Debunking the Myth that Upfront Requirements are Infeasible for Scientific Computing Software

Spencer Smith and Malavika Srinivasan
Computing and Software Department
McMaster University, Canada
Email: smiths@mcmaster.ca, sriniva@mcmaster.ca

Sumanth Shankar
Mechanical Engineering Department
McMaster University, Canada
Email: shankar@mcmaster.ca

Abstract—**Many in the Scientific Computing Software community believe that upfront requirements are impossible, or at least infeasible. This paper shows requirements are feasible with the following: i) an appropriate perspective ('faking' the final documentation as if requirements were correct and complete from the start, and gathering requirements as if for a family of programs); ii) the aid of the right principles (abstraction, separation of concerns, anticipation of change, and generality); iii) employing SCS specific templates (for Software Requirements and Module Interface Specification); iv) using a design process that enables change (information hiding); and, v) the aid of modern tools (version control, issue tracking, checking, generation and automation tools). Not only are upfront requirements feasible, they provide significant benefits, including facilitating communication, early identification of errors, better design decisions and enabling replicability. The topics listed above are explained, justified and illustrated via an example of software developed by a small team of software and mechanical engineers for modelling the solidification of a metal alloy.**

Index Terms—**software engineering; scientific computing; requirements analysis; information hiding; documentation; casting**

I. Introduction

Upfront requirements are often considered infeasible for Scientific Computing Software (SCS). The truth is that requirements for SCS are no more challenging than for any other domain – requirements are difficult for everyone. What is needed is not excuses, but careful, rigorous, sometimes painful thought and effort, as early and as often as possible. With a change in attitude and perspective, the right tools, templates and principles, upfront requirements are possible for SCS.

Throughout this paper, the term upfront requirements does not literally mean requirements documentation that is entirely complete and correct as the first activity in the development process. Rather, upfront requirements means that requirements are considered early in the design process and the documentation is 'faked' [1] as if requirements were the first step.

Requirements are documented in a Software Requirements Specification (SRS). An SRS describes the functionalities, expected performance, goals, context, design constraints, external interfaces, and other quality attributes of the software [2]. In a scientific context, an SRS records the necessary terminology, notations, symbol definitions, units, sign conventions, physical system descriptions, goals, assumptions, theoretical models, data definitions, instance models and data

constraints [3]. An SRS provides many advantages during software development [1]. 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. As Boehm's data shows [4], early identification of errors implies a significant return on investment, since the cost of fixing errors increases dramatically for later development stages. An SRS is invaluable for verification, since the quality of software cannot be assessed without a standard against which to judge it. Better design decisions are facilitated by the information captured in an SRS.

Despite these benefits, an SRS is rarely written for SCS software. As the following quotes highlight, previous research has repeatedly shown that many in the community believe that upfront requirements are infeasible for SCS:

- "Full up-front requirement specifications are impossible: requirements emerge as the software and the concomitant understanding of the domain progress." [5]
- "Since scientific software is deeply embedded into an exploratory process, you never know where its development might take you. Thus, it is hard to specify the requirements for this kind of software up front as demanded by traditional software processes." [6]
- "The research scientists ... do not appreciate the need to articulate requirements fully and upfront as demanded by a staged methodology, and found this articulation very difficult to do." [7]
- "computational scientists generally adopt an agile development approach because they generally do not know the requirements up front." [8]
- "Supplying requirements upfront ran counter to the [scientists] previous experience of developing their own software in the laboratory." [9]
- "None of our interviewees created an up-front formal requirements specification. If regulations in their field mandated a requirements document, they wrote it when the software was almost complete." [10]

Why is the negative attitude toward upfront requirements, and requirements in general, so prevalent? Part of the reasoning may be that SCS practitioners rarely have formal training in Software Engineering (SE) [11]. Moreover, experiments

with applying SE to SCS have often made the mistake of not tailoring the approach to scientists. SCS developers have historically not documented requirements, but that does not mean that they should not. We cannot ignore the possibility that requirements are feasible with the right principles, techniques and tools. Given the benefits of requirements, investigating their feasibility is worth further effort.

This effort is justified because of quality concerns. Faulk et al. [12] observe, "growing concern about the reliability of scientific results based on ... software." Embarrassing failures have occurred, like a retraction of derived molecular protein structures [13], false reproduction of sonoluminescent fusion [14], and fixing and then reintroducing the same error in a large code base three times in 20 year [15]. A recent report on directions for SCS research and education states: "While the volume and complexity of [SCS] have grown substantially ... [SCS] traditionally has not received the focused attention it so desperately needs ... to fulfill this key role as a cornerstone of long-term collaboration and scientific progress" [16]. Estimates suggest that the number of released faults per thousand executable lines of code during a given program's life cycle is at best 0.1, and more likely 10 to 100 times worse [17].

The sections of this paper present arguments for the value and feasibility of upfront requirements for SCS. To lend weight to the arguments, a specific real-world software development project is cited. The example consists of software to model the solidification of molten metal alloys [18]. An overview is given in Section II. The arguments in favour of upfront requirements starts with the previously mentioned idea of 'faking' a rational development process (Section III). Following this, the specific principles and templates for writing an SRS tailored to the needs of SCS are presented in Section IV. For upfront requirements to be effective in an environment where those requirements are constantly changing and evolving, the approach to design also needs to accommodate change. Application of the principles of design for change, along with examples, is discussed in Section V. Experience has shown that principles and techniques are not enough to lead to a change in software development; therefore, tools that facilitate upfront requirements are summarized in Section VI.

II. SOFTWARE FOR SOLIDIFICATION

The running example throughout this paper is called SFS: Software For Solidification [18]. This software is used to analyze solidification data for molten metal alloys. SFS was developed by this paper's authors as a partnership between software and mechanical engineers.

SFS is typical of SCS for two reasons: 1) the motivation is from a practical scientific/engineering challenge – in this case, how to reduce the number of defective cast metal parts; and, 2) the high level goal (using experimentally measured temperature history data to understand the effect of temperature and temperature gradients on the solid fraction) was clear from the outset, but the details on the appropriate assumptions and numerical techniques were initially uncertain.

The main input for the software consists of temperature history data at different heights within a cylindrical sand mould containing a molten metal alloy. The dimensions of the cylinder are selected to ensure approximately unidirectional heat removal. The experimental setup is shown in Figure 1. The thermocouples for recording temperatures are represented by the label T_i, where i is the thermocouple number. A water jet is directed at the bottom of the cylinder to aid in the cooling process and to facilitate 1D heat transfer.
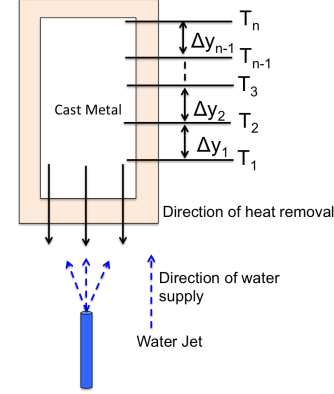

Fig. 1: Experimental setup for cooling a liquid metal alloy

During solidification, heat is given out from the liquid phase to form a solid. During this process, the alloy undergoes a phase transition from liquid to 2 phase (solid + liquid) and finally to solid. Typical temperature data for the sequence of thermocouples is shown in Figure 2. The numbering of the thermocouples increases as the distance from the bottom of the cylinder increases. For any instant of time, the thermocouples with the higher temperatures and slower rates of cooling are those nearer the top of the cylinder.
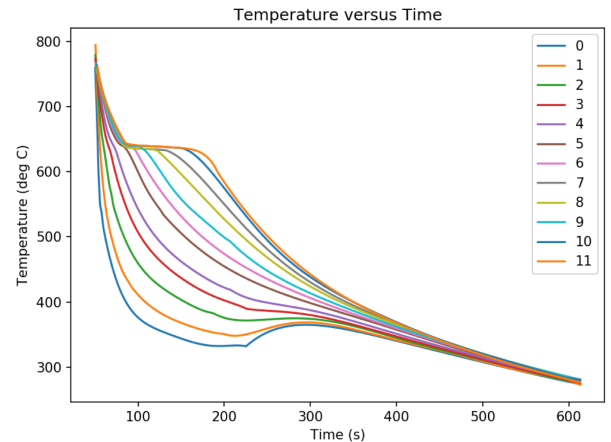

Fig. 2: Typical cooling curves

The output of the software is the fraction solid, either as a function of time, temperature or cooling rate. The fraction solid ranges from 0 (liquid) to 1 (solid). The fraction solid

is obtained by solving an Ordinary Differential Equation (ODE) based on the temperature data, material properties and processing conditions [18].

III. Faking a Rational Design Process

'Upfront requirements' are (almost always) impossible, *if* the term means software development has to start with a complete and correct statement of requirements. However, 'upfront requirements' are possible, if the definition is softened to mean requirements documentation is started as early as possible, and the documentation is continuously maintained, as if a rational design process has been followed [1]. The almost impossibility of having perfect information at the start of a project does not justify not writing requirements. Eventually we will understand the problem at hand – the documentation should be (re)written from that perspective.

An argument against documentation is that developers consider reports for each stage of software development as counterproductive [19, p. 373]. However, reports are only counterproductive if the development process has to follow the same rational process implied by the documentation. In SCS (and other software domains for that matter) frequent change is inevitable. As Parnas and Clements [1] point out, the most logical way to present the documentation is still 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" [20]. Reusability and maintainability are important qualities for scientific software. Documentation that follows a faked rationale design process is easier to maintain and reuse because the documentation is understandable and standardized. Understandability is improved because the faked documentation only includes the 'best' version of any artifacts, with no need to incorporate confusing details around the history of their discovery [1]. Standardization on any process, with a rational process being a logical choice as a standard, facilitates design reviews, change management and the transfer (and modification) of ideas and software between projects [1].

The rational design process promoted here (3), and employed for SFS, is a variation on the waterfall model described by Parnas and Clements [1]. This V-model variation has a testing related phase associated with each step in the typical waterfall process. The steps include writing requirements in a Software Requirements Specification (SRS), a software architecture in the Module Guide (MG), and the detailed design in the Module Interface Specification (MIS) [21], [22]. Each of these steps has a corresponding Verification and Validation (VnV) plan and associated report.

Upfront requirements are possible if the documentation is maintained from this 'faking it' perspective. However, the documentation will only be feasible if the requirements (Section IV) and design documentation (Section V) facilitate change. Tool support (Section VI) will also be critical, or the frequent changes will be overwhelming. For the SFS example, we started with the SRS, then proceeded to the code, and then refined the requirements and began the design documentation.
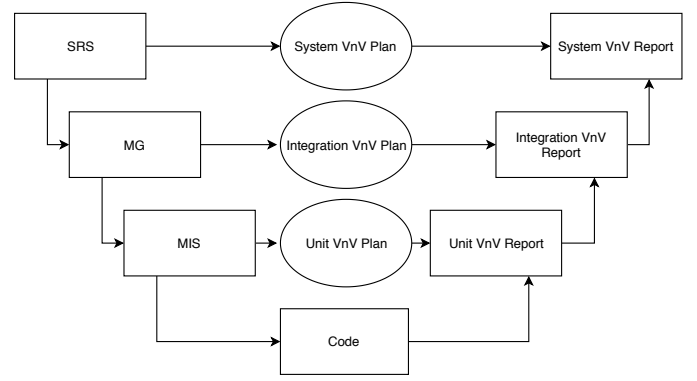

Fig. 3: Rational design process, following the V-Model

Following this first "pass," the focus for subsequent development switched frequently between documents, code and testing. However, the documentation was always maintained as if all of the steps occurred in a rational order.

IV. Requirements for Change

Upfront requirements can be feasible with a template tailored to the needs of SCS (Section IV-A) and the use of the following principles: Abstraction (Section IV-B), Separation of Concerns (Section IV-C), Anticipation of Change (Section IV-D), Generality (Section IV-E) and Replicability (Section IV-F). The first four principles come from SE [23], while the last is the cornerstone of the scientific method.

A. SRS Template

Writing an SRS generally starts with a template, which provides guidelines and rules for documenting the requirements. Several existing templates contain suggestions on how to avoid complications and how to achieve qualities such as verifiability, maintainability and reusability [24], [2], [25]. Templates are generally selected to suit the problem domain. A template designed for the needs of SCS [26], [3] was selected for documenting SFS, as illustrated in Figure 4. The recommended template is suitable for science, because of its hierarchical structure, which decomposes abstract goals to concrete instance models, with the support of data definitions, assumptions and terminology. Excerpts from the SRS for SFS will be used in the sections below to illustrate how the template and the writing of the SRS supports the SE and scientific principles that make upfront requirements documentation feasible.

B. Abstraction

Abstraction focuses on what is important, while ignoring what is irrelevant [23]. Requirements at the right abstraction level lead to a model where the likely changes are excluded as irrelevant details. With the aid of the principle of abstraction, the SRS for SFS underwent very little modification over the life of the project. Although considerable exploration and experimentation were necessary, this exploration took place through the design and implementation, not the SRS.

Fig. 4: Table of Contents for SRS

For a given experiment with a metal alloy, using the thermocouple locations, temperature readings, material properties and initial conditions, SFS:

GS1: Computes the solid fraction (f_s) as a function of temperature (T) and cooling rate ($f_s(T, \frac{dT}{dt})$).

Fig. 5: Goal Statement

The overall goal for SFS (Figure 5) was written so that it would remain true throughout the project. The goal statement does not list specific material properties or initial conditions, since they may change as different models are explored.

The goal statement mentions temperature readings. This concept needs to be further refined to how discrete readings are transformed into a continuous approximation of the evolution of the temperature field over time. This is done through the general definition (GD4) of the transformation of the experimental data (Figure 6). GD4 abstracts the concept of taking experimental data ($Tdata$) at the thermocouples (shown in Figure 2) over time and determining a function $T(y, t)$. This function takes the position y (m), as measured from the bottom of the cylinder, and the time t (second) and returns the temperature (°C). fit() is a function that takes the thermocouple data $Tdata$, the locations of the thermocouples y_{TC} and the

Number	GD4
Equation	$T(y, t) = \text{fit}(Tdata, y_{TC}, dt)$ where $\text{fit} : \mathbb{R}^{m \times n} \to \mathbb{R}^n \to \mathbb{R} \to (\mathbb{R} \to \mathbb{R} \to \mathbb{R})$
Descript.	...

Fig. 6: General definition of transforming experimental data

Number	IM4
Input	$T(y, t)$ (see DD4), from which $\frac{\partial T}{\partial t}$ and $\frac{\partial^2 T}{\partial y^2}$ can be derived, as required
	Material properties $C_v^L(T)$, $C_v^S(T)$, $\rho_L(T)$, $\rho_S(T)$, α_b (from IM2), α_e (from IM3), and L
	y^*, (t_L, T_L) from DD6 and (t_S, T_S) from DD7
Output	Solve $f_s(t)$ at location y^* such that the following ODE is satisfied with $f_s(t_L) = 0$: $$\dot{f}_s(f_s, t) = \frac{C_v(f_s)}{L\rho(f_s)} \left[\frac{\partial T(t)}{\partial t} - \alpha(f_s) \frac{\partial^2 T(t)}{\partial y^2} \right] \text{ where...}$$

Fig. 7: Instance model for finding fraction solid

time step dt, and returns the appropriate function $T(y, t)$. m is the number of instants of time where the thermocouple data is measured and n is the number of thermocouples. The specific approach for fitting, such as interpolation versus regression, is not specified, since these details are not relevant at this stage. All that needs to be known is that there will be a for finding the temperature at any point in space and time.

The function $T(y, t)$ is used to find the solid fraction f_S, as given in the excerpt of an instance model (Figure 7). The instance models are the closest the SRS gets to code, since they list the program inputs and outputs. Abstraction is still employed though, since the numerical algorithm for finding partial derivatives of $T(y, t)$ is not given, nor is the algorithm for solving the ODE. The cross-references to Data Definitions (DD) in Figure 7 show the interrelation between the SRS parts.

C. Separation of Concerns

With the principle of separation of concerns, we reduce a complex problem to a set of simpler problems [23]. This principle was used in the design of the SRS template [26], [3]. For instance, the different sections of the table of contents (Figure 4) can each be considered separately. As an example, thinking about functional requirements, like the governing equations for physical models, is made easier if one momentarily neglects consideration of nonfunctional requirements, like performance and portability.

Upfront requirements become much more stable if the SRS documents the physical models, but does not address numerical methods. The decisions on numerical techniques, such as the ODE solver, should be postponed to the design stage. Knowing the most appropriate numerical technique is

difficult at the outset, and the initial choice is likely to change. The physics on the other hand, if written with the right abstraction, is much less likely to change.

Separation of concerns allows us to improve the overall software quality and software developer productivity because it allows us to distribute the human effort and thought throughout the development process. Human nature is to put off difficult tasks. For instance, a developer might prefer to develop code rather than think about how to test their code. However, many testing concerns do not have to wait until the code is written. The associated thought and effort can happen upfront. For instance, the properties of a correct solution (Section 5.vii of the SRS) are often known at the outset of a project. These properties should be documented, and used as the basis for test cases in the System VnV Plan (Figure 3). What the SRS template terms 'properties of a correct solution' is often called initial hypotheses and metamorphic tests by others [27].

SFS provides an example of a property of a correct solution for the temperature field $T(y, t)$, which is shown graphically in Figure 2 and described in Figure 6. Physics imposes the property that for any t, $t > 0$, $(\forall y_1, y_2 : \mathbb{R} | 0 \leq y_1 \leq H \wedge 0 \leq y_2 \leq H \wedge y_1 < y_2 : T_t(y_2) \geq T_t(y_1))$ where $T_t = \lambda y : T(y, t)$. That is $T_t(y)$ is non-decreasing with increasing y (λ is an anonymous function). This property was known at the outset of the project and provided assertions in the unit tests that later detected a mistake in the calibration of the thermocouples.

D. Anticipation of Change

Upfront requirements are possible, if we anticipate future changes. The SRS template used for SFS was designed with change in mind [26], [3]. For instance, assumptions are listed with their traceability to other parts of the SRS explicitly indicated. As an example, for SFS one of the assumptions is that the density of the alloy can be expressed as a linear combination of the density values at the beginning and end of the solidification. This assumption maps to the definition of density and the models that use density for their calculations. If this assumption should change, the consequences of this change are clearly indicated. Other aspects of the template that support change include explicit traceability between the information in the SRS, as indicated in Figure 8 (documented by cross-references and a full traceability matrix), a section on likely changes (for instance, the heat transfer at the bottom of the cylinder may later be changed from constant to temperature dependent), and the use of symbolic constants.

The template anticipates change, but successfully using it requires a new perspective where the author(s) think of their task as documenting a family of SCS programs, instead of just one. As for the properties of a correct solution mentioned in Section IV-C, this will mean intense effort/thought at the beginning of the development process. A program family approach, where commonalities are reused and variabilities are identified and systematically handled, is natural for scientific software [28]. The laws of physics are stable; they are almost universally accepted, well understood, and slow to change. At the appropriate abstraction level, many problems have
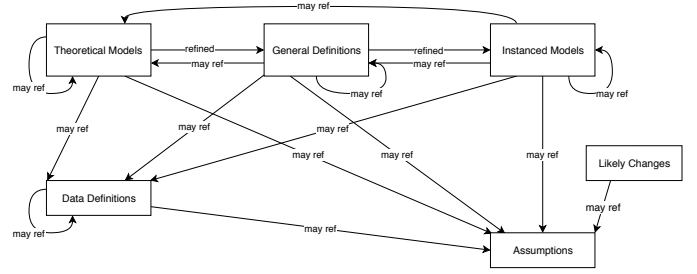

Fig. 8: Relationship between parts of the SRS

significant commonality, since a large class of physical models are instances of a relatively small number of conservation equations (conservation of energy, mass and momentum). The challenge is to know which simplifying assumptions are appropriate. Therefore, as mentioned above, the assumptions need to be documented clearly, and explicit traceability is required to show which parts of the model they influence. For general purpose SCS, like a solver for a system of linear equations, the program family should generally be clear from the start, since general purpose tools are based on well understood mathematics.

SFS has benefitted from application of the principle of anticipation of change because several sections are straightforward modifications of previous SRS documents. For instance, SFS shares verbatim the abstract theoretical model for conservation of thermal energy with projects on modelling a fuel pin in a nuclear reactor [29] and predicting the temperature of a solar water heating tank (https://github.com/smiths/swhs).

E. Generality

To aid future reuse and to facilitate software design and implementation, the SRS should be written with generality. This often involves using abstraction, which is why the abstract theoretical model of conservation of thermal energy can be reused in multiple projects (Section IV-D).

A key example of generality in SFS is found in the decision to write the instance model on the calculation of f_S in the standard form of a system of ODEs (Figure 7). Other work related to calculation of f_S does not target this standard form [30]. As a consequence, this other work requires development of an ad hoc algorithm for determining the solid fraction. The design and implementation of SFS on the other hand can, because of its generality, take advantage of the wealth of numerical ODE solver libraries available.

F. Replicability

Although reproducibility is the cornerstone of the scientific method, until recently it has not been treated seriously in software [31]. Fortunately, in recent years multiple conferences, workshops and individuals are calling for dramatic change [32]. Reproducibility problems are even more extreme when the goal is replicability. A third party should be able to repeat a study using only the description of the methodology

from a published article , with no access to the original code or computing environment [31]. However, replicability is rarely achieved, as shown for microarray gene expression [33] and for economics modelling [34]. Replicability is not achieved because journal papers cannot possibly document all definitions, assumptions, models etc. An SRS addresses completeness and ambiguity problems, since a template is followed, there is no page limit and, SE principles are followed. The goal is to capture all of the required knowledge, including derivation of equations and rationales. If the SCS community is serious about replicability, documentation of an SRS at the level of detail described in this paper is required.

V. DESIGN FOR CHANGE

Upfront requirements, using the principle of 'faking it,' are only feasible if the design can be modified in response to changing requirements. This is possible by applying the same principles listed in the previous section. The design of SFS is captured in the Module Interface Specification (MIS). This document follows a previous template [22]. This template is based on a textual design specification [23, Chapter 4] and the mathematical formalism of a module state machine [35].

An important module in SFS is the one responsible for calculating the temperature at any point in space and time by using the temperature history data at the thermocouples, as shown in the SRS under general definition GD4 (Figure 6). The design uses a special technique for anticipation of change: information hiding [36]. The changeable secrets of the modules are hidden behind the stable interface. This means the design decision can be changed, without a need to modify the modules that rely on the provided services.

Separation of concerns is used in the design to handle the complexity of fitting over two dimensions (distance and time). First an Abstract Data Type (FunctT) is introduced to capture the cooling curves, which are shown for each thermocouple in Figure 2. Second an ADT is introduced for the contours (ContrT), which consists of a sequence of cooling curves (FunctT objects), with each curve at a different location. Apply the principle of generality, the modules are not specific for temperature; they can be used for any two dimensional data. For instance, the same approach is used in a program to calculate the risk of glass panes failing (https://github.com/smiths/caseStudies/tree/master/CaseStudies/glass).

The syntax of the access programs for FunctT is given in Figure 9. The MIS template shows application of the principle of separation of concerns, by separating the syntax and semantics. The interface provides a constructor (new FunctT) that takes two vectors of data and returns a FunctT object. Accessors include minD and maxD for finding the extreme limits of the independent data variable. The accessor eval returns the value of the dependent variable (x) given a value for the independent variable (y). The syntax of FuncT shows exceptions if the data for the independent variable data is not in ascending order, of if the number of data points in the two sequences do not match, or if a function evaluation is sought outside of the domain of the given data.

Routine	In	Out	Exceptions
new FunctT	X_{in}:$\mathbb{R}^n$, Y_{in}: $\mathbb{R}^n$	FunctT	IVarNotAscend, SeqSizeMismatch
minD		$\mathbb{R}$	
maxD		$\mathbb{R}$	
eval	$x : \mathbb{R}$	$\mathbb{R}$	OutOfDomain

Fig. 9: MIS syntax for FunctT

State Variables
f: $\mathbb{R} \to \mathbb{R}$
minX: $\mathbb{R}$
maxX: $\mathbb{R}$

new FunctT(X_{in}, Y_{in}):
- transition: $minX, maxX, f := X_0, X_{|X|-1}, (\lambda v : interp(X_{in}, Y_{in}, v))$
- output: $out := self$
- exception: $(\neg isAscending(X_{in}) \Rightarrow IVarNotAscend| |X_{in}| \neq |Y_{in}| \Rightarrow SeqSizeMismatch)$
...
eval(x):
- output: $out := f(x)$
- exception: $(\neg(minX \leq x \leq maxX) \Rightarrow OutOfDomain))$

Local Functions
interp: $\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$
interp(X, Y, v)
$\equiv$ interpQuad($X_{i-1}, Y_{i-1}, X_i, Y_i, X_{i+1}, Y_{i+1}, v$) where $i = $ index(X, v)
interpQuad: $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
interpQuad($x_0, y_0, x_1, y_1, x_2, y_2, x$) $\equiv y_1 + \frac{y_2-y_0}{x_2-x_0}(x - x_1) + \frac{y_2-2y_1+y_0}{2(x_2-x_1)^2}(x - x_1)^2$
index: $\mathbb{R}^n \times \mathbb{R} \to \mathbb{N}$ *# constructor ensures seq. is ascending*
index(X, x) $\equiv i$ such that $X_i \leq x < X_{i+1}$

Fig. 10: MIS semantics for FunctT

The corresponding semantics for FunctT is given in Figure 10. The MIS uses abstraction, with the functional fit represented by a state variable with a function type ($\mathbb{R} \to \mathbb{R}$). The specific function is hidden. To make it easy to change, a local function is defined for interpolation (interp). This function can be changed to any order of interpolation, or it could be changed to regression, or a spline. Defining a new functional fit, just requires redefining interp. Following the principle of information hiding, the interface to FunctT would remain unchanged. This facilitates experimentation, which was necessary for determining the best approach for SFS.

Figure 11 shows the syntax and semantics for the MIS for ContourADT. The full data set is built up by adding each successive thermocouple's data. The accessors, eval, dydx and d2ydx2 are used to calculate the values of $T(y, t)$, $\frac{\partial T}{\partial t}$ and $\frac{\partial^2 T}{\partial y^2}$, respectively. The access program slice returns a new

Syntax: *Exported Access Programs*			
Routine	**In**	**Out**	**Exceptions**
ContrT			
add	s: FunctT, $z : \mathbb{R}$		IVarNotAscend
getC	$i : \mathbb{N}$		InvalidIndex
eval	$x : \mathbb{R}, z : \mathbb{R}$		OutOfDomain
dydx	$x : \mathbb{R}, z : \mathbb{R}$		OutOfDomain
d2ydx2	$x : \mathbb{R}, z : \mathbb{R}$		OutOfDomain
slice	$x : \mathbb{R}$	FunctT	

Semantics: *State Variables*
S: sequence of FunctT
Z: sequence of $\mathbb{R}$

new ContrT(i):
- transition: $S, Z := <>, <>$
- exception: none

add(s, z):
- transition: $S, Z := S || <s>, Z || <z>$
- exception: $exc := (|Z| > 0 \land z < Z_{|Z|-1} \Rightarrow$ IVarNotAscend)

eval(x, z):
- output: $out := \text{self.slice}(x).\text{eval}(z)$
- exception: none

...
slice(x):
- output: $out := \text{FunctT}(Z, \langle S_0.\text{eval}(x), ..., S_{|S|-1}.\text{eval}(x)\rangle)$
- exception: None

Fig. 11: MIS for ContrT

FuncT that would hold the temperature values through the height of the cylinder for a given value of the dependent variable t. In this case the slice will use the same fitting algorithm as for the cooling curves. In the actual MIS for SFS the order of the interpolating polynomial was exposed as an input variable, so that the fit through the height could be a different order than the fit for an individual cooling curve.

VI. Tool Support

Experience has shown that principles and templates are not enough for dramatic change in development practices. Although the value of documentation is recognized, there is a sense that writing and maintaining the documentation is too great an effort [37]. The positive influence of tool support is observed in the CRAN (Comprehensive R Archive Network) community, where documentation generation and consistency checking tools allow a single SCS developer to achieve similar software quality to a team of developers [38].

SCS developers should use tools for issue tracking and version control. Issue tracking is considered a central quality assurance process [39]. For SFS, git was used for version control and GitLab for issue tracking. The issue tracker

facilitated bringing the team knowledge together, especially when review questions were assigned to the domain experts.

Going forward, a knowledge-based approach for scientific software development holds promise. Ideally, developers should be able to create high quality documentation without the drudgery of writing and maintaining it. A potential solution is to generate the documentation automatically by using Domain Specific Languages (DSLs) over a base of scientific knowledge. DSLs and code/document generation provide a transformative technology for documentation, design and verification [6], [40]. DSLs allow upfront changes to be propagated through a project and they provide checks for consistency and completeness. Moreover, a generative approach removes the maintenance nightmare of documentation duplicates and near duplicates, since knowledge is only captured once and automatically transformed as needed.

VII. Concluding Remarks

Without dramatic intervention, our collective confidence SCS is due for a collapse. Successfully building SCS requires communication between software developers and experts from multiple domains. Collaboration is difficult at the best of times, and is made worse because developers avoid the upfront effort of thinking about the requirements and the known properties of a correct solution. Developers tend to favour handcrafted solutions over adapting SE processes, methods and tools [12]. Handcrafted solutions do not account for the inevitable changes in requirements, design and implementation. The twin challenges of changing requirements and inadequate documentation conspire to make computational results notoriously difficult to reproduce, especially in the case of one researcher independently replicating another's results [40].

The solution to quality problems is applying, adapting and developing SE principles, tools and techniques. However, typical processes are a barrier to progress. "To break the gridlock, we must establish a degree of cooperation and collaboration with the [SE] community that does not yet exist" [12]. "There is a need to improve the transfer of existing practices and tools from ... [SE] to [SCS]. In addition, ... there is a need for research to specifically develop methods and tools that are tailored to the domain" [41]. "Use of [SE] practices could increase the correctness of scientific software and the efficiency of its development." [42]. The current paper has highlighted perspectives, SE principles, templates and tools as part of a path toward interdisciplinary SE and SCS.

References

[1] D. L. Parnas and P. Clements, "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 251–257, Feb. 1986.
[2] IEEE, "Recommended practice for software requirements specifications," *IEEE Std 830-1998*, pp. 1–40, Oct. 1998.

[3] W. S. Smith, L. Lai, and R. Khedri, "Requirements analysis for engineering computation: A systematic approach for improving software reliability," *Reliable Computing, Special Issue on Reliable Engineering Computation*, vol. 13, no. 1, pp. 83–107, Feb. 2007. [Online]. Available: https://doi.org/10.1007/s11155-006-9020-7

[4] B. Boehm, *Software Engineering Economics*. Englewood Cliffs, New Jersey: Prentice Hall, 1981.

[5] J. Segal and C. Morris, "Developing scientific software," *IEEE Software*, vol. 25, no. 4, pp. 18–20, July/August 2008.

[6] A. N. Johanson and W. Hasselbring, "Software engineering for computational science: Past, present, future," *Computing in Science & Engineering*, vol. Accepted, pp. 1–31, 2018.

[7] J. Segal, "When software engineers met research scientists: A case study," *Empirical Software Engineering*, vol. 10, no. 4, pp. 517–536, Oct. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10664-005-3865-y

[8] S. M. Easterbrook and T. C. Johns, "Engineering the software for understanding climate change," *Comuting in Science & Engineering*, vol. 11, no. 6, pp. 65–74, November/December 2009.

[9] J. Segal, "Some problems of professional end user developers," in *VL-HCC '07: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 111–118.

[10] R. Sanders and D. Kelly, "Dealing with risk in scientific software development," *IEEE Software*, vol. 4, pp. 21–28, July/August 2008.

[11] J. Carver, D. Heaton, L. Hochstein, and R. Bartlett, "Self-perceptions about software engineering: A survey of scientists and engineers," *Computing in Science Engineering*, vol. 15, no. 1, pp. 7–11, January/February 2013.

[12] S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta, "Scientific computing's productivity gridlock: How software engineering can help," *Computing in Science Engineering*, vol. 11, no. 6, pp. 30–39, Nov 2009.

[13] G. Miller, "SCIENTIFIC PUBLISHING: A Scientist's Nightmare: Software Problem Leads to Five Retractions," *Science*, vol. 314, no. 5807, pp. 1856–1857, 2006. [Online]. Available: http://www.sciencemag.org

[14] D. E. Post and L. G. Votta, "Computational Science Demands a New Paradigm," *Physics Today*, vol. 58, no. 1, pp. 35–41, Jan. 2005.

[15] R. Milewicz and E. M. Raybourn, "Talk to me: A case study on coordinating expertise in large-scale scientific software projects," *ArXiv e-prints*, September 2018.

[16] U. Rüde, K. Willcox, L. McInnes, and H. Sterck, "Research and education in computational science and engineering," *SIAM Review*, vol. 60, no. 3, pp. 707–754, 2018. [Online]. Available: https://doi.org/10.1137/16M1096840

[17] L. Hatton, "The chimera of software quality," *Computer*, vol. 40, no. 8, Aug. 2007.

[18] M. Srinivasan, "Investigating common perceptions of software engineering methods applied to scientific computing software," Master's thesis, McMaster University, Hamilton, Ontario, Canada, 2018.

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

[20] T. Maibaum and A. Wassyng, "A product-focused approach to software certification," *IEEE Computer*, vol. 41, no. 2, pp. 91–93, 2008.

[21] W. S. Smith, "A rational document driven design process for scientific computing software," in *Software Engineering for Science*, ser. Chapman & Hall/CRC Computational Science, J. C. Carver, N. C. Hong, and G. Thiruvathukal, Eds. Boca Raton, FL: Taylor & Francis, 2016, ch. Examples of the Application of Traditional Software Engineering Practices to Science, pp. 33–63.

[22] W. S. Smith and W. Yu, "A document driven methodology for improving the quality of a parallel mesh generation toolbox," *Advances in Engineering Software*, vol. 40, no. 11, pp. 1155–1167, Nov. 2009.

[23] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[24] ESA, "ESA software engineering standards, PSS-05-0 issue 2," European Space Agency, Tech. Rep., February 1991.

[25] NASA, "Software requirements DID, SMAP-DID-P200-SW, release 4.3," National Aeronautics and Space Agency, Tech. Rep., 1989.

[26] W. S. Smith and L. Lai, "A new requirements template for scientific computing," in *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes,*

SREP'05, J. Ralyté, P. Ȧgerfalk, and N. Kraiem, Eds. Paris, France: In conjunction with 13th IEEE International Requirements Engineering Conference, 2005, pp. 107–121.

[27] M. Patrick, "Making scientific software easier to understand, test, and communicate through software engineering," in *Software Engineering for Science*, ser. Chapman & Hall/CRC Computational Science, J. C. Carver, N. C. Hong, and G. Thiruvathukal, Eds. Boca Raton, FL: Taylor & Francis, 2016, ch. Examples of the Application of Traditional Software Engineering Practices to Science, pp. 53–88.

[28] W. S. Smith, J. McCutchan, and F. Cao, "Program families in scientific computing," in *7th OOPSLA Workshop on Domain Specific Modelling (DSM'07)*, J. Sprinkle, J. Gray, M. Rossi, and J.-P. Tolvanen, Eds., Montréal, Québec, Oct. 2007, pp. 39–47.

[29] W. S. Smith and N. Koothoor, "A document-driven method for certifying scientific computing software for use in nuclear safety analysis," *Nuclear Engineering and Technology*, vol. 48, no. 2, pp. 404–418, Apr. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1738573315002582

[30] H. M. Şahin, K. Kocatepe, R. Kayıkcı, and N. Akar, "Determination of unidirectional heat transfer coefficient during unsteady-state solidification at metal casting–chill interface," *Energy conversion and management*, vol. 47, no. 1, pp. 19–34, 2006.

[31] F. Benureau and N. Rougier, "Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions," *ArXiv e-prints*, Aug. 2017.

[32] D. H. Bailey, J. M. Borwein, and V. Stodden, *Reproducibility: Principles, Problems, Practices*. New York: John Wiley and Sons, 2016, ch. Facilitating reproducibility in scientific computing: principles and practice, pp. 205–232.

[33] J. P. Ioannidis, D. B. Allison, C. A. Ball, I. Coulibaly, X. Cui, A. C. Culhane, M. Falchi, C. Furnello, L. Game, G. Jurman *et al.*, "Repeatability of published microarray gene expression analyses," *Nature genetics*, vol. 41, no. 2, pp. 149–155, 2009.

[34] C. Ionescu and P. Jansson, "Dependently-Typed Programming in Scientific Computing — Examples from Economic Modelling," in *Revised Selected Papers of the 24th International Symposium on Implementation and Application of Functional Languages*, ser. Lecture Notes in Computer Science, vol. 8241. Springer International Publishing, 2012, pp. 140–156.

[35] D. M. Hoffman and P. A. Strooper, *Software Design, Automated Testing, and Maintenance: A Practical Approach*. New York, NY, USA: International Thomson Computer Press, 1995. [Online]. Available: http://citeseer.ist.psu.edu/428727.html

[36] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. ACM*, vol. 15, no. 2, pp. 1053–1058, Dec. 1972.

[37] W. S. Smith, T. Jegatheesan, and D. F. Kelly, "Advantages, disadvantages and misunderstandings about document driven design for scientific software," in *Proceedings of the Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCE)*, Nov. 2016, 8 pp.

[38] W. S. Smith, Y. Sun, and J. Carette, "Comparing psychometrics software development between CRAN and other communities," McMaster University, Technical Report CAS-15-01-SS, Jan. 2015, 43 pp.

[39] A. Beckhause, D. Neumann, and L. Karg, "The impact of communcation structure on issue tracking efficiency at a large business software vendor," *Issues in Information Systems*, vol. X, no. 2, pp. 316–323, 2009.

[40] W. S. Smith, "Beyond software carpentry," in *2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18)*, 2018, pp. 1–8.

[41] T. Storer, "Bridging the chasm: A survey of software engineering practice in scientific programming," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 47:1–47:32, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3084225

[42] D. Heaton and J. C. Carver, "Claims about the use of software engineering practices in science," *Inf. Softw. Technol.*, vol. 67, no. C, pp. 207–219, Nov. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2015.07.011