Learning to Cook 'ware

A Family Approach

Daniel Szymczak
McMaster University
szymczdm@mcmaster.ca

Spencer Smith Jacques Carette
McMaster University
smiths at mcmaster.ca/carette at mcmaster.ca

Abstract

This is where we will put the abstract of our paper. It will be super-fantastic and make all the reviewers think that this should not only be accepted, but most definitely published.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords Program families, generative programming, documentation, scientific computing, literate programming

1. Introduction

Scientific computing (SC) was the first application of computers. It is still used today for a wide variety of tasks: constructing mathematical models, performing quantitative analyses, creating simulations, solving scientific problems, etc. SC software has been developed for increasingly safety and security critical systems (nuclear reactor simulation, satellite guidance) as well as predictive systems. It has applications including (but not limited to) predicting weather patterns and natural disasters, and simulating economic fluctuations. As such, it is an incredibly important part of an increasing number of industries today.

In the medical, nuclear power, aerospace, automotive, and manufacturing fields there are many safety critical systems in play. With each system, there is the possibility of a catastrophic failure endangering lives. It is incredibly important then to have some means of certifying and assuring the quality of each software system. As Smith et al. [**?**] stated "Certification of Scientic Computing (SC) software is official recognition by an authority or regulatory body that the software is t for its intended use." These regulatory bodies determine certain certification standards that must be met for a system to become recognized as certified. One example of a certification standard is the Canadian Standards Association (CSA) requirements for quality assurance of scientific software for nuclear power plants.

The main goal of software certification is to "... systematically determine, based on the principles of science, engineering, and measurement theory, whether a software product satisfies accepted, well-defined and measurable criteria" [**?**]. As such, certification would not only involve analyzing the code systematically and rigorously, but also analyzing the documentation. Essentially, this means the software must be both valid and verifiable, reliable, usable, maintainable, reusable, understandable, and reproducible.

Developing certifiable software can end up being a much more involved process than developing uncertified software: it takes more money, time, and effort on the part of developers to produce. These increased costs lead to reluctance from practitioners to develop certifiable software [**?**]. However, in our opinion, cost is not the only contributing factor for the developers. As it stands in the field, scientists seem to prefer a more agile development process [**?**]. Typically this would lead to problems maintaining the documentation, meaning that not all aspects of the software would be traceable through the design process and making certification more difficult (if not impossible).

Given proper methods and tools, scientists would be able to follow a more structured approach (while capitalizing on frequent feedback and course correction) to meet documentation requirements for certification as well as improve their overall productivity. This is where our work comes in: our goal is to eat our cake and have it too. We want to improve the qualities (verifiability, reliability, understandability etc.) and at the same time improve performance. Moreover, we want to improve developer productivity; save time and money on SC software development, certification and re-certification. To accomplish this we need to do the following:

1. Remove duplication between software artifacts for scientific computing software [**?**]
2. Provide complete traceability between all artifacts

To achieve the above goals, we propose the following:

1. Provide methods, tools and techniques to support developing scientific software using a literate process
2. Use ideas from software product lines, or program families
3. Use code generation

Section 2 will give a more in-depth look at SC software, specifically focusing on SC software quality (including historical attempts to improve quality), the program family approach, and literate programming. Section 3 focuses on what a literate family approach can achieve, specifically related to software certification; domain knowledge capture; simplification, reusability, and portability; optimization; verification; and how it can incorporate non-functional requirements as well as functional. Finally, section 4 will provide a few concluding remarks.

2. Background

Throughout the history of computing, specifically scientific computing, there have been many challenges towards assuring the quality of software. Many attempts have been made (some successful, others not) at improving the software quality. In this section we will discuss those attempts and challenges, as well as introduce the ideas behind our proposed approach.

2.1 Challenges for SC Software Quality

SC software has certain characteristics which create challenges for its development. We will not discuss them all in depth, however, those of interest to us are the approximation, unknown solution, technique, input output, and modification challenges as described by Yu [**?**].

The *approximation challenge* is a challenge to SC software's reliability. Since real numbers are approximated by floating point numbers on computers, round off and truncation errors can be introduced to a SC software system during computation steps. While any single step may introduce a very small error, they can compound quickly and become unmanageable.

The *unknown solution challenge* is another challenge to SC software's reliability. This challenge arises when SC programs are used to solve problems with unknown true solutions, i.e. there are very limited test cases with known solutions. As such, the accuracy and precision of the software can be hard (if not impossible) to judge.

The *technique selection challenge* is a challenge that comes up often when dealing with continuous mathematical equations. Since these cannot be solved directly (due to computers being discrete), some technique must be chosen which can produce a solution which is "good enough" for the user. Choosing the technique to use is typically left to domain experts as each technique will (not) satisfy certain non-functional requirements.

The *input output* challenge impacts the usability of SC software and typically comes up where considerable amounts of input data are used in the production of large volumes of output. The main issue in this case is the complicated nature of the input data and the output, leading developers to recreate existing library routines (slightly modified) to deal with the nature of the input and output.

Finally, the *modification challenge* tends to come up as requirements change. As SC software is used at the forefront of scientific knowledge, there can be a high frequency of requirements changes. As changing requirements can mean completely modified systems, it poses a problem for scientists: commercial software can be difficult/expensive to modify and noncommercial programs are not often flexible enough to change.

2.2 History of Attempts to Improve Quality

SC software has not seen many focused attempts at improving software quality. Most attempts at improving software quality have been aimed at a broader target than SC specifically. However, many of the methods used have been useful to SC software. Yu [**?**] addresses the issue in-depth, but we will give a brief overview.

The object-orientation (OO) approach aims to increase software quality through the use of several mechanisms. *Encapsulation* ensures information hiding [**?**] by keeping information hidden from the classes that do not require it. *Inheritance* is useful in the OO approach, but not necessarily when applied to SC as many SC problems do not deal with a hierarchical structure. *Polymorphism* can improve reusability by, for example, allowing algorithms to be bound at runtime (though the use of abstract classes). However, for SC software, the algorithm being solved is typically known ahead of time and thus a general program using different algorithms is not necessarily useful.

Agile methods, on the other hand, are one of the most prevalent approaches used by software engineers. It is useful in the context of SC software as it allows for flexibility in dealing with unpredictably changing requirements, on the flip side, if the requirements are fairly stable there is no real benefit to using agile methods. However, many practitioners of SC may claim that their requirements are not stable. This may be true for individual programs, but if we think of the exercise as developing a family of related programs, then the requirements are stable. The experimentation performed by SC developers is an exercise in modifying the variabilities within a program family.

A program family approach is meant to improve reusability. As the program family is reused and retested, it is likely that any defects will be discovered, thus improving reliability. Developing a program family is a general, plan-driven approach. Where agile methods dealt with unpredictably changing requirements, the program family approach is tailored toward projects with stable or predictably changing requirements. The program family is designed such that each

member solves a small set of problems, and we will discuss it in more depth in Section 2.3.

There are several other techniques for improving SC software quality. The use of *libraries*, for one, is a fairly standard technique used in many areas of software development. However, in SC software, libraries can be cumbersome or difficult to use due to the variety of their subroutines. *Component-based development (CBD)* is similar to the library approach, but the components that are being reused are not restricted to subroutines, and CBD suggests techniques for the development of reusable components.

A *problem solving environment (PSE)* is similar to both libraries and CBD, however, it is much more involved. A PSE provides all of the facilities necessary for solving a class of problems. This includes (semi-)automatic solution method selection, advanced solution methods, and the means to include novel solutions. Combining PSEs and libraries can help avoid the difficulties of using libraries independently [**?**]. PSEs have already been used for SC for years, with major examples being Matlab and Maple.

Another technique is *aspect-oriented programming (AOP)*. It is similar to the OO approach mentioned above in that the main focus is separating concerns, however, AOP deals with cross-cutting concerns of a system (ex: logging). Aspects modularize supporting functions and isolate them from the business logic of the main program.

Generic programming is a technique for developing reusable software. Generic programming looks at an algorithm and determines the commonalities between similar implementations of it. From there, a single, generic algorithm is created (through the use of abstractions) which will be able to cover many concrete implementations.

Generative programming is used for automatically creating members of program families. Given a requirements specification, generative programming can produce highly customized and optimized products on demand. The product is manufactured using reusable implementation components and specific configuration knowledge.

Finally, *design patterns* are solutions to recurring problems in software design. Each pattern is both general and reusable. Essentially, design patterns are templates for solving problems (that can occur in many different situations), and are typically found in OO design. As such, design patterns can be used in SC in certain cases where the OO approach is taken.

As far as we know, there has not been an empirical study to measure the effectiveness of any of the above approaches for SC software.

2.3 Program Family Approach

In the previous section, we introduced the idea of a program family approach. To re-iterate: the program family approach should be used in projects with stable, or predictably changing, requirements. The program family approach is reusable, reliable, and easily maintainable.

Reusability of program families is enforced by design. If the product were not reusable, there would be no reason to undertake a program family approach. Reliability stems from the reuse of the common core of the program family; defects are more likely to be found. Finally, since the core assets of the program family are managed together, it is easier to maintain.

In a program family, each member is not a general purpose program. The family members each solve a specific (small) set of problems. This can lead to increased usability, as there is no need to configure a general purpose program for a specific problem.

Determining how or when to apply the program family approach can be difficult. However, Weiss [**?**] proposed a strategy which relies on three assumptions (phrased as hypotheses) which aid in determining whether a domain is suitable for the program family approach. The hypotheses are as follows:

- *The Redevelopment Hypothesis*: Most software development consists of a majority of redevelopment. Existing software systems are modified to create new variations as opposed to entirely new software systems. In most cases, these variations are more alike than different from each other.
- *The Oracle Hypothesis*: Changes to a piece of software over its lifetime can be predicted. Specifically, necessary types of variations of a system can be predicted.
- *The Organizational Hypothesis*: The software may be organized in such a way that predicted changes of one type can be made independently of other types of changes. Essentially, changing at most a few modules in the system for any given type of change.

Many SC programs can be developed as families because they adhere to these hypotheses. With respect to the redevelopment hypothesis, SC software that performs simulations, for example, typically has several of its own assumptions. If these assumptions are modified, then it is likely that the software can now perform a different (hopefully useful) simulation. The same applies to any SC software which solve problems in similar ways: there is reuse of common functionality so it would necessarily be redeveloped in other situations.

Many SC software adhere to the oracle hypothesis since they are equations that model scientific theories which have been studied over many, many years. The theories are stable, so any change to the software is predictable. Changes to the computational design are generally known ahead of time or can be estimated by the domain specialists (scientists).

The organizational hypothesis holds for SC software as well, but perhaps not as strongly as the other two (yet). As we mentioned, a major area of change are the underlying assumptions for programs. These assumption changes for the models are not often strongly dependent, thus few modules would need to be changed to modify the software. However,

there is typically a strong connection between algorithms and data structures in SC software, but separating the data structures from the algorithms is not impossible [**?** **?**].

2.4 Literate Programming

Literate programming (LP) is a programming methodology introduced by Knuth [**?**]. The main idea behind it is writing programs in a way that allows us to explain (to humans) what we want the computer to do, as opposed to simply telling the computer what to do. There is a focus on ordering the material to promote better human understanding.

In a literate program, the documentation and code are kept together in one source. The program is an interconnected web of code pieces, presented in any sequence. As part of the development process, the algorithms used in literate programming are broken down into small, easy to understand parts (known as "sections" [**?**] or "chunks" [**?**]) which are explained, documented, and implemented in a more intuitive order for better understanding.

To extract the source code or documentation from the literate program, certain processes must be run. To get working source code, the *tangle* process would be run, essentially extracting the code from the literate document and reordering it into an appropriate structure for the computer to understand. To get a human-readable document, the *weaving* process must be run to extract and typeset the documentation.

By adhering to the LP methodology, a literate program ends up with documentation that is expertly written and of publishable quality. The code also ends up being incredibly well documented and high-quality. There are several examples of SC programs being written in LP style, and two that stand out are VNODE-LP [**?**] and "Physically Based Rendering: From Theory to Implementation" [**?**] (a literate program / textbook).

3. What is Possible with a Literate Family Approach

A literate family approach is a combination of literate programming, program families, and code generation. The main goals of this approach (mentioned earlier) are to improve qualities (verifiability, reliability, etc.) while improving performance and productivity to save money and time on software development, certification, and re-certification.

To achieve these goals, we begin by taking a literate approach to software development. For our purposes we will be focusing on SC software, but there are many potential domains of application for the literate family approach. The literate approach helps to break down the problems into easily explained chunks, capturing domain knowledge and ensuring high-quality code and documentation.

As we will be looking at SC software, we also take a program family approach to find commonalities between similar implementations of algorithms. Analyzing commonalities allows us to determine the core versus the configuration of an algorithm. Once the core has been determined, we intend to use code generation to implement the program family (and its members) in a simple and straightforward manner. Since the code will be generated from the knowledge captured in the LP-style chunks (which we sometimes refer to as *ingredients*) according to a *recipe*, the documentation for each family member can also be generated from a recipe. Having a family of programs with high-quality documentation for little cost to developers would be an enormous benefit and would aid in certifying the family of programs.

3.1 Software Certification

To certify software there is a need for high-quality documentation. This documentation needs to be created without impeding the work of scientists. One of the major problems with creating documentation for SC software stems from the modification challenge (Section 2.1). As the requirements change, the documentation must be updated. The further scientists are into a project, the greater the effect a change of requirements will have on the documentation, leading to maintainability and traceability issues.

The cost of making changes to the documentation should be reasonable. The best way to keep costs low is to ensure the traceability of the documentation. Traceability allows the developers to track which areas of a project will be affected by a change, thus allowing them to ensure that those areas are updated accordingly.

Depending on the regulatory body and set of standards for certification, many types of documents can be required. Our approach aims to generate all of these documents alongside the code, accounting for any changes made. As the changes will affect ingredients and those ingredients will be used to generate the documentation, there is a guarantee that changes will propogate throughout all of the documents. The following are examples of "default" documents that we want to create (we assume software engineers will be familiar with most, if not all, of these):

- Problem Statement: A description of the problem to be solved, essentially summarizing the purpose of the software. There should be no mention of how the problem is to be solved.
- Development Plan: An overview of the development process and infrastructure. This should cover the documents to be created, which templates (if any) will be followed, internal processes to be employed, coding standards, and other details relevant to how the software will be developed.
- Requirements Specification: The desired functions and qualities of the software should be documented herein. The requirements should cover functionality, goals, context, design constraints, expected performance, and any other quality attributes of the software.

- Verification and Validation plan: Documents how the documentation artifacts will be shown to be internally correct (verification) as well as how to ensure that the right problem has been solved (validation).
- Design Specification: How will the requirements be realized? Document the software architecture and a detailed design of the modules/interfaces to be used. Can be thought of as a programmer's manual for maintenance.
- Code: Implementation of the design specification.
- Verification and Validation Report: Summarize the steps taken for performing verification and validation (this should mirror the V & V plan from above) and include any testing results.
- User Manual: Full instructions on how the software is to be used, beginning with installation instructions and proceeding through detailed examples of the program's use.

The documents listed above share non-trivial amounts of information, which is where traceability comes into play. To make any changes to the documentation, there must be some way to determine how a change will affect **all** of the documents.

This is especially important for (re-) certification, as the documents must be consistent.

In the context of re-certification, if a program was developed using the literate family approach and some changes needed to be made, updating the documentation and submitting it for re-certification would be completely trivial. All documentation would be generated from the newly modified ingredients, according to their existing recipes.

On another note, if a document standard were to be changed during the development cycle, it would not necessitate re-writing the entire document. All of the information in the ingredients would remain intact, only the recipe would need to be changed to accomodate the new standard.

An example would be good here. We could use fuelpin example.

3.2 Knowledge Capture

Capturing scientific knowledge is essential to implementing effective SC software families. Many formulae are commonly shared between a variety of different practical applications.

As an example, consider the conservation of thermal energy equation:

$$CONSERVATIONEQN. \tag{1}$$

This equation is used for thermal analysis of fuel pins, but is also used in applications like solar water heating tanks [].

From the fuel pin example (again), if we look at the equation for the gap conductance from Figure 1 and the equation for heat transfer coefficient between clad and coolant

Label	h_g
Units	$ML^0t^{-3}T^{-1}$
SI equivalent	$\frac{\text{kW}}{\text{m}^{2\circ}\text{C}}$
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 initial gap film conductance k_c is the clad conductivity

Figure 1. SRS data definition of h_g

Label	h_c
Units	$ML^0t^{-3}T^{-1}$
SI equivalent	$\frac{\text{kW}}{\text{m}^{2\circ}\text{C}}$
Equation	$h_c = \frac{2k_c h_b}{2k_c + \tau_c h_b}$
Description	h_c is the effective heat transfer coefficient between the clad and the coolant τ_c is the clad thickness h_b is initial coolant film conductance k_c is the clad conductivity

Figure 2. SRS data definition of h_c

from Figure 2, we can see that there is common knowledge necessary for calculating both of these values (i.e. k_c and τ_c). Now, rather than redefining the common terms in these equations in each document where they appear, we consider each unique symbol as its own ingredient. Both h_g and h_c are also ingredients, thus they need only reference the ingredients used in their equations to express all of the requisite knowledge for implementation.

For the documentation related to h_g and h_c, the recipe for each document will fill in the appropriate details related to any and all symbols that appear in the equations, which can be seen in the "Description" section of Figures 1 & 2. Now if we needed to update the description of any symbol, we would simply update it in one place (its ingredient) and the recipe would handle propagating the change through all of the documentation and code.

From the above example it should be clear that this approach allows for the development of a library of artifacts (ingredients) related to SC software that capture domain knowledge and can be reused in multiple contexts while remaining maintainable and traceable. Another benefit of creating such a library of artifacts is it would combine many sources of knowledge while keeping the notations and terminology consistent and ensuring that all underlying assumptions are well documented.

3.3 "Everything should be made as simple as possible, but not simpler." (Einstein quote)

Currently there exist many powerful, general commercial programs for solving problems using the finite element method. However, they are not often used to develop new "widgets" because of the cost and complexity of doing so. As it stands, engineers often have to resort to building pro-

totypes and testing them instead of performing simulations due to a lack of tools that can assist with their exact set of problems.

A literate family approach could change that by generating members according to the needs of the engineers. For example, if an engineer is designing parts for strength, they could have a general stress analysis program. This program could be 3D or specialized for plane stress or strain, depending on which assumption would be the most appropriate. The program could even be specialized further, to be customized to the parameterized shape of the part that the engineer is interested in. This new specialized program can expose only the degrees of freedom that the engineer can change (ex. material properties or specific dimensions of the part), making the entire simulation process simpler and safer.

3.4 Optimization

- Connect optimization with analysis. Optimization requires running multiple analysis cases. Code generation can be used to build an efficient model that has just what is needed, and no more. As the optimization searches the design space, new models can be generated. - An optimization problem for a part where the shape and constitutive equation are degrees of freedom, cite family of material models (SmithMcCutchanAndCarette2014)

An example would be good here. Unfortunately, I do not have one that has previously been worked out.

3.5 Verification

When it comes to verification, requirements documents typically include so-called "sanity" checks that can be reused throughout subsequent phases of development. For instance, the requirement would state conservation of mass or that lengths are always positive. The former would be used to test the output and the latter to guard against some invalid inputs.

With a literate family approach, these sanity checks can be ensured by the knowledge capture mechanism. Each ingredient can maintain its own sanity checks and incorporate them into the final system to make sure that all inputs and outputs (including intermediaries) are valid.

- computational variability testing, from Yu (2011), FEM example - usual to do grid refinement tests - same order of interpolation, but more points - code generation allows for increases in the order of interpolation, for the same grid - Yu discusses in section 6.3 of her thesis

3.6 Incorporating Non-Functional Requirements in a decision support system for selecting the best design options

A literate family approach will necessarily be able to create multiple family members that can solve the same problem in slightly different ways (ex. different algorithm choices), so how should a decision be made between them? Traditionally when designing software, this is where the engineers would compare the non-functional requirements of the system to the different implementations, then make a decision based on how those requirements are being satisfied.

The decision making process can be performed through the literate family approach by using the Analytic Hierarchy Process (AHP) [**?**]. AHP is a method for comparing attributes by using a ratio scale to prioritize these attributes. The priorities are then determined by a series of pair-wise comparisons between attributes. AHP has had great success in decision analysis, and as such it is the method we have chosen for ranking non-functional requirements.

Once we have decided on the relative rankings of our different non-functional requirements, we can analyze the family members that solve our problem and select the one which does the best job of incorporating the non-functional requirements in their priority order.

4. Concluding Remarks

Concluding remarks.

References

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.