions on how to avoid complications and how to develop an SRS to achieve qualities of good documentation, such as verifiability, maintainability, and reusability [19,25,26]. There is no universally accepted template for an SRS. The current research adapts the SRS template developed for SC software in the report of Smith et al. [23]. This template fits the needs of SC software similar to FP because of its hierarchical structure, which shows the decomposition from abstract goals to concrete instance models. This natural refinement from general to specific improves understandability. The hierarchical structure, together with the traceability matrix, facilitates reusability and maintainability. (Maintainability is particularly important in SC, since many requirements are discovered and modified over the course of a typical project [8].) The template also explicitly addresses nonfunctional requirements for accuracy of the input data, the sensitivity of the model, the tolerance of the solution, and the solution verification and validation strategies [27].

The requirements document suggested in this paper does not exactly match the outline given in Clause 11.2.4 [2]. The SRS given here also includes elements of the theory manual (Clause 11.3.3) [2], such as the mathematical equations, assumptions, and constraints. The theory model is incorporated into the requirements because the theory is needed to express the input/output requirements. A further change to the documentation [2] is that the proposed template is more abstract, as described in Section 2.1. To leave different design decisions open, unlike in the previous case [2], the SRS is silent on the data structures and data flow requirements and programming language selection. These items are design decisions that are postponed until the design documentation.

2.3. *Literate programming*

LP was introduced by D. Knuth as a programming methodology [28]. Its essence can be captured as follows: "…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" ([29], p. 99). LP provides flexibility so that concepts are introduced "…in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other" ([29], p. 99).

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* [28,29] or *chunks* [30,31], 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 [29,32] is called a *weaving* process. Developing a literate program thus becomes a task that resembles 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 produces high-quality code that is impeccably documented.

Smith and Samadzadeh [33] provided an annotated bibliography of LP, up until 1991. Two significant examples of LP applied to SC were given by Nedialkov [34] and Pharr and Humphreys [35].

Moore and Payne [36] used LP to facilitate the verification of a network security device. They proposed that LP techniques be 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 for this lack is that specifications and proofs were documented to facilitate acceptance by mechanical tools rather than by humans.

Essentially, the authors conclude that LP greatly facilitates the development of assurance arguments, since (human) certifiers more naturally understand LP than they do descriptions of machine-generated proofs. This idea is also what motivates the current work that proposes LP to improve QA and to facilitate certification.

In the terminology of Section 1, LP can improve the quality of certification efforts. The improvements start with the quality of understandability. Improvements to understandability facilitates reviews, which in turn improves verifiability and the associated quality of reliability. (This improvement is particularly relevant in SC, in which verification by testing is challenging because of the lack of a test oracle [37].) Understandability also contributes to an improvement in maintainability, since maintainers need to be able to understand the code to make modifications. Whether scientists decide to even maintain or to reuse a program is strongly related to the understandability of code and its documentation. As explained by Roache ([6], p. 362), "User acceptance is also highly dependent on documentation. In my experience, code documentation is second to no other factor in user acceptance, not even ease of use." Finally, LP contributes to reproducibility, since many believe that "successful communication and verification of research results requires that … code is distributed together with results and explanatory prose" [38].

2.4. *FP case study*

The purpose of FP is to perform thermal analysis of a single fuelpin in a nuclear reactor. Each fuelpin includes the following elements, as presented in Fig. 2: a fuel pellet consisting of uranium dioxide (UO_2), the clad material (zircaloy) covering the pellet, and coolant surrounding the clad material. The definitions for the symbols in 2, along with their units of measurement, are given in Fig. 3.

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

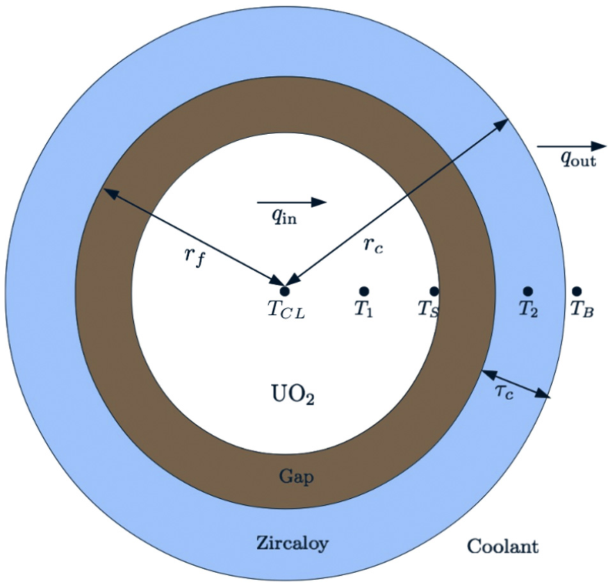


Fig. 2 – Fuel pellet representation (not to scale).

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 $(\frac{kW}{m^{2o}C})$
q_{out}	output heat $(\frac{kW}{m^{2o}C})$
q'_{MWR}	metal water reaction heat $(\frac{kW}{m})$
R_{FUEL}	thermal resistance of fuel $(\frac{m^oC}{kW})$
R_{CLAD}	clad resistance $(\frac{m^oC}{kW})$
R_{GAP}	gap resistance $(\frac{m^oC}{kW})$
R_{FILM}	coolant film resistance $(\frac{m^oC}{kW})$
C_1	thermal capacitance of the fuel $(\frac{kWs}{m^oC})$
C_2	thermal capacitance of the clad $(\frac{kWs}{m^oC})$
C_{CL}	thermal capacitance at the centreline $(\frac{kWs}{m^oC})$

Fig. 3 – Definition of selected symbols for fuelpin analysis.

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 performs thermal analysis using point neutron kinetics, decay heat equations, lumped-parameter fuel-modeling techniques, temperature-dependent thermodynamic properties, a metal–water reaction model, fuel stored energy, integrated fuel power calculations, and trip parameter modeling.

The model for the thermal analysis is based on an electrical circuit analogue of the fuelpin, as given by Fig. 4. A summary of the variables for Fig. 4 can be found in Fig. 3.

3. SRS for fuelpin thermal analysis

We borrowed the template developed by Smith et al. [23] for engineering mechanics and adapted it through addition of a few new sections to suit the nuclear physics domain. The table of contents for the requirements template provided in Fig. 5 shows how the problem is systematically decomposed into more concrete models. Specifically, we start with the high-level problem goals and then we determine the appropriate theoretical models to achieve the goals. The theoretical models are then refined into what are termed instance models, which provide the equations needed to solve the original 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. This facilitates the

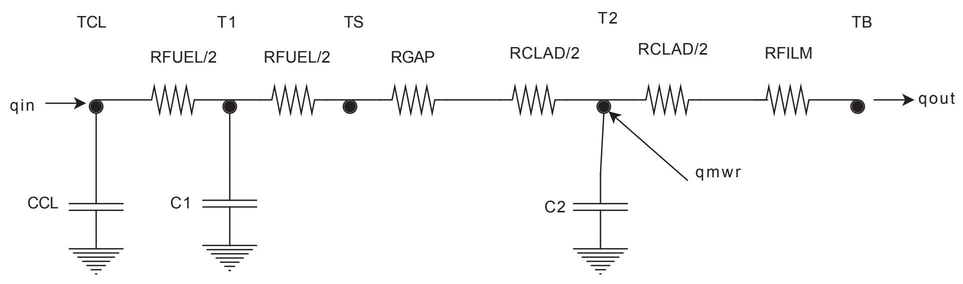


Fig. 4 – Electrical circuit analogue.

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. Having a standard template also helps when comparing between different projects. Besides filling in section headings, the template also requires that every equation either has a supporting external reference or a derivation. Furthermore, every symbol, general definition, data definition, and assumption needs to be used at least once.

The most important sections of (Fig. 5) for improving the desired qualities of an SRS are explained in Sections 3.1–3.6 below, which use subsections for motivation and content, as done in a previous work [23]. The full SRS is presented by Koothoor [16]. Following the excerpts, we evaluate the SRS against the original theory manual, using the desirable documentation qualities given in Section 2.1.

3.1. *Goals*

Motivation: To collect and to document the high-level objectives of the software.

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 2.4.

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

Fig. 5 – Table of contents for software requirements specification for fuelpin analysis.

3.2. *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, when appropriate, assumptions are documented with the forward references made to the data using them. An example assumption from the SRS for FP is given below:

A8: The spatial effects are neglected in the reactor kinetics formulations [IM5].

The notation IM5 is a forward reference to indicate that this assumption is relevant to the derivation of Instance Model 5, which is the label used for the point neutron kinetics model.

3.3. *Theoretical models*

Motivation: To develop an understanding of the theory or principles relevant to the problem [23].

Content: The theoretical models are sets of governing equations or axioms that are used to model the problem described in the problem definition section. In the context of nuclear physics, the theoretical models can be physical laws (including relevant equations), constitutive equations, and so forth. Given below is an example of a theoretical model from our case study.

Number	T1
Label	Conservation of energy
Equation	$-\nabla\cdot\mathbf{q}+q'''=\rho C\frac{\partial T}{\partial t}$
Description	The above equation gives the conservation of energy for 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 del operator, and t is the time.

The equation for conservation of energy is the most important theoretical model in our case study, as it forms the foundation for the derivation of the mathematical models of FP. The theory is given abstractly, without reference to a specific coordinate system, to make it reusable for other problems. The theory is later refined to instance models by applying assumptions and definitions. For example, the coordinate system needs to be selected. 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. How the symbolic equation of conservation of energy is used in deriving the instance models for FP is shown in Koothoor [16].

3.4. *General definitions*

Motivation: This section was added to the original template of Smith et al. [23] as a convenient way to gather and document all the necessary data that are repetitively used in deriving different data definitions.

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

Number	GD1
Label	Cylindrical coordinate system
Equation	$\nabla = \hat{\mathbf{e}}_r \frac{\partial}{\partial r} + \hat{\mathbf{e}}_\theta \frac{1}{r}\frac{\partial}{\partial \theta} + \hat{\mathbf{e}}_z \frac{\partial}{\partial z}$ where $\hat{\mathbf{e}}_r$, $\hat{\mathbf{e}}_\theta$, and $\hat{\mathbf{e}}_z$ form the natural basis of a cylindrical coordinate system.
Description	The location in a cylindrical coordinate system is given by (r, θ, z). Unlike in a Cartesian coordinate system, the basis vectors, except for $\hat{\mathbf{e}}_z$, change as the position changes. The gradient operator is defined above.
Source	FP theory manual, Malvern ([39], p. 667)

3.5. *Data definitions*

Motivation: To collect and organize all physical data needed to solve the problem [23].

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 should 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. Two examples, which are referenced in Section 3.7, are given below: the effective heat-transfer coefficient between the clad and fuel surface (h_g) and the effective thermal resistance (R_1) between T_1 and T_2. Details on the derivation of the associated equations are given by Koothoor [16], as part of the full SRS.

Number	DD19
Label	h_g
Units	$Mt^{-3}T^{-1}$
SI equiv.	$\frac{\mathrm{kW}}{\mathrm{m}^{2\circ}\mathrm{C}}$
Equation	$h_g = \frac{2k_c h_p}{2k_c + \tau_c h_p}$
Description	h_g is the gap conductance τ_c is the clad thickness h_p is the initial gap film conductance k_c is the clad conductivity
Sources	FP theory manual and FP code

Number	DD11
Label	R_1
Units	ML^3Tt^{-3}
SI equivalent	$\frac{\mathrm{m}^\circ\mathrm{C}}{\mathrm{kW}}$
Equation	$R_1 = \frac{f}{8\pi k_{AV}} + \frac{1}{2\pi r_f h_g}$
Description	R_1 is the thermal resistance between the average fuel temperature (T_1) and clad temperature (T_2) (see Fig. 4); f is the average flux depression factor; k_{AV} is the average thermal conductivity through the fuel; r_f is the fuel radius; h_g is the effective heat-transfer coefficient between the clad and fuel surface (see DD19).
Sources	FP theory manual and FP code

3.6. *Instanced model*

Motivation: The mathematical model has to become more concrete before it can be solved by 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, which gives the governing ODE defining the average fuel temperature (T_1), is given below.

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 Theory Manual

3.7. *Evaluation of the SRS*

The documentation (theory manual) accompanying the case study software was completed by highly qualified domain experts with the goal of fully explaining the theory behind FP for the purpose of QA. Presentation of the theory followed a format similar to that in a scientific or engineering journal or in a technical report. Notwithstanding our 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. Using the qualities from Section 2.1 as a reference point, we describe below some quality issues with the original documentation.

Incompleteness and Ambiguity: 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 Fig. 4 and used in several equations in the original documentation, was not defined. The lack of a definition led to ambiguity, which, in turn, led to considerable confusion, since some of the equations suggest that the actual intention may have been to apply T_2 to the surface of the clad, instead of to the centre of the clad, as shown in Figs. 2 and 4. 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, explanation is necessary in order for QA activities to properly judge whether the analogy holds.

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 (DD19), which was used to represent gap conductance in one instance (in the current documentation, this is represented by h_p), as well as to represent the effective heat transfer coefficient between clad and fuel surface in another.

Correctness and Verifiability Concerns: As per the definition of verifiability in Section 2.1, every specification in the document must be the one fulfilled by the software. However, an equation that was seemingly different from the equation in the original documentation solved the effective thermal resistance R_1 in the original source code, suggesting that one of the equations was incorrect. Unlike what is shown in DD11, which matches what was implemented in the code, the equation for R_1 in the theory manual is

$$R_1 = \frac{f}{8\pi k_{AV}} + \frac{1}{2\pi r_f h_g} + \frac{\tau_c}{4\pi r_f k_c}. \tag{1}$$

The apparent disagreement between the theory manual and the code was eventually tracked to an inconsistency in the documentation and not to a fault in the code. The equation in the code, as represented by DD11, is actually equivalent to Equation (1), as long as one corrects the mistake that h_g in Equation (1) should be h_p, as given in DD19. As shown in DD11, h_g combines the heat transfer from h_p and k_c into one term. Similar mismatches between the original theory manual and the source code occurred with the equations for R_2 and T_2. In each case, the issue was found to be inconsistencies between the code and documentation caused by different groupings of terms. Determining the cause of the problem was compounded by the use of the same symbols, albeit with different meanings, in the code versus the documentation. This problem may be avoided by using LP, as discussed in the next section.

Modifiability Concerns and Lack of Traceability: Unfortunately, the original documentation had traceability issues. For instance, there was no reference to the figure representing the electrical circuit analogue (Fig. 4) and the derivation of R_1. A lack of traceability leads to modifiability problems, since managing changes requires knowledge of the impact of the changes.

Abstraction Concerns: The original theory manual did not always meet the recommended goal of having abstract requirements, as discussed in 2.1. In some cases, design decisions were made in the theory manual. For instance, the theory manual included the numerical algorithm for solving the system of ODEs. In the revision of the documentation, the numerical algorithm choice is included in the LPM to facilitate managing complexity through separation of concerns. This improves maintainability, because future changes to the numerical algorithm do not require any changes to the SRS. As shown in Section 4.1, including the numerical algorithm in the LPM improves verifiability, since the algorithm and the associated code are given together.

The SRS proposed in this paper is intended to avoid the above problems with the previous documentation. To achieve the qualities of a good SRS, the template applies the principle of "separation of concerns" by including different sections in order to allow focus on one thing at a time. By dividing the problem into smaller steps and by considering each section simplifies complete and correct documentation of all the necessary information. Sections such as Theoretical Models, General Definitions, and Data Definitions are included before the Instance Models section to systematically solve the problem in a hierarchical manner. This approach of developing 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 helps domain experts to determine whether 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. There is also a requirement that every symbol in the table is used in the documentation somewhere. To address the problems with traceability and modifiability, we use cross referencing between the components. The template requires the use of a unique label for each component and the development of the 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 information needed to build the instanced models. This way of developing the concrete models from the abstract ones while maintaining traceability between them aids in achieving correctness. Inclusion of the derivations of the equations in these sections makes checking correctness easier. Correctness requires documenting every equation, assumption, definition, and model in the respective sections. Every equation included must have a source listed or a derivation provided. Every general definition, data definition, and assumption should be used by at least one other component of the document. Complete and correct specification of all parts, while maintaining traceability between them, makes the task of verification easier. The next step is to continue this systematic, methodical, and traceable approach in the documentation of the design and code, as discussed in the next section.

4. Literate programmer's manual for fuelpin

The stage in software development after complete documentation of the requirements, as shown in Fig. 1, is design. The design phase starts with decomposition of the program into modules, with the recommended decomposition having each module encapsulate a likely change in the software [40]. This division of the program into modules beyond the scope of the current work. (An example of a modular decomposition for SC can be found in a work by Smith and Yu [41]). Instead, we focus on the incremental approach of reimplementing one subroutine from the modules given in the original FORTRAN program. Our goal is to replace the original subroutine and thus to have the numerical outputs of the new literate program match the output of the original FP code. If we were to make a larger change to the original code, then comparison of the two approaches would become unclear. Moreover, if we were to make larger, non-incremental changes, then the explanation for any disagreements on the numerical results would be difficult to track.

The selected subroutine, which was renamed *fuel_temp_*, solves for the time histories of the reactor fuel and clad temperatures. The source code, written in C, was developed along with the logic behind it using LP and implemented with CWEB [32]. The document recording the logic and the source code is termed the LPM. The table of contents of the LPM for *fuel_temp_* is given in Fig. 6. To assist with the navigation of later sections, this figure includes the mapping (shown in italics) between the LPM sections and the corresponding figures in this report. The LPM begins with an abstract view, which is explicitly traced to the SRS. This view is then refined into a concrete implementation, which becomes the code. Tests comparing the original code and the literate code are matched according to their numerical output.

Fig. 6 – Table of contents for the literate programmer's manual.

In this section, we give excerpts from the LPM outlined in Fig. 6. The full LPM for *fuel_temp_* is presented by Koothoor [16]. The cross-references shown in the LPM excerpts refer to the numbering in the full LPM document. The subsections 4.1–4.5 below explain the important items from Fig. 6. The first subsection (Section 4.1) shows an overview of the numerical algorithm, using the notation from the SRS, with additional details related to the selected time-stepping algorithm. Section 4.2 presents a top-down view of the interface for *fuel_temp_*, with an emphasis on what it calculates, without giving the details on how the calculations are done. To connect the code to the SRS, Section 4.3 shows the traceability between the variable names. The trend from abstract to concrete is continued in Section 4.4, where the details from the initialization section for *fuel_temp_* are given. These details are further refined in Section 4.5, where the code corresponding to the chunks shown in Section 4.4 is given. The final section (Section 4.6) evaluates the qualities of LPM against the goals originally outlined in Section 2.1.

4.1. *Algorithm in the notation of the SRS*

Section 3.6 shows the ODE that represents one of the instance models (IM1) for FP. In addition to this ODE, FP is required to solve two other ODEs (IM2 and IM3). Fig. 7 is an excerpt from the LPM, which gives the time-stepping algorithm for solving these ODEs. All the ODEs, termed Equations 1.1–1.3 in Fig. 7, are solved by using the same generic framework developed in the original theory manual. As mentioned previously, given the goal of abstraction, the numerical algorithm is not presented in the SRS, but rather in the LPM. The algorithm uses the subscript k to indicate the current time step. Full details of the instanced models can be found in the work of Koothoor [16].

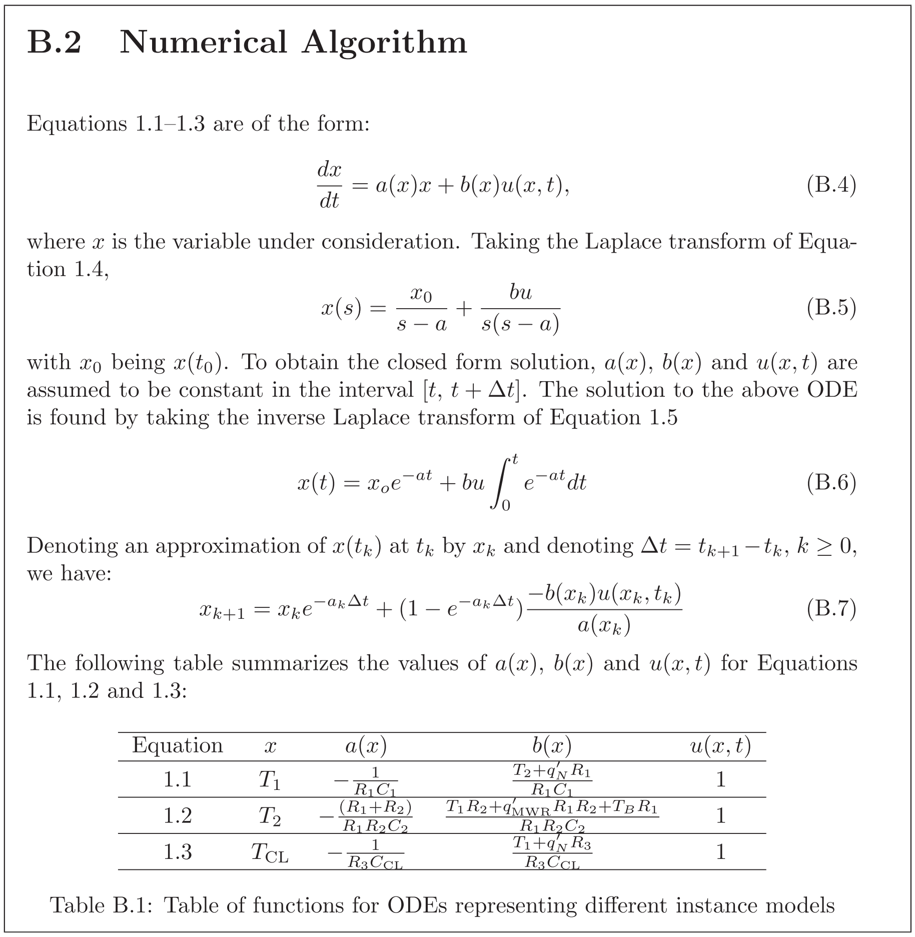
The ODEs are solved in function *fuel_temp_*, a high-level view of which is given in Fig. 8. This figure is written by using the familiar notation of the SRS, but subscripts are added to reflect the numerical algorithm given in Fig. 7. The list of arguments for *fuel_temp_* is long because of the requirement of passing common block variables for the original FORTRAN subroutine as input arguments to the C code implementation of *fuel_temp_*.

The overall control structure of *fuel_temp_* is divided into two sections depending on the value of the input *init_flag*:

1. Initialization section ($*$*init_flag* $==1$): In this section, the initial steady-state values of the variables are found.
2. Dynamic section ($*$*init_flag* $==0$): In this section, the transient values of the variables are determined.

4.2. *Function* fuel_temp_

The view of the function in the previous section shows the connection to the SRS and the numerical algorithm decisions, but it is not in the notation of the implementation language (C). Fig. 9 shows the translation of the function interface from Fig. 8 into C. To make it easier for the reviewer to understand the code, we present the C function using stepwise refinement [42]. That is, first, the overall function is developed with the



B.2 Numerical Algorithm

Equations 1.1–1.3 are of the form:

$$\frac{dx}{dt} = a(x)x + b(x)u(x,t), \tag{B.4}$$

where x is the variable under consideration. Taking the Laplace transform of Equation 1.4,

$$x(s) = \frac{x_0}{s-a} + \frac{bu}{s(s-a)} \tag{B.5}$$

with x_0 being $x(t_0)$. To obtain the closed form solution, $a(x)$, $b(x)$ and $u(x,t)$ are assumed to be constant in the interval $[t,\, t+\Delta t]$. The solution to the above ODE is found by taking the inverse Laplace transform of Equation 1.5

$$x(t) = x_o e^{-at} + bu \int_0^t e^{-at} dt \tag{B.6}$$

Denoting an approximation of $x(t_k)$ at t_k by x_k and denoting $\Delta t = t_{k+1} - t_k$, $k \geq 0$, we have:

$$x_{k+1} = x_k e^{-a_k \Delta t} + (1 - e^{-a_k \Delta t}) \frac{-b(x_k)u(x_k, t_k)}{a(x_k)} \tag{B.7}$$

The following table summarizes the values of $a(x)$, $b(x)$ and $u(x,t)$ for Equations 1.1, 1.2 and 1.3:

Equation	x	$a(x)$	$b(x)$	$u(x,t)$
1.1	T_1	$-\frac{1}{R_1 C_1}$	$\frac{T_2 + q'_N R_1}{R_1 C_1}$	1
1.2	T_2	$-\frac{(R_1+R_2)}{R_1 R_2 C_2}$	$\frac{T_1 R_2 + q'_{\text{MWR}} R_1 R_2 + T_B R_1}{R_1 R_2 C_2}$	1
1.3	T_{CL}	$-\frac{1}{R_3 C_{\text{CL}}}$	$\frac{T_1 + q'_N R_3}{R_3 C_{\text{CL}}}$	1

Table B.1: Table of functions for ODEs representing different instance models

Fig. 7 – Excerpt from the literate programmers manual showing the numerical algorithm for solving the ordinary differential equations.

details abstracted out and replaced by chunks. As shown in Fig. 9, the chunks are the initialization section (chunk 15) and the dynamic section (chunk 53). In the next refinement, shown in Section 4.4, we provide further details of these chunks, with the details themselves including additional chunks, whenever appropriate. The refinement is complete when all of the bottom level chunks are written in C code.

Fig. 9 shows how the SRS notation of the variables is changed to programming notation. Unlike the SRS, in which a wide variety of mathematical symbols and subscripts can be used, the C code is limited to the ASCII character set. We pass the input parameters for *fuel_temp_* by reference, since this is the C equivalent of how we handled the variables in the original FORTRAN code.

4.3. *Naming conventions and traceability back to SRS*

Comparing Figs. 8 and 9 implicitly shows how mathematical symbols such as Δt are translated into C code variables such as $*$*delta*. However, this information is important enough that the traceability between representations in the SRS and the C code needs to be made explicit. As Kelly [12] observed, for scientists to have confidence that their implementation matches their theory, they want variable names that are clearly "tied to the science they represent." Fig. 10 shows a sample of the explicit mapping between the SRS symbols and their C representation for FP. Items under the parameter column give the variables used in the computer code, while items under the store column give the mathematical notation for the respective variables.

Some of the variables that are passed as arguments to the *fuel_temp_* function are input variables, while the others act as both inputs and output. The interpretation of the meaning of the variable is given by the status of *init_flag*. The first variables listed at the top of the input lists in Fig. 10, such as $*$*r_f* and $*$*h_p* (seen in DD11 and DD19 from Section 3.5), represent numerical algorithm parameters, physical quantities, and material properties that are constant for the duration of the simulation; they are the same each time *fuel_temp_* is run. The interpretation of variables at the bottom of the input lists, such as $*$*h_g* and $*$*t_1*, changes depending on the context. As their value on the first run through the algorithm (**init_flag*≡1) does not matter, Fig. 10 shows it as "arbitrary." After initialization, these variables are input as their values at the current time step (k). The meaning of the output variables in Fig. 10 also changes depending on the status of *init_flag*. When the output comes after the first run (**init_flag*≡1), it corresponds to the value at the first time step ($k = 0$). After the initialization, the values correspond to the next time step ($k + 1$).

B.3 Algorithm

fuel_temp_$(\Delta t, q_{\text{NFRAC},k}, q'_{N_{\max}}, r_f, f, \rho_1, \rho_2, h_{ib}, h_p, \tau_g, \tau_c, T_b, p_{\text{dry}}, h_{\text{dry}}, \text{time}, \mathit{init\_flag}, \mathit{MW\_flag}, n,$
$h_{b,k}, q'_{N,k}, k_{c,k}, k_{\text{AV},k}, q'_{\text{MWR},k}, f_{p,k}, T_{1,k}, T_{2,k}, T_{\text{CL},k}, T_{S,k}, h_{c,k}, h_{g,k}, C_{1,k}, C_{2,k}, C_{\text{CL},k}, c_{p,1,k}, c_{p,2,k}, c_{p,3,k}, \Delta H(T_{\text{abs},k}),$
$\delta_{\text{ox},k}, R_{\text{ox},k}, q'_{\text{out},k}, q'_{\text{MWRI},k})$

1. Initialization section ($*\mathit{init\_flag} == 1$):

   - Input: Δt, $q_{\text{NFRAC},0}$, $q'_{N_{\max}}$,r_f, f, ρ_1, ρ_2, h_{ib}, h_p, τ_g, τ_c, T_b, *init_flag*.
   - At t_0 compute y_0
   - Output: y_0,

   where $y_0 = \{h_b, q'_N, k_c, k_{\text{AV}}, q'_{\text{MWR}}, f_p, T_1, T_2, T_{\text{CL}}, T_S, C_1, C_2, C_{\text{CL}}, c_{p,1}, c_{p,2}, c_{p,3}, h_c, h_g,$
   $\Delta H(T_{\text{abs}}), \delta_{\text{ox}}, R_{\text{ox}}, q'_{\text{out}}, q'_{\text{MWRI}}\}$.
   All elements of the set y_0 are evaluated at the 0th time step.

2. Dynamic section ($*\mathit{init\_flag} == 0$):

   At t_{k+1}, $k \geq 0$,

   - Input:Δt, $q_{\text{NFRAC},k+1}$, $q'_{N_{\max}}$,r_f, f, ρ_1, ρ_2, h_{ib}, h_p, τ_g, τ_c, T_b, p_{dry}, h_{dry}, time, *init_flag*, *MW_flag*, n, y_k.
   - compute y_{k+1}, update n when necessary.
   - Output: y_{k+1}, n,

   where $y_{k+1} = \{h_b, q'_N, k_c, k_{\text{AV}}, q'_{\text{MWR}}, f_p, T_1, T_2, T_{\text{CL}}, T_S, C_1, C_2, C_{\text{CL}}, c_{p,1}, c_{p,2}, c_{p,3}, h_c, h_g,$
   $\Delta H(T_{\text{abs}}), \delta_{\text{ox}}, R_{\text{ox}}, q'_{\text{out}}, q'_{\text{MWRI}}\}$.
   All elements of the set y_{k+1} are evaluated at the $k+1$th time step.

Fig. 8 – Algorithm for predicting transient reactor fuel and clad temperatures.

4.4. *Initialization section*

Fig. 11 shows the expansion of the initialization section chunk identified by number 15, which is first mentioned in the high-level view of function *fuel_temp_* in Fig. 9. As was mentioned previously, the details of the algorithm and the code are described in a stepwise manner. Chunk 15 gives the details of the initialization section using a mix of code, such as the initialization of $*n$ and *pi*, and further chunks. The main body of the initialization shows the steps in the initialization as chunks, which is later refined to code.

4.5. *Refinement into chunks*

Fig. 12 shows an example of the expansion of one of the chunks from the initialization section chunk from Fig. 11. In this case, the C code is given for calculating the heat-transfer coefficient (h_c) and the gap conductance (h_g), identified as chunk 21. The data definition for h_g is given in Section 3.5 as DD19. Fig. 12 shows that when a concept is introduced in the LPM, the data definition equation from the SRS is repeated. This is intended 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 checked.

B.4 Overall function

```
2   ⟨fuel temp function 2⟩ ≡
      void fuel_temp_(const float *delta, float *q_NFRAC, float *q_Nmax, float
              *r_f, float *f, float *rho_1, float *rho_2, float *h_ib, float *h_p, float
              *tau_g, float *tau_c, float *t_b, float *p_dry, float *h_dry, float
              *time, short int *init_flag, int *MW_flag, int *n, float *h_b, float
              *q_N, float *k_c, float *k_AV, float *q_MWR, float *f_p, float
              *t_1, float *t_2, float *t_CL, float *t_S, float *h_c, float *h_g, float
              *c_1, float *c_2, float *c_CL, float *c_p1, float *c_p2, float
              *c_p3, float *deltaHT_abs, float *delta_ox, float *rate_ox, float
              *q_out, float *q_MWRI)
      {
        if (*init_flag) { ⟨initialization section 15⟩}
        else { ⟨dynamic section 53⟩}
      }
    This code is used in chunk 94
```

Fig. 9 – High-level view of *fuel_temp_* function.

B.5 Naming Conventions

On input, if $*\mathit{init\_flag} \equiv 1$,

parameter	stores
$*r\_f$	r_f
$*h\_p$	h_p
$*tau\_c$	τ_c
…	…
$*h\_g$	arbitrary
$*k\_AV$	arbitrary
$*t\_1$	arbitrary
…	…

On input, if $\neg *\mathit{init\_flag}$,

parameter	stores
$*r\_f$	r_f
$*h\_p$	h_p
tau_c	τ_c
…	…
$*h\_g$	$h_{g,k},\ k \geq 0$
$*k\_AV$	$k_{AV,k},\ k \geq 0$
$*t\_1$	$T_{1,k},\ k \geq 0$
…	…

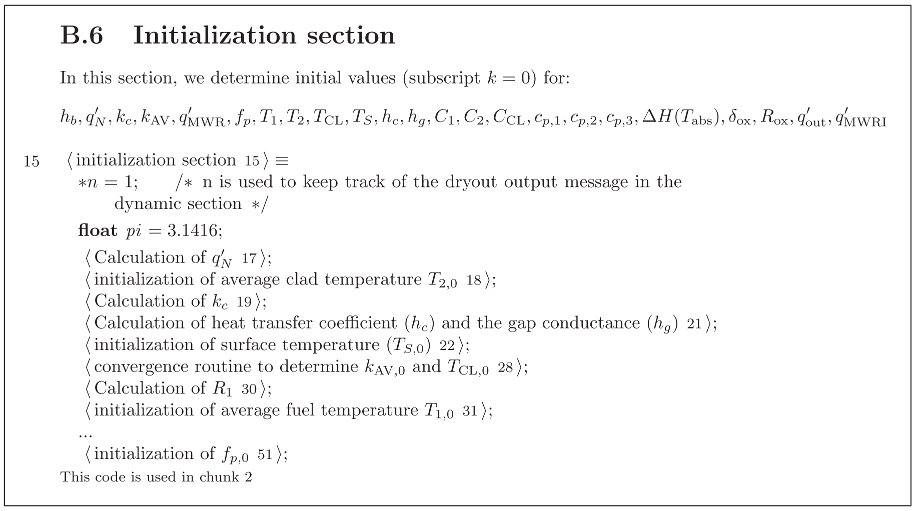
On output, if $*\mathit{init\_flag} \equiv 1$,

parameter	stores
$*h_g$	$h_{g,0}$
$*k\_AV$	$k_{AV,0}$
$*t\_1$	$T_{1,0}$
…	…

On output, if $\neg *\mathit{init\_flag}$,

parameter	stores
$*h_g$	$h_{g,k+1}$
$*k\_AV$	$k_{AV,k+1}$
$*t\_1$	$T_{1,k+1}$
…	…

Fig. 10 – Example of the mapping between SRS names and the code. SRS, software requirements specification.



B.6 Initialization section

In this section, we determine initial values (subscript $k = 0$) for:

$h_b, q'_N, k_c, k_{\mathrm{AV}}, q'_{\mathrm{MWR}}, f_p, T_1, T_2, T_{\mathrm{CL}}, T_S, h_c, h_g, C_1, C_2, C_{\mathrm{CL}}, c_{p,1}, c_{p,2}, c_{p,3}, \Delta H(T_{\mathrm{abs}}), \delta_{\mathrm{ox}}, R_{\mathrm{ox}}, q'_{\mathrm{out}}, q'_{\mathrm{MWRI}}$

15 ⟨initialization section 15⟩ ≡
$*n = 1$; /∗ n is used to keep track of the dryout output message in the dynamic section ∗/
float $pi = 3.1416$;
⟨Calculation of q'_N 17⟩;
⟨initialization of average clad temperature $T_{2,0}$ 18⟩;
⟨Calculation of k_c 19⟩;
⟨Calculation of heat transfer coefficient (h_c) and the gap conductance (h_g) 21⟩;
⟨initialization of surface temperature ($T_{S,0}$) 22⟩;
⟨convergence routine to determine $k_{\mathrm{AV},0}$ and $T_{\mathrm{CL},0}$ 28⟩;
⟨Calculation of R_1 30⟩;
⟨initialization of average fuel temperature $T_{1,0}$ 31⟩;
...
⟨initialization of $f_{p,0}$ 51⟩;
This code is used in chunk 2

Fig. 11 – Excerpt from initialization section of the literate programmers manual for the fuelpin analysis.

4.6. *Evaluation of the LPM*

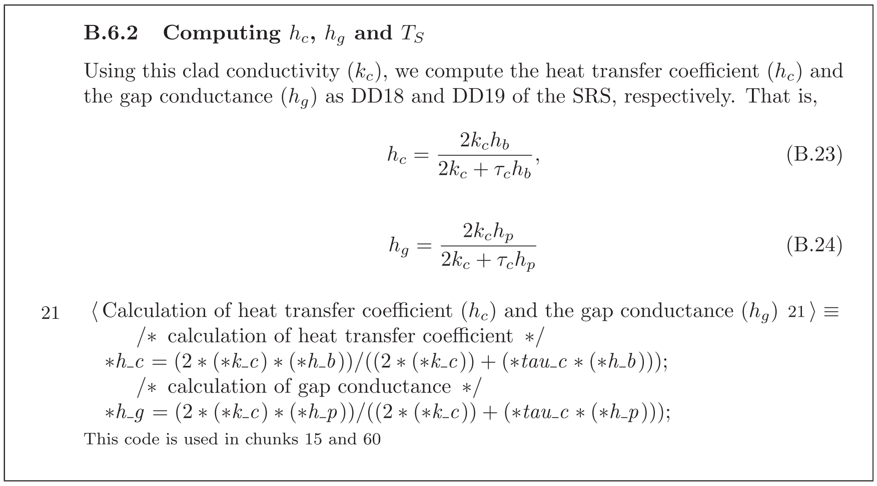
LP makes the design and logic behind the code understandable to a human reader. The LPM achieves the qualities of good documentation (mentioned in Section 2.1), as described below:

Complete: While developing the LP source file, we divided the main algorithm of the program 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. The LPM may also be verified to ensure that every instanced model from the SRS is addressed.

Consistent: To improve consistency, the naming conventions of the variables have been given during the design of the algorithm. As we developed the code following the naming conventions, the probability of inconsistencies is lower. Consistency is also improved since each term was developed as an individual chunk only once and then reused wherever necessary. As shown in Fig. 12, chunk 21 is used in both chunks 15 and 60. As Section 4.3 shows, consistency between the SRS symbols and the code can be explicitly documented.

Modifiable: As implementation of each term is only once, the task of modification becomes easier. If in the future, the implementation has to be changed, then only that chunk consisting of the code has to be modified. As repetition is avoided, consistency is not affected by the modification. Furthermore, as traceability and consistency are maintained, the modifiability is enhanced.

Traceable: For traceability between the components of the LPM, each equation, definition, and table has been labeled and referenced wherever necessary in the document. In LP, the program is developed as a web of interconnected chunks. The LP tools automatically assign 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, the LP tools automatically generate a hyperlink to the place where the chunk is being used. Additionally, at the place where the chunk is being called, the LP tools generate a reference to the number of the chunk that gives the code for the definition. For traceability between the LPM and the SRS, cross referencing is extensively used between the two documents.



B.6.2 Computing h_c, h_g and T_S

Using this clad conductivity (k_c), we compute the heat transfer coefficient (h_c) and the gap conductance (h_g) as DD18 and DD19 of the SRS, respectively. That is,

$$h_c = \frac{2k_c h_b}{2k_c + \tau_c h_b}, \tag{B.23}$$

$$h_g = \frac{2k_c h_p}{2k_c + \tau_c h_p} \tag{B.24}$$

21 ⟨Calculation of heat transfer coefficient (h_c) and the gap conductance (h_g) 21⟩ ≡
/∗ calculation of heat transfer coefficient ∗/
$*h\_c = (2 * (*k\_c) * (*h\_b))/((2 * (*k\_c)) + (*tau\_c * (*h\_b)));$
/∗ calculation of gap conductance ∗/
$*h\_g = (2 * (*k\_c) * (*h\_p))/((2 * (*k\_c)) + (*tau\_c * (*h\_p)));$
This code is used in chunks 15 and 60

Fig. 12 – Excerpt from LPM showing the calculation of h_c and h_g.

Unambiguous: As we developed each term or model only once as a chunk and reused wherever necessary, there is little chance of having two different interpretations for the same definition. Because everything behind the implementation is provided in detail, the chances of having two different interpretations through QA reviews can be reduced.

Correct: Because we developed LP 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 is given again in the LP document, as shown in Fig. 12. This way of implementing each term as a chunk in connection with SRS aids in checking whether the program is implementing each component of the SRS correctly. Confidence in correctness is built by verifying that every line of code traces back to one of the following: a description of the numerical algorithm (in the LPM), a data definition, an instance model, an assumption, or a value from the auxiliary constants table in the SRS.

Verifiable: The quality of verifiability is improved because all the necessary information behind the implementation, such as the development of numerical algorithms, solution techniques, assumptions, and the program flow is given. As traceability between the SRS and LPM has been maintained, compliance of the design and implementation with requirements can be checked. Because documentation of both the design and code is in the same document, it is sufficient for the verifier to have the SRS and LPM to confirm the correctness of the software. Code reading is a key activity for scientists verifying their code [12]. The understandability of LP is a great benefit for code-reading activities.

Abstract: The complexity of the code is managed via abstraction. Because of stepwise refinement, the amount of code that needs to be understood and verified at any step is small. This provides a separation of concerns so that reviewers can just focus on the details in front of them.

Achieving the above qualities implies that the LPM assists in achieving reliability, usability, maintainability, reusability, understandability, and reproducibility, as outlined in Section 1. As a final note on the LPM, the quality of reproducibility is also improved through the use of a makefile for the build system for FP [16]. The makefile explicitly records the FORTAN compilation option -fno-automatic, which is necessary so that variables used within a subroutine maintain their values between successive calls. Without the make file to record this information, a future user would find it difficult to reproduce our results. We know this because this information was lacking in the original documentation we received; therefore, we could not proceed with the project until we rediscovered the compiler switch for ourselves.

5. Concluding remarks

The importance of SC software for nuclear safety analysis has been recognized by the creation of standards and guidelines related to QA. However, standards specifying SC software need to provide more detail. It is not enough to say that a document should be complete, consistent, correct, and traceable; the practitioners need guidance on how to achieve these qualities. This paper provides this guidance in the form of an SRS template and an LPM. The value of the new approach is seen through a case study in which 27 issues of incompleteness and inconsistency were uncovered in the previous documentation. These issues mean that the previous documentation was not verifiable and that it had inadequate traceability to the implementation.

Although some of the problems in the original documentation for the case study would likely have been found with any effort to redo 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 for the developer and force him or her to fill in the necessary information. As the template is developed by 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 this way, the proposed template provides guidelines for documenting the requirements by suggesting an order for filling in the details, thus reducing the chances of missing information. Verification of the documentation involves checking that every symbol is defined; that every symbol is used at least once; that every equation either has a derivation, or a citation to its source; that every general definition, data definition, and assumption is used by at least one other component of the document; and, that every line of code either traces back to a description of the numerical algorithm (in the LPM), to a data definition, to an instance model, to an assumption, or to a value from the auxiliary constants table in the SRS.

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 original version of FP. LP, together with a rigorous change-management policy, mitigates this danger. LPM 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.

The current project documents an existing project, but the same quality of improvements may be achievable for a project developed from “scratch.” The templates are designed for maintainability and reusability; therefore, they will support the migration of the project over time. Although requirements in SC software tend to emerge over the course of the project [8], a commonality analysis usually shows that the requirements fit within the same program family [11]. Moreover, one must remember that here we are faking a rational process [14]. The development process may go down some false paths, but the final documentation and implementation can appear to follow a straight line.

One potential shortcoming of the proposed approach is its reliance on human beings. Following the SRS template and keeping the LP code and documentation in sync should produce high-quality software, but there is a burden on the developers and reviewers to pay attention to details. To reduce the burden on the human users, future work is planned. Additional tool support beyond just the LP tools can be incorporated into the process. Just as a compiler can check that all variables have been initialized, the new tools can check the SRS for completeness and consistency and verify that rules, such as the one that requires that all symbols are defined, is enforced. Code-generation techniques can be used to generalize the idea of LP from the code to cover all software artifacts. A domain-specific language can be designed for capturing mathematical knowledge for families of SC software to reduce the burden on developers. For instance, any repetition of the SRS material in the LPM, such as that shown in Section 4.5, can automatically be generated, rather than relying on a manual process. Ideally, code generation can be used to transform portions of the SRS requirements directly into code. Furthermore, generation techniques may be used to generate documentation to suit the needs of a particular user. For instance, details on the proof or derivation of equations can be removed for viewers using the software for maintenance purposes, but added back in for reviewers verifying the mathematical model. The user can specify the “recipe” for their required documentation using the developed domain-specific language.

Tool support will make the process easier, but practitioners should not wait. The document-driven methods presented here are feasible today and should be employed now to facilitate the certification of SC software used for nuclear safety analysis and design. If an approach such as that described in this paper becomes standard, then the work load will decrease over time with reuse of documentation and as practitioners become familiar with the templates, rules, and guidelines.

Conflicts of interest

The authors of this paper have NO affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the materials discussed in this manuscript.

Acknowledgments

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

REFERENCES

[1] Canadian Nuclear Safety Commission (CNSC), Computer Programs Used in Design and Safety Analyses of Nuclear Power Plants and Research Reactors, Technical Report G-149, Minister of Public Works and Government Services Canada, October 2000.
[2] CSA, Quality Assurance of Analytical, Scientific, and Design Computer Programs for Nuclear Power Plants, Technical Report N286.7–99, Canadian Standards Association, Ontario, (Canada), 1999.
[3] CSA, Guideline for the Application of N286.7-99, Quality Assurance of Analytical, Scientific, and Design Computer Programs for Nuclear Power Plants, Technical Report N286.7.1-09, Canadian Standards Association, Ontario, (Canada), 2009.
[4] United States Department of Energy, Assessment Criteria and Guidelines for Determining the Adequacy of Software Used in the Safety Analysis and Design of Defense Nuclear Facilities, Technical Report CRAD - 4.2.4.1, Office of Environment, Health, Safety & Security, Department of Energy, USA, October 2003.
[5] C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, second ed., Prentice Hall, Upper Saddle River (NJ), 2003.
[6] P.J. Roache, Verification and Validation in Computational Science and Engineering, Hermosa Publishers, Albuquerque, New Mexico, 1998.
[7] A.P. Davison, Automated capture of experiment context for easier reproducibility in computational research, Comput. Sci. Eng. 14 (2012) 48–56.
[8] J.C. Carver, R.P. Kendall, S.E. Squires, D.E. Post, Software development environments for scientific and engineering software: a series of case studies, in: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington (DC), 2007, pp. 550–559. Available from: http://dx.doi.org/10.1109/ICSE.2007.77.
[9] J. Segal, C. Morris, Developing scientific software, IEEE Softw. 25 (2008) 18–20.
[10] P.F. Dubois, Maintaining correctness in scientific programs, Comput. Sci. Eng. 7 (2005) 80–85. Available from: http://dx.doi.org/10.1109/MCSE.2005.54.
[11] W.S. Smith, J. McCutchan, F. Cao, Program families in scientific computing, in: J. Sprinkle, J. Gray, M. Rossi, J.-P. Tolvanen (Eds.), 7th OOPSLA Workshop on Domain Specific Modelling (DSM'07), Montréal, Québec, October 2007, pp. 39–47.
[12] D. Kelly, Industrial scientific software: a set of interviews on software development, in: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '13, IBM Corp., Riverton (NJ), 2013, pp. 299–310. Available from: http://dl.acm.org/citation.cfm?id=2555523.2555555.
[13] J. Segal, When software engineers met research scientists: a case study, Empir. Softw. Eng. 10 (2005) 517–536. Available from: http://dx.doi.org/10.1007/s10664-005-3865-y.
[14] D.L. Parnas, P.C. Clements, A rational design process: how and why to fake it, IEEE Trans. Softw. Eng. 12 (1986) 251–257.
[15] T. Maibaum, A. Wassyng, A product-focused approach to software certification, Computer 41 (2) (2008) 91–93.
[16] N. Koothoor, A Document Driven Approach to Certifying Scientific Computing Software, Master's thesis, McMaster University, Hamilton, Ontario (Canada), 2013.
[17] W.S. Smith, N. Koothoor, N. Nedialkov, Document driven certification of computational science and engineering software, in: Proceedings of the First International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCE), November 2013, 8 p.
[18] H. van Vliet, Software Engineering: Principles and Practice, second ed., John Wiley & Sons, Inc., New York, 2000.
[19] IEEE, Recommended practice for software requirements specifications, IEEE Std 830-1998 (1998) 1–40.
[20] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, Ettore Merlo, Recovering traceability links between code and documentation, IEEE Trans. Softw. Eng. 28 (2002) 970–983. Available from: http://dx.doi.org/10.1109/TSE.2002.1041053.
[21] I. Sommerville, P. Sawyer, Requirement Engineering: a Good Practice Guide, John Wiley & Sons Ltd., 1997.
[22] H. Remmel, B. Paech, C. Engwer, P. Bastian, Design and rationale of a quality assurance process for a scientific framework, in: Proceedings of the 5th International

Workshop on Software Engineering for Computational Science and Engineering, SE-CSE '13, IEEE Press, Piscataway (NJ), 2013, pp. 58–67. Available from: http://dl.acm.org/citation.cfm?id=2663370.2663379.
[23] W.S. Smith, L. Lai, R. Khedri, Requirements analysis for engineering computation: a systematic approach for improving software reliability, Reliable Comput. 13 (2007). Special Issue on Reliable Engineering Computation.
[24] D. Kelly, R. Sanders, The challenge of testing scientific software, in: Proceedings of the Conference for the Association for Software Testing, 2008, pp. 30–36.
[25] ESA, ESA Software Engineering Standards, PSS-05–0 issue 2, Technical Report, European Space Agency, February 1991.
[26] NASA, Software Requirements DID, SMAP-DID-P200-SW, release 4.3, Technical Report, National Aeronautics and Space Agency, 1989.
[27] W.S. Smith, L. Lai, A new requirements template for scientific computing, in: J. Ralyté, P. gerfalk, N. Kraiem (Eds.), Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-specific Requirements Engineering Processes, SREP'05, Paris, France, 2005, pp. 107–121. In conjunction with 13th IEEE International Requirements Engineering Conference.
[28] Donald E. Knuth, The WEB System of Structured Documentation, Stanford Computer Science Report CS980, Stanford University, Stanford (CA), September 1983.
[29] D.E. Knuth, Literate Programming, CSLI Lecture Notes Number 27, Center for the Study of Language and Information, 1992. Available from: http://csli-www.stanford.edu/publications/literate.html.
[30] A. Johnson, B. Johnson, Literate programming using noweb, Linux J. 42 (1997) 64–69.
[31] J. Schrod, The cweb Class, CTAN, the Comprehensive TEX Archive Network, TU Darmstadt, Computer Science Department, WG Systems Programming, Germany (November 1995). Available from: http://ctan.bppro.ca/macros/latex/contrib/cweb/cweb-user.pdf.
[32] D.E. Knuth, S. Levy, The CWEB System of Structured Documentation, Addison-Wesley, Reading, Massachusetts, 1993.
[33] L.M.C. Smith, M.H. Samadzadeh, An annotated bibliography of literate programming, ACM SIGPLAN Notices 26 (1991) 14–20.
[34] N.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, Hamilton, Ontario (Canada), 2006.
[35] M. Pharr, G. Humphreys, Physically Based Rendering: From Theory to Implementation, Morgan Kaufmann Publishers Inc., San Francisco (CA), 2004.
[36] A.P. Moore, C.N. Payne Jr., Increasing assurance with literate programming techniques, in: Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96, 1996, pp. 187–198.
[37] D.F. Kelly, W.S. Smith, N. Meng, Software engineering for scientists, Comput. Sci. Eng. 13 (2011) 7–11.
[38] E. Schulte, D. Davison, T. Dye, C. Dominik, A multi-language computing environment for literate programming and reproducible research, J. Stat. Softw. 46 (2012) 1–24. Available from: http://www.jstatsoft.org/v46/i03.
[39] L.E. Malvern, Introduction to the Mechanics of Continuous Medium, Prentice Hall, Englewood Cliffs, New Jersey, 1969.
[40] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (1972) 1053–1058.
[41] W.S. Smith, W. Yu, A document driven methodology for improving the quality of a parallel mesh generation toolbox, Adv. Eng. Softw. 40 (2009) 1155–1167. Available from: http://dx.doi.org/10.1016/j.advengsoft.2009.05.003.
[42] E.W. Dijkstra, Notes on structured programming, in: O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare (Eds.), Structure Programming, Academic Press Ltd., London, (UK), 1972, pp. 1–82. Available from: http://dl.acm.org/citation.cfm?id=1243380.1243381.