McMaster Centre for Software Certification

Certification of Scientific Computing Software

Gonzalo Sánchez, Spencer Smith and Ned Nedialkov
Certification of Scientific Computing Software

Gonzalo Sánchez\textsuperscript{1*}, Spencer Smith\textsuperscript{2} and Ned Nedialkov\textsuperscript{2}

\textsuperscript{1} Facultad de Ingeniería y Tecnologías, Universidad Católica del Uruguay, Montevideo, Uruguay  
gsanchez@ucu.edu.uy  
\textsuperscript{2} Department of Computing and Software, McMaster University  
Hamilton, Ontario, Canada  
\{smiths,nedialk\}@mcmaster.ca

Abstract. This document provides the authors’ view on certification of scientific computing software (SCS). Initially, the relevance of SCS is underlined, which sets the motivation for attempting the certification of SCS products. Following this, the semantics of terminology commonly used in the context of software certification is clarified, given the ambiguous and even contradictory uses in the literature. In particular, the key concepts of software verification and validation are presented and their relationship to certification is discussed in the context of SCS. Thereafter, this report describes approaches to software verification and validation the authors believe are most promising for SCS—formal specification, literate programming, document-driven development, code reviews, the method of manufactured solutions, software mutation, fault injection, and uncertainty quantification. The product-based approach for certifying software qualities is then presented, alongside the rationale for advocating it.

1 Introduction

Scientific computing software (SCS), which consists of using computer tools to simulate continuous mathematical models of real-world systems, so that we can better understand and predict the system’s behavior, is not usually regarded as safety-critical. The term safety-critical is typically reserved for real-time control software, where failure may result in injury or death to human beings, such as with control systems for aircrafts, or for nuclear power plants. However, examples such as the 1991 failure of a Patriot missile to hit an incoming Scud missile \cite{31} show that SCS can be an important component of safety critical software. Moreover, if we remove the real-time control aspect from the definition of safety critical, then there are many examples where SCS software is used for decision making that has a significant impact on health and safety, such as modeling of environmental hazards, weather forecasting, medical image analysis, design of treatment protocols for radiation therapy, and structural analysis. Given the importance of the decisions that are made using SCS software, it is vital that techniques and methodologies be available to certify this class of software.

SCS presents several challenges for certification and the associated Quality Assurance (QA) process. One of the most significant challenges is the frequent

\* Most of this work was done when G. Sánchez was with the McSC at McMaster University.
Certification of Scientific Computing Software

lack of a known, true solution against which computed results can be compared. Since SCS is often used when a physical experiment cannot be executed, or when the risks or costs are exceedingly high (such as to simulate the collision of galaxies, or the thermodynamics of a nuclear reactor core), there might be no possibility of validating the numerically calculated results against reality. Even for common mathematical problems, such as solving systems of linear equations or ordinary differential equation, there is often no test oracle available that can provide a true result for comparison purposes.

Another challenge for certification of SCS software is the complexity of the mathematical models that are simulated. This complexity creates many opportunities for errors. For instance, a documented conceptual model of real-world system may include implicit assumptions, which mean the software is invalid for certain inputs that its implementers and users are unaware of. In the future, the software may inadvertently be used for cases that the original designers never intended and for which the results are likely to be incorrect. The complexity of the mathematical models in SCS also causes difficulty for the analysis of how errors may propagate through the calculations, even when a numerical model may be unstable or very sensitive to small variations in problem parameters. Additional difficulty arises because compiler optimizations can further exaggerate the instability and sensitivity of calculations \(28\). The complexity of the SCS mathematical models means that it is often difficult to judge, and rarely possible to prove, the accuracy of computed results.

A further problem with SCS certification is that traditionally it has been developed by scientists and/or engineers with no formal training in software engineering. Therefore, even when software produces accurate results, it is often the case that it fails to support other software qualities such as verifiability, understandability, maintainability, portability and reusability. Failure to address these issues causes conceptually sound software solutions to be reinvented over and over again, thus aggravating the reusability and reliability concerns, and perpetuating the lack of interest in maintainability and portability.

We present in this report a synopsis of software certification techniques and assess their scope and suitability for certifying SCS. In this context, we advocate a product-based approach to QA of SCS, as opposed to the usual process-based approach that is currently applied to general software. As it is argued below, the product-based approach is a prudent choice given the risks of improper design, implementation, or usage of SCS.

This document is organized as follows. Section 2 presents key terminology in the literature on software certification. Section 3 discusses the SCS development process. Section 4 presents software constructive techniques and verification and validation methods. Section 5 introduces the product-based approach to software certification advocated by the McMaster Centre for Software Certification (McSCert). Concluding remarks are in Section 6.

2 Terminology and Semantics

The terminology in the software certification literature is not always consistent. Here, we clarify the meaning of the terms that are used throughout this document, namely, program (§2.1), software (§2.2), correctness, error, fault, failure (§2.3), rigor, formality (§2.4), precision, accuracy (§2.5), verification, validation (§2.6), and certification (§2.7).
2 Terminology and Semantics

2.1 Program

The definition of a program that best suits our needs is given by R. Hehner [15, p. 41]. Concisely summarized:

A program is the specification of the behavior of a computer, for which an implementation exists such that the target computer can execute it.

Against the usual envisage of a program as source code, this definition involves two parts, specification and implementation. That is, it requires that an executable implementation is obtainable from source code, to call it a program, but does not require an implementation to be part of the program.

The rationale for explicitly mentioning implementation in the definition of a program resides in the fact that the specification of a behavior is a completely different matter from the actual behavior of the computer, which might ‘misbehave’ for various reasons (e.g., faulty compiler or hardware failure) and, except for a hardware failure, the cause for deviations from specified behavior lies within an implementation. Note that, except when explicitly stated, we do not include hardware failures in our discussion on software certification.

2.2 Software

We use the term software without further qualifiers to refer to a documented program, where the documentation encompasses all other information provided besides source code, as a specification of the computer’s intended behavior, e.g., software requirements document, technical and/or user’s manuals, etc. By software documentation, we refer to the documentation included as part of software’s specification.

By software product we mean the program specification–implementation pair that defines and determines the behavior of a target computer (under the assumption of non-faulty hardware). A software product includes source code, object/executable code, and documentation.

We use the phrase software package to refer to a larger set of objects related to a software product, which may include any third party components required for distribution and deployment. We include in the ‘software package’ the related physical media for the software product, e.g., printed documentation and storage media containing the program.

The term software tools refers to any software product that participates in the development of the software of interest (e.g. compilers, source code version control programs), but that is not part of the software product itself.

2.3 Correctness, Errors, Faults, Failures

A failure is said to occur in a computer system if the output produced by a particular implementation of a program is unacceptable compared to that of its specification. Failure is defined relative to a specification, but nothing is implied here about the specification being correct or complete. Also, failure is usually considered against the specified functional requirements. The principle functional requirement for SCS is the requirement to solve the governing equations for the problem of interest. Non-functional requirements, such as usability, are harder to quantify.
If an implementation of a given specification can lead to a failure, then this implementation has a fault, and it is therefore not logically equivalent to the given specification. A fault is latent until it manifests itself as a failure [24, p. 7].

Faults are caused by errors. Errors can derive from mistakes at the specification level (omissions, inconsistencies), or can be generated as a consequence of failures in the tools used to produce an implementation, or failures in the computer used to execute the implementation (e.g. a faulty storage device or memory cell that corrupts the program implementation) [24, p. 7].

This calls for methods and technologies that assist in the design of a specification such that the chance of making an error is minimized, and ideally, avoided, as well as methods and technologies that assist in verifying that an implementation is free of faults. When verification is via testing, any latent faults may or may not be caught by testing techniques, depending on whether the fault causes a failure during testing.

We note that any approach that only checks for errors at program’s specification level of abstraction (i.e. at source code level) cannot uncover any latent faults caused by errors in the tools and libraries used to implement the program’s specification. This is one of the motivations for promoting a product-based approach to software certification.

The terms fault and failure, as defined above, match their use within a software verification context (see §2.6), and do not clash with any usage outside that context. However, the terms error and correctness have different meaning depending on the context. As defined above, they correspond with the use in a software verification context, where correctness is tantamount to a formal proof. However, in software validation (see §2.6), correctness refers to the suitability of the conceptual model for a problem, and then error refers to conceptual mistakes pertaining to the corresponding application domain. Also, error refers to uncertainty in the context of measurements. The reader should infer by context the appropriate semantics for these terms.

2.4 Rigour and Formality

According to Ghezzi et al [12, pp. 42–44], any sound engineering design process follows a sequence of well-defined, precisely stated, and supposedly sound steps. The methods followed, and the techniques applied at each step, are usually a combination of theoretical results, empirical adjustments, and rules of thumb based on past experience. Whenever the blend of these factors results in a systematic approach that can be easily explained and applied in other circumstances—that is, when it can give way to a methodology—then that process is indeed a rigorous approach to problem solving.

The main difference between rigor and formality is that the latter does not allow intuition in any step of the procedure under consideration, whereas the former admits intuition, when it is judiciously applied.

As an example of rigor versus formality, consider mathematics textbooks [12, pp. 42–44]. They are rigorous, but seldom formal. Proofs of theorems are carefully presented sequences of deductions, but each of these deductions relies on the reader’s intuition to convince him/her of the validity of a step, instead of referring to a list of axioms and rules of inference, which systematically applied to the initial symbols on the page generate a deduction chain. Even the language used in textbooks is not a formal language, but a blend of natural language and mathematical notations.
2.5 Precision and Accuracy

Accuracy expresses the quality of a result, and it is stated typically as a relative or absolute error. Hence, stating accuracy requires comparison to a ‘correct’ answer, which is usually not known.

Precision in SCS is related to the number of digits used for computation. Precision in most cases is IEEE double, sometimes IEEE single. Multiple-precision calculations are done in software through a multiple-precision package.

We note that increasing the precision does not guarantee increase in accuracy. However, decreasing the precision results in reduced accuracy.

2.6 Verification and Validation

The terms verification and validation are often used interchangeably, thus leading to inconsistencies throughout the software engineering literature. We start by looking at the IEEE Standard Glossary of Software Engineering Terminology [18]:

Verification (1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

The IEEE Standard Glossary of Software Engineering Terminology provides a second definition for verification: “(2) Formal proof of program correctness” [18, p. 81]. While this definition matches exactly the semantics advocated in [37] for the term verification, it would require additional terminology to address all other analysis techniques that are not rigorous (in mathematical sense) or formal (in mathematical-logic sense). We adopt here the broader definition (1) [18, p. 81].

Validation The process of evaluating a system or component during or at the end of a development process to determine whether it satisfies specified requirements [18, p. 80].

By the above definitions, verification is about checking consistency of each phase in a development process, whereas validation refers to checking requirements compliance for the complete development process. A verified system is not necessarily valid, because verification tests and techniques cannot decide whether the requirements specifications are appropriate for the problem that the software is intended to solve.

In practice, verification is seldom carried out without a subsequent validation process, and thus they are often referred to together as forming an overall process:

Verification and Validation (V&V) is the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements [18, p. 81].

Unlike in the independent definitions of verification and validation, the definition for V&V explicitly asks for a set of complete and correct requirements for a system or component. Yet, in practice, obtaining complete and correct requirements may be a difficult goal to achieve, for which the only viable method in
Certification of Scientific Computing Software

The long term seems to be, in our opinion, well-structured, and well-maintained documentation.

In conclusion, a pragmatic view of V&V can be summarized as:

- verification: are we building the software right?
- validation: are we building the right software?

2.6.1 SCS Verification and Validation. Given that virtually all scientific software has a mathematical model at its core, following Boehm (1981) and Blottner (1990), Roache provides an intuitive interpretation of what is addressed by verification and validation in SCS [37, p. 23]:

- verification: are we solving the equations right?
- validation: are we solving the right equations?

That is, verification processes for SCS try to assess the correctness of a software product, without judging the usefulness of what is being predicted by it. However, validation processes for SCS try to assess the suitability of the implemented theoretical model as a representation of reality from the perspective of its intended uses.

We emphasize here that most verification techniques address functional requirements of the software. Non-functional requirements are normally system-wide qualities of the software product, and by their nature, sometimes difficult to specify precisely or at least in a measurable manner. Usability, for example, is a software quality inherently difficult to quantify. Non-functional requirements satisfaction is perhaps an overlooked issue in SCS verification, with the notable exceptions of accuracy and performance. Therefore, non-functional requirements are harder to quantify, and thus, less amenable to checking within automated tests frameworks.

We also note that validation activities for SCS typically compare computed results to results from real-world experiments. This requires that experiments are feasible, and therefore, they have to be well-posed validation experiments, for experiments themselves can be inadequate, incomplete, and introduce additional sources of errors.

2.7 Software Certification

Verification and validation are activities implicitly assumed to be part of a software certification process. However, certification—considered as a QA process—also incorporates other requirements, such as documentation requirements, version control, and the QA system itself [37, p. 34].

In fact, according to Roache [37, pp. 33–35], software certification and software QA are two different designations for the same activities, which he associates with engineering management. Since in practice certification usually implies an official recognition by an authority or regulatory body that a certain standard is met, the connection with an engineering management activity seems to be well grounded. That is, software certification is an an engineering management activity towards QA.

The point of certification is to ascertain whether the product for which a certificate is being sought has appropriate characteristics [26, p. 91]. This implies asserting facts that determine whether the characteristics of a product are
appropriate. Hence, a certification process should be a measurement-based activity, and the facts should be measurable entities. Therefore, the goal for software certification should be to: “...systematically determine, based on the principles of science, engineering and measurement theory, whether a software product satisfies accepted, well-defined and measurable criteria” [13, p. 12].

3 SCS Development Process

There is a key distinction between mathematical model and computational model [10, ch. 2]. These two models, which lie between the problem to be solved and the program that implements a solution on a computer, are derived as follows.

Real World → Mathematical Model. Relevant physical magnitudes are selected and irrelevant ones are discarded. By applying theories pertinent to the application domain, mathematical relations among these variables are obtained, which for simplification purposes can be approximated (e.g. linearization, selective term eliminations, etc). This results in a mathematical model for the problem under study.

Mathematical Model → Computational Model. A discretization of the usually infinite and continuous mathematical model leads to a finite dimensional formulation. Further specifications on how to solve the resulting discrete equations result in the computational model for the problem. This computational model is the one to be coded for implementation on a computer system.

Computational Model → Software Product. The computational model is coded in a suitable programming language, and an implementation for a target computer system and operating environment is generated.

There need not be a real-world problem underlying an SCS product. For example, a linear system solver, a fast Fourier transform (FFT) engine, or a partial-differential equation (PDE) solver are all examples of non-trivial SCS that are key components of larger software products, but they do not originate directly from real-world applications.

We digress now from the aforementioned SCS development phases to consider the development process abstractly, as a generic design process that starts with design specification and advances stepwise until implementation. For that purpose, we first observe that in a design process:

- every step from specification to implementation is a design refinement stage, where the level of abstraction decreases, and the description detail increases; this is a synthesis activity;
- every step in from implementation to specification is a design abstraction stage, where the level of abstraction increases, and the description detail decreases; this is an analysis activity.

This suggests the design cell concept (Figure 1). A design cell is a specification–implementation pair of descriptions, between which a synthesis phase generates an implementation from specifications, and an analysis phase compares the implementation against specifications. Any design abstraction (analysis) step is
verification of the corresponding design refinement (synthesis) step. A successful completion of both processes therefore constitutes a design–verification cycle.

We observe that the design cell structure matches our definition of software product in §2.2. We also observe that:

- Any design process can be viewed as a stack of design cells, where the implementation of a design cell in a higher level of abstraction works as a specification for the design cell in the lower level of abstraction.
- Any design process can be summarized from specification to implementation as a single design cell.

Therefore, we can model the entire software design process by this structure, as depicted in Figure 2.

The design process as a stack of design cells allows us to describe intermediate steps in a complete process. In an ‘ideal world’ a successful verification stage that spans more than one level of abstraction would imply that the intermediate levels have been automatically verified. Conversely, independent verification of two adjacent levels of abstraction would imply that the aggregate of these results is verified. Therefore, regardless of how many intermediate steps are defined in the process, a chain of successful design–verification cycles from implementation to specification would, ideally, yield a verified specification-implementation pair. Figure 3 presents the aforementioned situation in a more convenient way.
Figure 3 synthesize from level 1 to 3 could be either accomplished in a single step $s[1–3]$, or as two successive steps $s[1–2]$ and $s[2–3]$. In the opposite direction, if level 4 can be directly verified against level 1, then the verification process is the step labeled $a[4–1]$. If the result of such verification $a[4–1]$ is successful, then all of $a[4–3]$, $a[3–2]$, and $a[3–1]$ would implicitly be successful verification steps.

Figures 2 and 3 show abstract structures, applicable to any design process. In particular, the waterfall model (advocated in software engineering) can be trivially mapped onto this structure, if each derivation in the design process in the waterfall model is associated with a synthesis phase, and any verification with an analysis phase.

Figure 4 extends the idea presented in Figure 3 to include the application domain from where the design specification is derived, and the domain where the design implementation is interpreted, deployed or used.

The application domain $A$ in Figure 4 is typically a real-world situation, from which the specification 1 is derived, and domain $B$ is an instance of computer system’s behavior, where the implementation of 3 will intervene. For example, a scientific software development process following Einarsson et al.’s modeling stages can be mapped onto the structure in Figure 4 using the following assignments:

$$A \leftarrow \text{Application domain}$$
$$1 \leftarrow \text{Mathematical model}$$
$$2 \leftarrow \text{Computational model}$$
$$3 \leftarrow \text{Software product}$$
$$B \leftarrow \text{Computer behavior}$$

In Figure 4 we have used several double-headed arrows. Considering the above assignment for Figure 4, the double-headed arrow for $s[A–1]$ is used to
indicate that the application domain experts are those in charge of the development of a mathematical model. The double-headed arrows in the upward direction in Figure 4 indicate validation processes, since these are against the real world, and hence can only be carried out by application domain experts. It is unlikely, though, that a validation activity could check a software product against reality (path a[3-A]) or the computational model against reality (path a[2-A]). Probably only the mathematical model and the results produced by a software product for a particular run can be validated in practice (paths a[1-A] and a[B-A], respectively).

As another example of mapping, we present the document-driven design for SCS advocated in [39]. In fact, the document-driven design emphasizes the distinction between mathematical model and computational model. The software requirements specification document describes a mathematical model (plus non-functional requirements) and the module guide and module interface specification documents describe the computational model design (see also §4.1.4). This mapping requires assigning the following meanings to the structure’s elements:

A ← Application domain
1 ← Software Requirements Specification
2 ← Module Guide
3 ← Module Interface Specification
B ← Program

As a further example, if an extra step 4 is added to the diagram in As a further example, if an extra step 4 is added to the diagram in Figure 4, an appropriate mapping onto the resulting structure could depict the preprocessing, compilation, assembly and static-linking stages for languages such as, e.g., C/C++, Fortran.
We relate now verification and validation to the waterfall model, using the abstract design process of Figure 4.

**Validation** For SCS development process mapped onto the abstract design process of Figure 4, all analysis steps coming from any level under, and including the specification level, that go upwards to the application domain level (depicted by double-headed arrows), represent validation activities. Their distinctive signature is that it can only be performed by application domain experts, which for the case of SCS are scientists and/or engineers working in the corresponding application domain.

**Verification** For SCS development process mapped onto the abstract design process of Figure 4, all analysis steps coming from any level under, and including the specification level, that go upwards to any other level, except the application domain level (depicted by single-headed arrows), represent verification activities.

### 4 Verification & Validation Methods

Scientific computing software presents three difficulties for certification, as discussed in §1 and as repeated below:

1. A ‘test oracle’ is frequently lacking.
2. Complex mathematical models are often simulated, which means that in many cases important assumptions are left unstated and that error analysis can be extremely difficult.
3. From a software engineering perspective, there is often the difficulty of dealing with software products that, due to their design structure and lack of proper documentation, are difficult to understand, verify, maintain, and reuse.

As a natural consequence of the aforementioned fundamental difficulties:

> “It is rare that correctness of scientific software can be rigorously demonstrated. Instead, the verification and validation processes provide a series of techniques, each of which serves to increase our confidence that the software is behaving in the desired manner.” [10, p. 23]

However elusive rigorous demonstrations of correctness might be for SCS, attempts have been made using techniques and methods developed specifically for numerical analysis and through adapting methodologies employed for more general software systems. The discussion below first focuses, in §4.1 on methodologies that support creating correct software. The next subsections, §4.2 and §4.3 provide an overview of methodologies for assessing the quality of already built systems through verification and validation techniques, respectively. The methodologies discussed here are not independent; often there is overlap between them. Building confidence in a software product typically involves multiple methods and techniques.
4.1 Constructive Techniques

We briefly describe some design approaches that aim to design out any errors that can lead to faults and potential failures in a software-intensive system. We believe these approaches follow the line advocated by sound engineering judgment, which insist that an up-front investment in a careful design results in much more reliable systems than those obtained by detecting and eliminating faults on a sloppy design.

4.1.1 Formal specification

Formal specification is probably the most well-known constructive technique for systems, particularly, software-intensive systems. Formal specifications require the use of languages with a mathematically defined syntax and semantics, which makes functional or behavioral properties easier to specify formally than non-functional or non-behavioral requirements (e.g. performance, usability, reliability, etc).

One appeal of developing formal specifications is that they enable formal analyses of the relationship between the specifications and the software source code, thus laying the path to analysis automation. Formal specifications can facilitate the generation of test cases (see [16]).

Having a formal specification is also a good starting point for source-code generation, sometimes referred to as automatic programming or generative programming. Code generation consists of having one program write the source code for another program. Examples of generative programming for scientific computing can be found in [1], [6], [7] and [11].

“Unfortunately, formal specifications are difficult to write, especially for large systems. In addition, formal specifications do not cope well with the uncertainties of floating-point arithmetic. Thus, they have rarely been employed in this context. A notable exception is the formal verification of low-level floating-point arithmetic functions.” [10] p. 24

4.1.2 Program Families

The main motivation for abstraction is to reduce the amount of detail that designers must handle, hence increasing the understandability and reliability of the design. But abstraction also allows broadening the scope of the potential design, and thus enables thinking not just of programs but of program families — a set of closely related programs that share common core assets and a managed set of resources that facilitate systematic instantiation of variabilities.

In the program family approach, a commonality analysis among the potential members of the family can be used as a basis for defining a Domain Specific Language (DSL). This DSL is easier to use to specify the program’s behavior than a general purpose programming language. Using appropriate generative programming techniques, program family members can be automatically generated from a specification provided in the DSL.

As argued in [35], a program family strategy improves reusability and usability of software, which ultimately contributes to its reliability. Combined with the possibility of code generation from this higher level of abstraction, the program family approach promotes a correct-by-construction design.

A classical reference for the topic is [32]; more recent work on the subject can be found in [8], [36] and [42]. Program families for scientific computing are discussed in [4], [5] and [43].
4 Verification & Validation Methods

4.1.3 Literate Programming Another constructive technique is Knuth’s *Literate Programming* [21] which advocates a departure from the traditional programming approach.

In the traditional programming approach, the programmer writes code containing comments to document it, and eventually produces separate documentation for the program. In the Literate Programming approach this is reversed: the programmer writes documentation containing the code. The idea is that the programmer produces a *document* in which he or she explains to human beings what is expected as the computer’s behavior, and where sections of the program can be inserted accordingly to the unfolding explanations. The program can then be described in the order that makes it easier for its understandability, rather than in the format required for the code compilation process.

Literate Programming requires the combination of two languages, a *document formatting language* and a *programming language*, plus tools that can extract either the documentation or the code from the file where both are simultaneously described, to produce the corresponding software documentation file and software source code files. The original choices by Knuth were \( \TeX \) and Pascal, but there are currently many other combinations of \( \TeX \) and \( \LaTeX \) with other programming languages.

The appeal of the literate programming approach is as follows:

- Given that the emphasis is in documentation, the long standing problem of incomplete and inadequate documentation should be greatly minimized.
- Since software documentation and software source code are extracted from the same file, the chances of these becoming unsynchronized are diminished.
- Given that the emphasis is on adequately explaining to human beings what each part of the code is intended for, this approach greatly facilitates peer review of the software documentation and software source code. After all, peer-review is the way that mathematical proofs are typically validated. Therefore, literate programming could reach the same degree of rigor as mathematical proof.

One example of Literate Programming in scientific computing is provided in [29]. Another example of Literate Programming applied for a non-trivial scientific software product is [35].

4.1.4 Document-Driven Development The document-driven approach forces the documentation set for the software product to ‘fake a rational design process’, as advocated by Parnas [33]. It explicitly requires the developers to document all assumptions and theoretical models underlying the software product, and to include explicit traceability matrices between these and the software requirements, software goals, software structure, and software implementation. Thus, all design decisions end up documented in a hierarchical and organized manner, which significantly improves the software qualities of understandability, maintainability, and verifiability.

An example of document-driven SCS is presented in [39]. Following this approach, a complete documentation set consists of:

**Software Requirements Specification.** A document that specifies program requirements, including:

- Definitions for the terminology and symbols used in the complete documentation set.
Certification of Scientific Computing Software

- An index of all requirements, functional and non-functional.
- A general system description, including the expected system context, expected system constraints, and expected user’s characteristics.
- The description of the problem that the program solves, and an explicit list of the program’s goals (what exactly the program is expected to achieve and output).
- The theoretical model or models (conceptual model or models) that the program implements for the solution of the problem it tackles, explicitly including a list of all the assumptions that the conceptual model makes, a list of all the data definitions, and a list of all the numerical constants that the system uses for each model.
- A traceability matrix between specifications, program goals, assumptions, data definitions, and theoretical model(s).
- A list of possible changes in the requirements (facilitates ‘design for change’)

Modules Guide. A document that describes what the modules that compose the program are intended to do, and the ‘uses’ relationship between them.

Modules Interface Specification. A document that describes the syntax and semantics of each access function for each module, but without any reference on how the functions are implemented. Formal mathematical specification is encouraged at this point.

Source Code Listing. The listing of all the software source code.

Verification & Validation report. This report should be added if the program was subject to relevant tests.

We believe that the document-driven approach can integrate the aforementioned program families, formal specifications and literate programming approaches:

- The commonality analysis that enables the program families approach is naturally admitted by the document-driven approach, and in fact, the latter can help in spotting a potential program family by requiring a careful consideration of assumptions and theoretical models underlying the intended software product, and the traceability between these and the projected software structure.
- Formal specifications can be introduced as needed at the software requirements specification stage, or even at the module interface specification stage.
- Literate programming could be applied at the module interface specification stage to intersperse the actual code fragments within the lowest level algorithmic specifications.

This way, a single file (or a set of files that conform a single ‘logical’ documentation file) would then contain all documentation and source code, organized in such a way that the information can be accessed in an agile way by peer-reviewers, maintainers, testers, and even users of the software product.

4.2 Verification Techniques

We now delve into software verification techniques, which aim at detecting all latent faults to enable their removal. There are several software verification techniques, but they all fall into two major categories depending on whether a technique requires execution of the software product under verification, or
4 Verification & Validation Methods

not: *dynamic analysis* and *static analysis*, respectively. Static analysis considers all execution paths in a program, whereas dynamic analysis may not consider them all. Dynamic analysis is frequently referred to as *dynamic testing*, for the above reason.

Verification techniques can also be classified according to the extent which implementation details are required to be known: *black-box testing*, *grey-box testing*, and *white-box testing*, in increasing order of needed knowledge of internal details.

For the particular case of scientific computing, Einarsson *et al* [10, sec. 2.6] further classify verification techniques into two categories, *code verification* and *solution verification*, corresponding to the model against which an implementation is being verified.

4.2.1 Static Analysis Static analysis refers to any software verification technique that does not require the execution of a program implementation to carry out the analysis. Below, we discuss four static analysis approaches. All of the approaches fall into the white-box testing techniques, since the software source code has to be available for the analyses.

*Code analyzers.* Code analyzers are software tools that typically examine a software product’s source code to detect potential errors. These tools’ analyses are carried out in terms of programming language constructs like functions, statements, and variables.

Code analyzers can examine binary code, which can either be in object code form or executable code form, or even in an intermediate representation form such as byte-codes for a virtual machine (VM), like the Java VM. Analysis at the binary level involve machine entities such as instructions, registers and memory locations, and unlike source code analysis, it is beyond a human reviewer’s capabilities. An advantage of binary analysis is that the original source code is not needed, so it can deal with system and third party libraries, which frequently are provided solely in binary format.

*Code reviews.* Code reviewing consists in having the software source code examined by humans (as opposed to its examination by computers). It is often conducted as peer reviews, and can be viewed as a specific case of software inspection.

Given that the reliance on mathematical proofs actually depends on peer reviews, it is surprising that code reviews are infrequent [19]. The approach in [19] proposes a method to facilitate the adoption of software inspections—including code reviews—by associating these with software documentation work.

It is worth noting that both literate programming and the document-driven approach, introduced as constructive techniques in §4.1, can significantly contribute to the static analysis of software source code. The document-driven approach accommodates overall software inspections, and literate programming facilitates code reviews—particularly for SCS, given the nuances inherent to numerically-oriented code that require precise, comprehensive documentation.

*Model checking.* A finite model is built, such that every property of interest to verify in the software can be checked in the model instead (thus the designation, model checking). For model checking, an exhaustive test on the model is carried
out, and given that the model is finite, the test is guaranteed to terminate. However, for
the case of scientific software, model checking might not be a good fit given the enormous
number of cases that need to be considered when floating point numbers are involved.

Theorem proving. Provide a proof that the formal model of the system satisfies the for-
amal model of the specification.

The last two approaches to static analysis belong to the class of formal methods. As
pointed out in §4.1.1 formal methods can be used for verification. Although in general the
correctness problem is undecidable, formal analysis has been successfully applied for
software verification with either restricted goals, or in particular cases.

4.2.2 Dynamic Analysis Dynamic analysis, or testing, encompasses all the software verifica-
tion techniques that require the execution of the software implemen-
tation. We present an overview of several dynamic testing tools and
techniques. We do not emphasize conventional black-, grey- or white-box testing,
since these topics are covered in many texts on testing of general purpose soft-
ware ([12], [34]).

Debugging tools. Debuggers are possibly among the most well-known tools for
dynamic analysis. Ubiquitous debuggers in the Unix world are gdb (the standard
debugger for the GNU software system) and dbx (the typical debugger for Solaris,
AIX, IRIX, and BSD Unix systems). These tools basically allow the execution of
the target program in a controlled environment. Current microprocessors offer
hardware support for debugging, which some tools use to their advantage for
single-stepping or setting and checking execution breakpoints and values.

The controlled environment provided by a debugger is usually at odds with
real-time software, or embedded systems that interact with the real world and
must deal with random, asynchronous events. Analyzing parallel applications
with debuggers is a challenging task due to the added communication dynamics
of threads or messages, although debuggers tailored to such kind of applications
are becoming available and improving notably.

SCS does not usually have real-time constraints, but current trends are towards
parallel computation, particularly aimed at High-Performance Computing
(HPC) clusters. Additionally, scientific software usually requires managing huge
amounts of numerical information (typically in terms of vectors and matrices) and
the interpretation of this kind of information requires specialized visualization
aids—a feature that traditional debuggers lack. The use of debuggers in SCS
contexts is useful for examining small computation kernels, or exercising
minute software mutation and fault injection techniques (see §4.2.2 and respectively).

Profiling tools. Profiling tools aim to analyze and report the behavior of a com-
puter running the software product under evaluation, mainly in terms of execu-
tion time (and sometimes of memory usage) for the different code sections in the
software. Profiling information is usually gathered as frequency and duration of
function calls (and eventually, corresponding allocated memory). The informa-
tion gathered and reported is typically used for optimization purposes, since any
performance bottleneck in speed (or space) is likely to stand out in a profiling summary.

Profilers differ in the degree of intrusion into the software under analysis. For example, the standard GNU profiler \texttt{gprof} uses operating system interrupts for sampling the program counter to obtain timing statistics, but relies on code instrumentation for gathering data pertaining to function calls. Code instrumentation is done at compile time, and requires passing an appropriate option to the GNU compiler \texttt{gcc} so that the generated object code is instrumented for \texttt{gprof} usage.

Even when simple ‘sampling’ profilers can only provide statistical information, in most cases that is enough to identify the ‘critical’ sections of code where the software spends the most time, and thus are the prime candidates for performance optimization. That same information can pinpoint a potential fault, such as if an iterative solver is taking more iterations than theoretically expected to converge.

\textit{Memory checking tools.} Memory checkers are software tools that specialize in detecting the analyzed software’s memory management and usage. Their main use is for detection of memory leaks. Cache memory usage—a key consideration for certain SCS products—is examined by some of these tools, which can therefore be used to complement profilers.

A well-known example of memory checker is \texttt{memcheck}, included in the Valgrind framework \cite{Valgrind}. The suite also includes Cachegrind, a cache memory processors. It can detect and count cache misses, memory references and instructions executed per line of source code. Valgrind is a \textit{dynamic binary instrumentation} framework; that is, it instruments the target software’s binary code at runtime (as opposed to compile time, like some profiling tools require) and therefore performs the analysis of machine code at run-time. Since Valgrind works directly with the software binary code, it is programming language agnostic, and can work with any compiled, just-in-time compiled, or interpreted languages—including system and third party libraries, for which usually the only form available is object code. It also copes well with dynamically generated code. However, the instrumentation cost incurred at run-time causes \texttt{memcheck} to run programs about 10 to 30 times slower than normal, and Cachegrind to run programs about 20 to 100 times slower than normal.

\textit{Unit and regression testing strategies.} The main idea behind unit testing is to test a ‘unit’ of code, such as a function. A ‘unit’ is the smallest testable part of the code. The motivation for this approach is that it is usually easier to write tests for units of code than for the entire program. Unit testing also helps to identify the problem sections when tests at the system level fail.

Effective unit testing requires that each test unit is independent from the others. Usually it is best to write a software unit and the corresponding test at the same time. By definition, unit testing only addresses the functionality of the units, so it cannot detect integration or system-level errors. However, by testing the units of a program first and then testing their ensemble, the integration testing—tests devised for the progressive aggregation of units—is significantly facilitated.

All units and their aggregates are progressively subject to integration tests until the complete set of units successfully passes an initial system integration test (meaning the software product has been achieved). The new concern at this
Certification of Scientific Computing Software

stage is to ensure that the software product continues to work under further changes, adjustments, features update, or fault detection and removal. Regression testing refers to any test suite aimed at checking whether new errors could have occurred after any of the said changes to the program.

A typical regression testing strategy consists of running previously run tests after changes to the software have been introduced, to check whether program behavior has changed, and whether previously fixed faults have surfaced again. Another common strategy is to run regression tests after every software build cycle, which is automatically performed in a regular fashion (e.g. nightly or weekly, depending on the software product’s complexity). Version control tools (such as svn and git) are indispensable tools for proper unit and regression testing under test automation frameworks.

Testing cases with known solutions—The Method of Manufactured Solutions. Many unit and integration tests will require the use of test data or problems to exercise the software or its components. For the case of SCS, such tests will generally be particular instances of the mathematical model underlying the scientific software product [10, p. 26]. Ideally, we will know the analytical solution to a test problem, so that we can assess the true error of the computed solution.

An obvious way to generate test data or to set up a test problem is to use an analytical solution to the mathematical problem, when that solution is known. If analytical solutions to the general problem are not available, sometimes it is still feasible to obtain an analytical solution to a special, a simple or a degenerate case, which will nevertheless be useful for the testing purposes, if it exercises all aspects of the software product’s code [10, p. 26].

Whenever known analytical solutions to the mathematical problem cannot be found, it might yet be possible to artificially construct problems with known solutions for most operator equations, such as integral and differential equations, or systems of algebraic equations. This procedure is aptly designated as the Method of Manufactured Solutions (MMS) [10, p. 27].

As an example of the method of manufactured solutions, assume the mathematical model for some physical problem requires solving Poisson’s equation on the domain \([0,1] \times [0,1]\) with the solution \(u(x,y)\) specified on the domain’s boundary [10, p. 27]:

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y).
\]  

(1)

Then, a test problem for the mathematical model can be constructed (‘manufactured’) by picking up any sufficiently smooth (at least twice differentiable) function \(u(x,y)\), for example:

\[
u(x,y) = x + \sin(\pi x) \cdot \cos(\pi y).
\]

(2)

Substituting (2) into (1) and differentiating appropriately, we obtain the right-hand side function \(f(x,y)\):

\[
f(x,y) = -2\pi^2 \cdot \sin(\pi x) \cdot \cos(\pi y).
\]

Then, we have a completely specified Poisson problem:

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2\pi^2 \cdot \sin(\pi x) \cdot \cos(\pi y),
\]
since the boundary conditions for the domain are obtained by direct evaluation of the function \( u(x, y) \) in (2) and, by construction, the analytical solution to this problem is the ‘manufactured’ \( u(x, y) \) given by equation (2).

The arbitrary solution of the above example might not be representative of a real problem in the application domain, since the obtained function \( f(x, y) \) is probably an unnatural forcing function for several conceivable real cases. Yet, if the purpose is to exercise all aspects of a software product that implements a solver for a two-dimensional Poisson’s equation on a rectangular domain, with Dirichlet’s boundary conditions, then—as far as software verification is concerned—we have an appropriate analytical solution as a ‘test oracle’.

Roache [37, ch. 3] presents MMS in the context of Computational Fluid Dynamics (CFD). According to Hook et al. [17] this method (developed by Roache and colleagues) constitutes the only technique available to specifically address faults in SCS code.

**Convergence testing in SCS.** Numerical methods almost always have some discretization parameters, either in space (e.g. a characteristic mesh size, \( h \)) or in time (e.g. the time step, \( t \)), or perhaps the number of iterations \( n \). *Convergent methods* are those in which the computed solution approaches the ‘exact’ or ‘true’ solution as the space or time step tends towards zero, or the number of iterations tends to infinity [10, p. 28].

Convergence tests might uncover coding errors. For some problems, theory can tell the rate of convergence of a convergent method. A systematic grid convergence test—that is, repeating the computation with decreasing mesh size \( h \)—allows one to check the actual convergence rate of the numerical method to that expected from theory. Discrepancies between these are either the result of a problem with the code, or with the method’s stability. The latter issue requires establishing regions of convergence for the software [10, pp. 29–30]. An extensive presentation of this issues is given in [37, ch. 5-8] in the context of CFD.

**Software mutation.** Software mutation is a technique that aims to test the software’s own test suite. This technique consists of introducing slight and purposeful modifications to source code, which resemble typical programming mistakes such as typing a ‘+’ sign in lieu of a ‘-’ sign in a mathematical expression, or mistaking the name of a variable, or typing an assignment in lieu of a logical comparison. Other traditional mutations consist of deleting a whole statement, or substituting boolean expressions with arbitrary truth values.

The altered software product is rebuilt and subjected to the available test suite. If the change is detected (that is, if at least one of the tests in the test suite fails after a mutation) then the introduced mutation was effectively ‘caught’ by the test suite. If the test suite fails to catch the mutation, then chances are that the test suite is either ‘weak’ or faulty. However, not all mutations change the software’s semantics or outputs and thus the test suite cannot be declared faulty until each uncaught mutation is investigated.

See [3] for an application of software mutation to assess and compare test suites in terms of their cost-effectiveness. The suitability of this technique for SCS has been addressed in [17].

**Fault Injection.** A somewhat similar idea to software mutation is that of fault injection. The purpose here is to test error handling routines. It is usually associ-
Certification of Scientific Computing Software

ated with stress testing techniques applied to robust and fault-tolerant software, or dependable software.

There are two mechanisms for fault injection. Hardware fault injection attempts to mimic random hardware failures by affecting the hardware (for example, short-circuiting a board, or introducing a faulty component). Software fault injection attempts to mimic hardware faults by software (for example, by changing values in a register or memory position).

Software fault injection can be performed at compile time or run time. The former approach requires availability of source code, and is in fact a software mutation process. A variant of this method is to add code instead of modifying the original source code, and can be regarded as a source code instrumentation technique. On the other hand, run-time fault injection has to be initiated by an event during execution, and consists in corrupting data in either registers or memory, or corrupting internal or external communications by data loss or alteration of some sort.

Software fault injection methods can take advantage of the hardware support for debugging and monitoring already existing in modern processors to produce more realistic faults with minimum interference with the target software. Those facilities can also be exploited to trace the system behavior after the fault injection. A further advantage is that by being generated at the hardware level, and not depending on any operating system service to inject the fault, even the operating system itself can be affected by the programmed fault [25]. Note that source code is not required for this particular methodology of run-time software fault injection.

Uncertainty quantification. SCS is typically characterized by an underlying mathematical model derived from a ’real-world’ problem or situation, so it is expected that there is uncertainty inherent to the model structure (Is the problem’s physics described by the equations in the model?), uncertainty in model parameters (How accurately can we estimate the model parameters?) and uncertainty in the problem setup (How accurately is the geometry of the problem known? Do we have precise initial or boundary conditions?).

Uncertainty quantification (UQ) denotes the quantitative characterization and reduction of uncertainty. We can identify the following sources:

- **Modeling uncertainty** in the continuous mathematical model derived from the physical phenomenon under study.
- **Discretization uncertainty** in the discrete mathematical model (the computational model) derived from the continuous mathematical model.
- **Software uncertainty** in the software implementation of the computational model.

According to [23], uncertainty can be classified as:

- **Aleatoric.** The probability distributions for the variabilities are known.
- **Epistemic.** The probability distributions for the variabilities are unknown.
- **Mixed.** The probability distributions for the variabilities are known but their characteristic parameters (mean, standard deviation) are unknown.
- **Model dependent.** The probability distributions for the variabilities are particular to each model.
Approaches to UQ depend on the uncertainty type. Aleatoric uncertainty quantification is relatively straightforward to perform, by application of statistical techniques such as Monte Carlo directly to simulations. Epistemic and Mixed uncertainty quantification has been tackled with methods such as fuzzy logic or evidence theory. Model form uncertainty is the most challenging sort of uncertainty due to the extreme lack of knowledge. Uncertainties have been approached by attempting to reduce them to aleatoric type, but sometimes all types of uncertainty are present in the problem to be solved.

UQ is not limited to studying the propagation of uncertainty from the inputs to the output of interest; it can encompass uncertainty analysis, sensitivity analysis, forward and backward error analysis, a posteriori error estimation, design optimization and design validation and calibration. Because of this, some authors consider UQ as a natural extension of SCS V&V processes. Note that UQ techniques do not require the software source code.

Interval Arithmetic. When it comes to quantifying uncertainties in physical measurements, it is natural to express measurements as intervals instead of a single real number. By performing appropriate uncertainty propagation, scientists can compute the uncertainty in a derived magnitude of interest.

Surprisingly enough, computing with intervals is not commonplace in SCS, despite the fact that interval arithmetic—that is, rules for calculating with intervals and other sets of real numbers—has been proposed as early as 1931, and began to gain momentum shortly after the 1950s [14, p. 486].

The idea behind interval computations on a digital computer is to replace floating-point numbers with intervals and develop algorithms that produce tight, guaranteed bounds on the computed results. Usually, an interval algorithm is substantially different from its floating-point counterpart.

Unlike floating-point arithmetic, interval arithmetic allows performing rigorous error analysis, by computing mathematical bounds on the set of all possible solutions [40, p. 5]. Similarly to the uncertainty propagation that scientists use in their calculations, using interval arithmetic one can propagate the uncertainty of an input quantity through a computation, and therefore provide the corresponding uncertainty in the result [10, p. 30].

Current mainstream processors do not provide interval-specific hardware instructions for the basic arithmetic operations (addition, subtraction, multiplication and division). A quite recent proposal for hardware support for Interval Arithmetic is given in [20]. However, almost all processors implement the IEEE 754 Floating-Point standard, which provides directed rounding modes that can be exploited to implement interval computations. An extensive list of pointers to interval computation resources is kept at [22].

The main weakness of interval arithmetic is that the bounds for cascaded computations tend to grow wider than the true range. To decrease this excess interval width, approximations by linear functions (affine arithmetic [9]) or by general polynomials (Taylor series methods, [30]) are used to keep track of the data dependency, and this information allows one to tighten the bounds on the intermediate results.

4.3 Validation Techniques

For SCS, validation is essentially checking whether the software results agree with some real-world problem, typically, experimental results. Given that real-
Certification of Scientific Computing Software

world experiments are themselves a possible source of uncertainty, validating the experiments design and execution is a key activity for SCS validation.

Note that if, for any reason, there are no experimental results available for comparison (e.g. the physical experiment is infeasible), achieving full validation of SCS, as per our definition, can range from difficult to impossible. However, partial validation can be achieved by conducting solution verification to demonstrate that the computational model is a sufficiently accurate representation of the mathematical one [10, p. 17].

Among general methods and tools for validation in scientific computing, we can mention the following.

Data visualization tools. These kind of tools facilitate the peer-review of scientific software results in a suitable graphical format for the application domain. Given the huge amount of data involved in some scientific computing experiments, such tools are probably non-trivial software designs; it is not uncommon that they need to run in high-performance computer clusters, where the scientific computing originally takes place, so there is an additional computational complexity to these software products. These tools are beyond the software of interest itself (and thus, not part of the V&V scheme) but clearly demand verification schemes themselves, if validation of the scientific computing software will partially or completely depend on these tools.

Data comparison tools. These tools are of the sort of the typical diff tools in Unix environments, but tailored to compare experimental data to computed results. As such, they are geared towards comparing floating-point numbers, either by computing relative or absolute differences between quantities, or specific norms for vector and matrix quantities.

Conservation of physical magnitudes. Conservation of physical magnitudes like mass, energy, or entropy, can be used as an aid in the scientific software validation process. Agreement with the conservation laws does not guarantee validity, but disagreement clearly shows that the solution is invalid.

Relative benchmarking. In the software validation context, relative benchmarking consists in comparing the results of the software under test against other ‘trusted’ (or validated) software systems. Preferably, the comparison should be against software products that use different mathematical and computational models, so that there is independent support for the conclusions. To know whether the software models and implementations are different requires knowledge of the internals of both software products. The idea of a relative comparison between independent software solutions is also an option for verification of non-functional software requirements.

5 Product-based approach to Software Certification

The rationale for a product-based approach to software certification advocated in this document is succinctly conveyed in the web page of McMaster’s Centre for Software Certification (McSCert, [27]):

“Critical, software-intensive devices are typically certified on the basis of the process used to develop them. We believe that this is inadequate, that while a good process may be necessary for producing dependable software, it is not sufficient: certification must also be based on evidence
obtained from the product. Our research is therefore into what kind of evidence is sufficient, and how different kinds of evidence may be combined into an argument for safety that is sufficient.”

The product-based approach advocated by McSCert reinforces the fact that at the end of the day it is the product that matters. It may be easier to certify the process used to create the product, but it is the quality of the product itself that determines success or failure of the project. The emphasis on the product encompasses an often overlooked fact, that the safety of products that depend on software is a problem in systems engineering. Even the hardware that contains the software has to be part of the engineering—and thus, part of the certification—and so should any tools used for the development and execution of the software in the product (any preprocessor, compiler, virtual machine, assembler, static linker, dynamic linker, loader, third party and/or system libraries, and executive kernel or full-fledged operating system involved in development, deployment and execution).

The evidence sought in the product-based approach to software certification can be both analytical evidence and experimental evidence. The analytical evidence is typically related to the specification aspects, and is collected at the design stages. The experimental evidence is typically related to the actual implementation, and is collected at the verification—and, eventually, validation—stages. Product based software certification is not easy. Several hurdles need to be overcome, as presented in [13].

6 Conclusions

In light of the underestimated yet widespread reliance on SCS for decision making, and the role SCS plays in a number of safety- and mission-critical applications, we believe that certification of this class of software is an endeavor of paramount importance. In particular, the potential consequences of uncertified SCS in environments such as nuclear, aerospace and transportation, justifies the extra cost and effort in pursuing a product-based rather than a process-based approach to certification, despite the current challenges the product-based approach has to overcome.

We favor a constructive approach to software development, that intends to design out potential faults. In particular, for SCS we advocate a document-driven development process that judiciously applies literate programming techniques to critical source code sections, and uses formal methods when feasible and cost-effective (possibly, for the specification phases of development). This approach facilitates software inspections, and particularly, code revisions.

The method of manufactured solutions should be part of any serious verification process for SCS. Uncertainty quantification is a desirable goal in SCS. Lastly, fault injection and software mutation techniques can be applied with a high probability that they uncover latent faults. In particular, using different rounding modes can potentially flag unstable algorithms at the core of SCS.

Software engineering practices and general software engineering tools (such as version control tools, and automated code documentation tools and code generation) can greatly improve the qualities of any piece of software. There is certainly room for improvement in the area of integrating software engineering methodologies into SCS development.
References


41. Valgrind web page. [http://valgrind.org/]
