When Scientific Software Meets Software Engineering

Article · January 2021

CITATIONS
0

READS
7

4 authors, including:

Dorian Leroy
Université de Rennes 1
7 PUBLICATIONS **36** CITATIONS
SEE PROFILE

Johann Bourcier
Université de Rennes 1
49 PUBLICATIONS **423** CITATIONS
SEE PROFILE

When Scientific Software Meets Software Engineering

Dorian Leroy, June Sallou, Johann Bourcier, Benoit Combemale

Department: Head
Editor: Name, xxxx@email

When Scientific Software Meets Software Engineering

Dorian Leroy
Université de Rennes 1, Inria, France

June Sallou
Université de Rennes 1, Inria, CNRS, IRISA & Géosciences Rennes, OSUR, France

Johann Bourcier
Université de Rennes 1, Inria, CNRS, IRISA, France

Benoit Combemale
Université de Rennes 1, Inria, CNRS, IRISA, France

***Abstract*—The development of scientific software relies on the collaboration of various stakeholders for the scientific computing and software engineering activities. Computer languages have an impact on both activities and related concerns, as well as on the engineering principles required to ensure the development of reliable scientific software. The more general-purpose the language is —with low-level, computing-related, system abstractions— the more flexibility it will provide, but also the more rigorous engineering principles and Validation & Verification (V&V) activities it will require from the language user. In this paper, we investigate the different levels of abstraction, linked to the diverse artifacts of the scientific software development process, a software language can propose, and the V&V facilities associated to the corresponding level of abstraction the language can provide to the user. We aim to raise awareness among scientists, engineers and language providers on their shared responsibility in developing reliable scientific software.**

SCIENTIFIC SOFTWARE INTO THE WILD

Scientific computing, also known as computational sciences, consists in the use of advanced computing capabilities to understand and solve complex scientific problems (e.g., biological, physical, and social), but also engineering and humanities problems. It aims to predict the behavior or the outcome of a physical system, being natural or man-made. While initially restricted to the activity of certain experts such as scientists or engineers, scientific computing has been made popular and widely exposed to citizens due to major and damaging societal problems such as climate change and more recently pandemics (e.g., COVID-19). Most of the political

Published by the IEEE Computer Society

decisions regarding these wicked problems are backed up by scientific findings, all based on scientific computing.

Scientific computing is a cross-cutting field, but its heart and soul lie in the development of mathematical models to understand physical systems through their simulations. Those models can be numerical (e.g., systems of differential equations), non-numerical (e.g., agent-based models) or based on analytics (e.g., machine learning models), and capture the behavior of the modeled system. Numerical models can be further refined as continuous or discrete. Simulations of mathematical models correspond to the execution of the computer programs containing these models, the so-called "simulation codes". In this paper, we refer to the subsuming concept of *scientific software*, which we define as software dedicated to scientific computing and simulation. The development of these scientific software includes both software engineering and scientific computing concerns. Mathematical models and scientific software are therefore tightly intertwined throughout their life cycles. The tools and methods used for their development (e.g., computer languages) have an impact on the definition of both, as well as on the engineering principles required to ensure the development of reliable scientific software.

Due to the ever-increasing intrinsic complexity of such mathematical models and the pursuit of ever more efficient simulations, the scientific computing and software engineering communities have worked together for decades to explore various approaches to assess the reliability of the proposed models and the correctness of their implementations in scientific software [1]. However, when the time comes to implement a new model – and thus new simulation software – scientists and engineers are faced with decisive choices such as what computer language(s) to use (e.g., MATLAB, Mathematica, Fortran, Python, C++, or even an in-house domain-specific language). This choice has important consequences on the expressiveness at hand to implement the model and the corresponding simulation code, but also in terms of software engineering practices to apply to develop reliable and efficient scientific software. As usual, power comes with responsibility! The more general-purpose the language is – with low-level, computing-related, system abstractions – the more flexibility and performance it may provide, but also the more rigorous engineering principles and Validation & Verification (V&V) activities it will require from the language user to develop a reliable piece of scientific software.

However, most scientists and engineers do not have a background in software engineering and are thus not aware of its best practices beyond programming (*e.g.,* version management, component reuse, continuous integration, unit testing) [2], which has actually prompted initiatives aiming to tackle the issue, such as the Research Software Engineering [3]. Since the final goal is to build and apply the model encoded in the simulation code, the code itself is merely a means to that end. Final stakeholders (e.g., citizens, policy and decision makers, research institutions, system users...) may even be ignorant of the importance of code for science and engineering. This situation has recently been brought to light by Konrad Hinsen in a post reporting criticisms about an epidemics simulation code (http://blog.khinsen.net/posts/2020/05/18/an-open-letter-to-software-engineers-criticizing-neil-ferguson-s-epidemics-simulation-code/), and the subsequent discussions on the blog as well as on social media (https://twitter.com/khinsen/status/1262307434632282112).

In this paper, we explore the overall scientific software development process, we provide an integrated view of the scientific computing and software engineering activities, artifacts and roles, and we discuss the trade-offs on the required V&V activities with regard to the computer languages at hand. This work offers an holistic view through which we introduce the existing V&V approaches from the literature to support such activities and the accountability of the respective roles. We aim to raise awareness among scientists, engineers and language providers on their shared responsibility in developing reliable scientific software.

ON THE ENGINEERING OF SCIENTIFIC SOFTWARE

Scientific software can rely on various types of scientific models that represent the behavior of the physical system under study. A com-

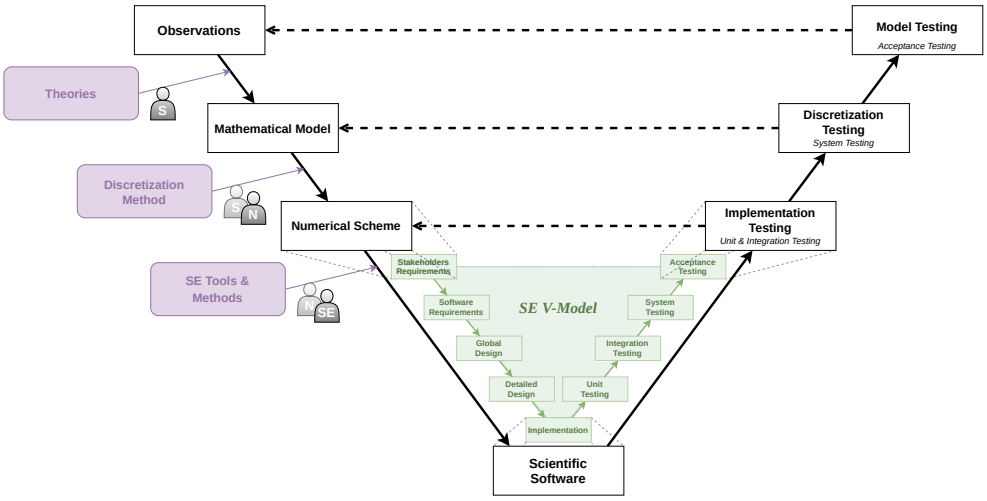


Figure 1: Overall scientific software development process across the V-model.

mon classification separates numerical models (e.g., systems of differential equations) from non-numerical models (e.g., agent-based models). We focus our discussion on scientific software based solely on continuous numerical models, which we will address as *mathematical models* in the rest of this paper.

Software engineers generally follow an engineering process, aka. Software Development Life Cycle (SDLC), that introduces the required activities to produce software. For instance, the V-model (cf. green part in Figure 1) has been introduced to highlight the relationships between each activity of the development life cycle and its associated activity of V&V. In the rest of the paper, we use the V-model as a support for the identification of the main software engineering activities, regardless of the associated method (e.g., incremental, iterative or agile). The same activities can be retrieved in all methods, possibly with different time dimensions (e.g., a V-model in each sprint of an agile method).

In Figure 1, we propose an engineering process adapted to scientific software development in the form of a V-model, aka. *scientific V-model*, subsuming the original *software engineering (SE) V-model*. We introduce the roles, activities and artifacts of this V-model, with an emphasis on V&V concerns. The V-model is composed of two branches: the first one on the left is top-down, corresponding to the successive elaboration of the various artifacts, and the second on the right is bottom-up, representing the validation process of the scientific software. At each level, a V&V activity on the upward branch is located across from an artifact on the downward branch. The purpose of the V&V activity is to validate the scientific software with respect to the artifact in question. This is represented by the dashed arrows in Figure 1.

The development of scientific software involves several activities that manipulate different types of artifacts. Thus, the design of scientific software based on mathematical models requires the involvement and cooperation of various stakeholders ranging from scientists and engineers to experts in software engineering or numerical analysis. These stakeholders play one of three roles (according to the development context, one person might endorse more than one role): *scientists* as domain experts (*S* in Figure 1), *numerical analysts* as experts in the discretization of a continuous phenomenon on a computer (*N* in Figure 1), and *software engineers* as experts of software development to deliver the expected services (*SE* in Figure 1). It is worth noting the very different background and expertise of these three roles, and the tight collaboration required across the development process.

The tasks linked to each role are associated with the activities and artifacts forming the V-model represented in Figure 1. From a set of

observations of a phenomenon and *theories* (illustrated in the upper left part of the figure), *scientists* develop a *mathematical model*, *a.k.a.* continuous numerical model, often taking the form of systems of differential equations. Since these systems of equations cannot generally be solved analytically, they must be solved numerically, which requires a discretization process. This consists in applying a *discretization method* (*e.g.,* finite differences, finite volumes) to obtain a *numerical scheme*. This numerical scheme is a discretized mathematical model, which specifies the sequence of computations to execute, given a discretization of the domain according to discretization parameters (*e.g.,* space, time), an initial state (*e.g.,* initial temperature in every point of the domain) and inputs from the environment (*e.g.,* rainfall), to compute a numerical (as opposed to analytical) solution to the model. The implementation of the discretization process requires specific skills and it is undertaken by *numerical analysts* (*N* in dark grey in Figure 1), in collaboration with *scientists* who act as domain experts (*S* in light grey in Figure 1).

From that point on, software engineering concerns enter the development process, as well as the *software engineers*. While *numerical analysts* are required as domain expert (*N* in light grey in Figure 1), such software engineering concerns require specific skills brought by *software engineers* (*SE* in dark grey in Figure 1). This is represented on the middle lower part of Figure 1 with a second, nested V-model (in green) corresponding to the usual V-model used in software engineering. This V-model effectively bridges the gap between the *numerical scheme* – which can be considered as the *stakeholder requirements* for *scientific software* – and the software engineering process. It encompasses all the software engineering concerns to obtain a reliable working software, including performance, concurrency, targeted architectures, memory management, data access, and so on.

On the left top-down branch, the stakeholder requirements are refined and extended with more specific software concerns corresponding to the different levels of organization of the software. The elaboration of the different specifications follows a standard process. First, the *software requirements* are formulated from the global stakeholder requirements and they specify the expectations for the software as a whole. They contain information about the general software organization and the expected behaviors of the software in various scenarios of interest (*e.g.,* use case diagrams). Then, the *general design* is drawn to more specifically describe the different components that shape the software (*e.g.,* component diagrams). This leads to the formulation of the *detailed design* of the software which aims to document all the structural details of the software (*e.g.,* class and sequence diagrams) in order to then allow the implementation of the software in a clear and efficient way. Using the various requirements contained in the different previous specifications, the *implementation* of the scientific software is generated.

The resulting scientific software is then validated according to the stakeholder requirements, through V&V activities under the form of *unit*, *integration*, *system*, and *acceptance testing*, all encompassed by *implementation testing* from the point of view of the *numerical analyst*. First, *unit testing* ensures that the implemented scientific software respects the guidelines of the detailed design, meaning that each unit of the software code performs as expected (*e.g.,* mock-based testing). The fulfillment of the general design specifications are checked through *integration testing* as it makes sure that the different components interact correctly (*e.g.,* Big Bang testing, incremental testing). Then, *system testing* controls that the implementation meets the software requirements by validating the complete and fully integrated software – considered here as the system of interest – thanks to various testing techniques (*e.g.,* usability testing, load testing, regression testing, functional testing). Finally, compliance with the stakeholders' expectations is validated through *acceptance testing* (*e.g.,* user acceptance testing, business acceptance testing, operational testing).

Once the software engineering V&V concerns have been addressed, the conformity of the scientific software to its original mathematical model is ensured throughout a step called *discretization testing*. During this testing activity, *numerical analysts* aim to validate whether the scientific software produces results that are within an acceptable margin with regards to the results

that would be obtained by analytically solving the mathematical model. The techniques used for discretization testing vary in their level of rigor, depending mostly on the availability of an exact solution to the mathematical model. Without such a solution, techniques such as symmetry, conservation, or Galilean invariance testing can be employed [4].

These techniques leverage domain knowledge that the numerical solution should exhibit specific characteristics such as symmetry when provided with symmetric geometry and conditions, conservation of some physical properties (*e.g.,* mass, energy), or be unaffected by operational changes such as inverting the axes of the coordinate system. Note that, as these tests rely on domain knowledge, some might not be used from one field to another (*e.g.,* when no symmetry is present). Another technique that can be used without an exact solution is code-to-code comparison [5], which consists in running a simulation of the same mathematical model embedded in another piece of scientific software, and comparing the results.

Yet, obtaining the highest degree of confidence in the discretized model requires techniques leveraging an exact solution to the mathematical model. When an analytical solution is not available, exact solutions can be obtained through the use of the method of manufactured solutions [6], which constrains the simulator to compute a solution chosen beforehand (the so-called manufactured solution). Whether analytical or manufactured, an exact solution allows to employ rigorous techniques such as discretization error quantification, iterative convergence testing, and order-of-accuracy testing, respectively in order of increasing rigor [7]. In essence, these techniques aim to assess whether the discretization error, *i.e.,* the difference between the numerical and the exact solutions, tends toward zero as the value of the discretization parameters decreases, and thus that the numerical scheme is a correct discretization of the mathematical model.

Finally, the last V&V step we designate as *model testing* is conducted by *scientists*. Model testing (a.k.a. model validation) aims to statistically quantify the disagreement between the numerical solutions and the available experimental data, and to assess whether the resulting uncertainty is acceptable within the application domain of the model [8]. Ideally, these data are collected through a set of validation experiments. Validation experiments are a special kind of experiment aiming to measure the conditions of the experiment as precisely and completely as possible, with less emphasis on controlling the environment than for other kinds of experiments [9]. This distinction allows to run simulations of the model whose inputs match very closely those of the experiments, and thus to assess the fidelity, in these precise conditions, of the outputs of the simulations with regard to collected measurements.

A general approach to model testing is proposed in [8]. Broadly, this approach consists in (i) identifying and characterizing mathematically every source of uncertainty (aleatory and epistemic) [10] in both the scientific software and experimental measurements used as a reference, (ii) propagating those uncertainties to the outputs through techniques such as sensitivity analysis [11], (iii) defining a validation metric, *i.e.,* a mathematical operator comparing the numerical solution and the experimental measurements, and applying it to produce an assessment of the validity of the model over the validation domain, and finally (iv) employing statistical techniques such as regression fits of the validity metric over the validation domain, which enables to interpolate or extrapolate the validity metric over the domain of intended use, thereby providing the uncertainty of the simulation results over the domain of application.

Overall, the different testing activities of the scientific V-model can be related to the corresponding steps of the software engineering V-model: the *implementation testing* step can be conceptually compared to the *unit and integration testing* step, the *discretization testing* to the *system testing*, and the *model testing* to the final scientific software *acceptance testing*.

These activities serve the same purpose but use different techniques, according to the nature of the artifact under test. The observations fulfill the role of the stakeholders requirements in that they directly represent the essence of the phenomenon to be modeled. The mathematical model can be considered as the software requirements as it encompasses the global scientific behavior of

the physical system the software should encode. The numerical scheme reflects the purpose of the global and detailed designs as it specifies how the numerical solution to the mathematical model should be computed in the software, and how each step of computation is related to one another, in a similar way as the behavior and interaction of units and components are described in the design specifications.

However, the techniques used are distinctive as they depend on very different natures of artifacts – present on the left top-down branch of the V-model depicted in Figure 1 – on which they validate the software. For instance, discretization testing, which we relate to system testing in the context of scientific software development, aims at validating the scientific software with regards to the mathematical model. Since the mathematical model consists of mathematical equations, associated V&V techniques manipulate mathematical concepts such as the convergence of numerical sequences in the case of iterative convergence testing. Alternatively, the system testing phase of the nested V-model uses the software requirements as a reference to assess the validity of the software. As those are software specifications, the testing techniques manipulate software states and behaviors. They ensure that the different states are consistent from a software perspective. For example, that multiple simultaneous runs of the software do not affect the output of each individual run.

POWER COMES WITH RESPONSIBILITY!

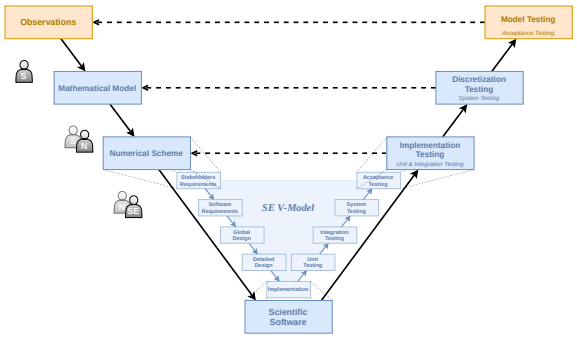
The overall development process is complex and challenging when it comes to ensure reliable scientific software. To mitigate this complexity and allow scientists and numerical analysts to directly implement their scientific software, existing computer languages come with dedicated constructs that balance the responsibility between the *language user* and the *language provider*. We explore various computer languages at hand to implement scientific software according to their provided constructs, and discuss the consequences on the expressive power and on the required V&V activities. We illustrate this in Figure 2, where we rely on the V-cycle for the development of scientific software proposed in Figure 1. In Figure 2, the orange and blue colors are used to identify who supplies what artifacts and who is responsible for what V&V activities. The orange color refers to language users, while the blue color refers to language providers.

REUSING EXISTING SIMULATORS

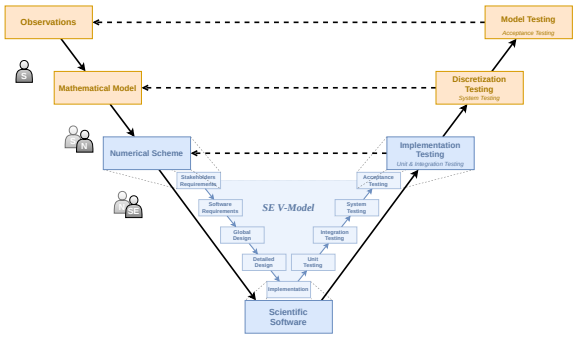
Figure 2a shows the situation where scientists directly use existing scientific software such as MODFLOW [12] or MITgcm [13]. In this situation, language users are scientists configuring the models encoded into an existing piece of scientific software for their specific case. The language used in this case corresponds to a configuration language used to configure simulators and run simulations according to specific parameters (*e.g.,* FloPy for MODFLOW [14]). Language users specify fixed input quantities such as the geometry, or physical modeling parameters of the system and obtain a simulator specific to a certain context (*e.g.,* groundwater flows in a specific watershed area). Then, for each simulation, language users specify the input quantities that vary from one simulation to another, such as the initial and boundary conditions, or the system excitation over time.

Oftentimes, some of those input quantities are unknown (*e.g.,* when they cannot be measured in the field), or known only with a high uncertainty. In such cases, language users will need to conduct *model calibration* to reduce the parametric uncertainty of their model and obtain useful results. Over the years, a number of techniques have been developed to conduct model calibration [15]. Yet, model calibration cannot eliminate every source of uncertainty. Thus, some means of propagating uncertainties over the inputs through the model to obtain the corresponding output uncertainties are required (*e.g.,* sensitivity analysis [11]). While numerous black-box methods exist, white-box (a.k.a. open-box) methods that leverage domain knowledge receive more attention [16]. However, such methods require access to internal variables of the embedded model. The latter must thus be exposed as configurable simulation parameters to enable the use of certain state-of-the-art techniques.

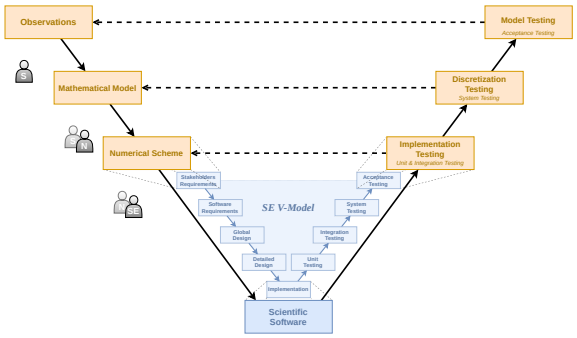
In the situation depicted in Figure 2a, the simulator offers abstractions specific to the embedded mathematical model (*e.g.,* soil perme-



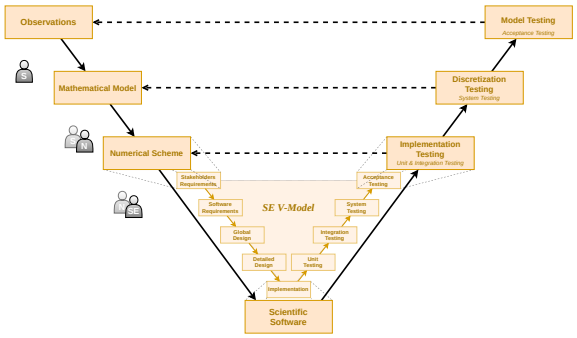
(a) Scientific software development using an existing simulator. (eg., Modflow [12])



(b) Scientific software development using languages for mathematical models. (eg., Matlab, Mathematica)



(c) Scientific software development using languages for numerical schemes. (eg., Blitz++, SciPy, NabLab)



(d) Scientific software development using languages with system abstractions. (eg., C++, Python)

Figure 2: Scientific software development: responsibilities among the language user (orange) and the language provider (blue), depending on the abstraction level of the provided language constructs.

ability, porosity), which users can parameterize to run context-specific simulations through the underlying scientific software. Thus, developers of a simulator must first endorse all the responsibilities associated with the development of a complete piece of scientific software, as covered in the previous section, including *implementation*, *discretization*, and *model testing*.

An important language-related responsibility that befalls simulator developers is to clearly identify, document, and enforce as much as possible the validity envelope of the abstractions exposed by the configuration language of the simulator, to prevent its misuse. In the general sense, we define the validity envelope of a language as the set of valid programs one can write with that language. In the case of a configuration language dedicated to a specific piece of scientific software (such as FloPy), this validity envelope must prevent unsound module and parameter combinations, and enforce the value of parameters to be within their bounds (*e.g.,* non-negative volume of rainfall).

Due to combinatorial explosion, an abstraction of the domain is generally required to identify and enforce the validity envelope of configuration languages, as far as unsound combinations of parameters are concerned particularly. This will, in turn, necessarily result in false positives or false negatives. Techniques such as sensitivity analysis can be used to identify the relationships between the different parameters, and thus detect such unsound combinations.

LANGUAGES FOR MATHEMATICAL MODELS

When existing scientific software do not embed the desired mathematical model, one will need to define their own. Defining a *mathematical model* and deriving the corresponding *scientific software* can be done directly, using languages such as Mathematica (https://www.wolfram.com/mathematica) or MATLAB (https:

//matlab.mathworks.com). Figure 2b depicts the development of such languages and the development of scientific software using them.

In this scenario, language users are scientists implementing their mathematical model using a language that offers constructs at the corresponding abstraction level (*i.e.,* continuous mathematics), and allows them to derive the corresponding piece of scientific software. Thus, they endorse the responsibility of addressing the V&V concerns associated with model testing – identified in the previous section – since those are out of scope of such a language. In addition, in order to validate the implementation of their mathematical model with regard to its specification, language users perform discretization testing.

However, as explained in the previous section, when no analytical solution to the mathematical model is available, the most rigorous techniques for discretization testing (*e.g.,* iterative convergence testing, order-of-accuracy testing) can only be used by employing the *method of manufactured solutions* [6], which requires to inject source terms into the discretized model. Therefore, to enable the use of these techniques, the language needs to either provide tools allowing it, or to directly give access to the discretized model. Otherwise, only less rigorous techniques such as those described in the previous section can be employed to conduct discretization testing.

Language providers are in charge of designing continuous mathematical constructs (*e.g.,* differential blocks in MATLAB's block diagrams) allowing language users to define their mathematical models, and also developing the infrastructure (*e.g.,* compiler) to derive reliable working scientific software from any mathematical model. This language infrastructure performs automated and possibly configurable discretization and scientific software derivation (as indicated by the blue *numerical scheme* and *scientific software* boxes on Figure 2b). So, language providers must handle the corresponding V&V concerns (*i.e.,* the concerns pertaining to the discretization of their language constructs), as well as the software engineering concerns – encapsulated into the *SE V-model* – which are tied to the implementation of their discretized language constructs.

For that reason, in addition to *software engineers*, *numerical analysts* take an active part in language design, and *scientists* come in assistance to provide the language requirements and criteria for the verification and validation of the language constructs and of their discretization. While the techniques used to address the V&V concerns related to the discretization of language constructs and of plain mathematical models are the same, they are here applied to individual language constructs and combinations thereof, instead of complete mathematical models. The objective is to cover as many valid uses of the language as possible to check that its validity envelope subsumes the application domain intended for the language.

Finally, to allow language users to validate the implementation of their mathematical model, tools for discretization testing should be provided, in particular tools allowing an automated use of the method of manufactured solutions. This allows language users to perform the required validation at the level of abstraction of the language (the continuous mathematical level in this case), which is an essential functionality of software languages.

LANGUAGES FOR NUMERICAL SCHEMES

When more control is required over what discretization methods to apply and how to apply them, and when languages at the mathematical level of abstraction do not fit this need, one has to handle the discretization step oneself. Figure 2c depicts a situation in which numerical analysts use languages dedicated to the definition of *numerical schemes* (or with the right abstractions to do so) such as NabLab [17], Julia [18], SciPy [19], GNU Octave (https://www.gnu.org/software/octave/), or Blitz++ (https://github.com/blitzpp/blitz). These languages allow to automatically derive the corresponding piece of scientific software, on which the scientific V&V activities listed in the previous section can be conducted, without having to handle software engineering concerns.

In this scenario, language users are numerical analysts who apply discretization methods on mathematical models provided by scientists to obtain numerical schemes, that they implement using a language thaht offers constructs at the corresponding level of abstraction, and that allows them to derive a piece of scientific software

for the discretized model. Therefore, they endorse the responsibility of addressing the V&V concerns corresponding to discretization testing while scientists endorse the one associated with model testing. Indeed, those responsibilities are out of the scope of a language at the numerical scheme level of abstraction, as indicated by the orange color of the corresponding boxes in Figure 2c. Finally, language users must validate their implementation of their numerical scheme with regard to its specification, which is the aim of implementation testing.

In this situation, implementation testing is performed with the aid of tool support from the language that allows language users to stay at the same abstraction level (*i.e.,* discrete mathematics), without having to consider system concerns such as memory management or concurrency.

Language providers expose discrete mathematical constructs that allow language users to implement their numerical scheme. From the numerical schemes defined with those constructs, the infrastructure of the language (*e.g.,* model transformations, interpreters, compilers, code generators) derives the corresponding piece of scientific software. Therefore, language providers must tackle the system concerns (*e.g.,* memory management, concurrency, data management) and corresponding V&V concerns of the language infrastructure to make it possible to tackle the concerns of the scientific software thus generated. In essence, this means that the concerns must be addressed for any valid numerical scheme written by the language user, which is depicted in Figure 2c by the blue color of the *SE V-model* and *scientific software*.

While this task is mainly undertaken by software engineers, the assistance of numerical analysts is required to both define the required expressivity for the language (which in turn allows to derive the stakeholder requirements from a given numerical scheme), and perform the necessary V&V activities to ensure that the effective semantics of the derived piece of software is equivalent to the semantics of the provided numerical schemes. In other words, software engineers require the assistance of numerical analysts to tackle concerns related to the abstractions at the discrete mathematics level, and clearly define the validity envelope of the language.

Then, to foster the robustness of the V&V process spanning the entire V-model, language providers should offer tool support allowing language users to perform implementation testing at the discrete mathematics level of abstraction. This tooling can take the form of a dedicated debugger, and optimally includes facilities dedicated to the analysis of numerical schemes such as plot drawing, monitoring variable values and watch expressions (*i.e.,* exogenous expressions of no interest for actual simulations, evaluated for debugging purposes only), and conditional breakpoints and step-by-step execution to detect and investigate faults in the numerical scheme.

LANGUAGES FOR SCIENTIFIC SOFTWARE

Finally, when performance is critical to a particular piece of scientific software, be it in terms of memory or execution time, specific hardware and architectures need to be targeted (*e.g.,* heterogeneous and performance-oriented infrastructures such as distributed CPU-GPU architectures). If no language for either mathematical models or numerical schemes supports the targeted type of architecture, system-level languages such as C, C++, and Fortran can be used, conjointly with APIs such as OpenMP, and standards such as MPI and SYCL. Continuous progress in the hardware domain and in the field of high performance computing (HPC) pushes the developers of performance-critical scientific software to use such system-level abstractions, as they are often the only way to leverage the latest HPC-specific technologies.

Figure 2d depicts developments where language users express the particularities of their simulator with regard to all the concerns involved in the development of scientific software, ranging from the mathematical model, to the encoded numerical scheme, to system-level concerns such as concurrency, memory, and data handling. In this case, language users must also endorse the responsibilities of addressing the V&V concerns corresponding to each step in the development process, from the V&V of the aforementioned system concerns, to discretization testing, to model testing. In such a situation, another possibility for language users is to take the role of language provider for an existing language for mathematical models or numerical schemes.

This places them as language providers in the situations depicted in Figure 2c or Figure 2b. As such, they can broaden the range of architectures that can be targeted by the existing language by including their architecture of choice, and thus foster the use of the language in the community as a language at a higher abstraction level than system-level languages that nonetheless allows to target state-of-the-art hardware.

KEY TAKEAWAYS AND LOOKING AHEAD

When developing scientific software, dedicated languages for mathematical models and numerical schemes (*e.g.,* MATLAB, NabLab, SciPy) bring benefits to users from the scientific community with regard to languages at a low abstraction level (*e.g.,* C++, Fortran), as low-level V&V and performance concerns are addressed as part of the development of those high-level, dedicated languages. This means that users of such high-level languages, *i.e.,* scientists and numerical analysts, can leave aside software engineering concerns to focus on their area of expertise and the associated V&V concerns, resulting in an increased robustness of the end-to-end development process of scientific software.

In exchange for these benefits to language users, designers of high-level, dedicated languages face additional challenges: the higher the abstraction level offered by a language is, the more steps and complexity in the V&V process of the language itself there are. These languages have to guarantee the correctness and performance of the scientific software resulting from any valid piece of code, mathematical model, or numerical scheme written with them, and provide tools to language users to validate their use of the language. For instance, developing a dedicated language at the mathematical level requires the designers to ensure not only that their implementation and discretization testing cover any mathematical model written with the language, but also that users are able to perform discretization testing specific to their precise piece of scientific software.

We highlight in this paper the importance of extensively documenting the V&V processes that are conducted as part of the development of a language. This allows language users to clearly identify what V&V concerns are or not already addressed as part of a language. Consequently, users can pick languages according to both their goals in terms of scientific software development, and to the V&V concerns they are able to continuously address themselves according to the state of the current knowledge. This way, scientists are less likely to have to handle software engineering concerns when they do not want to double as software engineers. In addition, this allows to capitalize on the aforementioned development and V&V overhead that comes with the development of those languages.

Another key point is the need for language providers to offer, as part of the language infrastructure, facilities to allow language users to conduct V&V activities tailored to their specific simulation model. This enables the continuity of the V&V process, from user-defined program to scientific software, and fosters both transparency of the complete development process, and trust in the resulting scientific software.

The work presented in this paper leads to a clarification of the roles of the various stakeholders taking part in the development of scientific software. A reasoned approach for the development of reliable scientific software has been proposed that allows to clearly characterize the validity envelope of this type of software. We believe that the generalized use of our approach will allow to systematically characterize the validity envelope of scientific software, to make it explicit and thus lead to a better and safer use of these software.

ACKNOWLEDGMENT

We would like to thank our colleagues at the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), at the Geoscience department of the Observatoire des Sciences de l'Univers de Rennes (OSUR), as well as at the Commissariat à l'énergie atomique et aux énergies alternatives (CEA), for their input and providing such interesting conversations that contributed to the writing and improvement of this paper. We also want to thank Hinsen Konrad and Gordon Blair for their valuable comments, which helped to improve the paper.

REFERENCES

1. Kanewala, U. and Bieman, J.M., 2014. Testing scientific software: A systematic literature review. Information and software technology, 56(10), pp.1219-1232.
2. Wilson, G., Aruliah, D.A., Brown, C.T., Hong, N.P.C., Davis, M., Guy, R.T., Haddock, S.H., Huff, K.D., Mitchell, I.M., Plumbley, M.D. and Waugh, B., 2014. Best practices for scientific computing. PLoS biology, 12(1).
3. Cohen, J., Katz, D.S., Barker, M., Hong, N.C., Haines, R. and Jay, C., 2020. The four pillars of research software engineering. IEEE Software, 38(1), pp.97-105.
4. Robertson, B.E., Kravtsov, A.V., Gnedin, N.Y., Abel, T. and Rudd, D.H., 2010. Computational Eulerian hydrodynamics and Galilean invariance. Monthly Notices of the Royal Astronomical Society, 401(4), pp.2463-2476.
5. Oberkampf, W.L., Trucano, T.G. and Pilch, M.M., 2003. On the role of code comparisons in verification and validation (No. SAND2003-2752). Sandia National Labs.
6. Roache, P.J., 2002. Code verification by the method of manufactured solutions. J. Fluids Eng., 124(1), pp.4-10.
7. Oberkampf, W.L. and Roy, C.J., 2010. Verification and validation in scientific computing. Cambridge Univ. Press.
8. Roy, C.J. and Oberkampf, W.L., 2011. A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing. Computer methods in applied mechanics and engineering, 200(25-28), pp.2131-2144.
9. Oberkampf, W.L. and Smith, B.L., 2017. Assessment criteria for computational fluid dynamics model validation experiments. Journal of Verification, Validation and Uncertainty Quantification, 2(3).
10. Helton, J.C., Johnson, J.D., Oberkampf, W.L. and Sallaberry, C.J., 2010. Representation of analysis results involving aleatory and epistemic uncertainty. International Journal of General Systems, 39(6), pp.605-646.
11. Saltelli, A., 2002. Sensitivity analysis for importance assessment. Risk analysis, 22(3), pp.579-590.
12. Harbaugh, A.W., 2005. MODFLOW-2005, the US Geological Survey modular ground-water model: the groundwater flow process (pp. 6-A16). Reston, VA: US Department of the Interior, US Geological Survey.
13. Adcroft, A., Campin, J.M., Doddridge, S.D., Evangelinos, C., Ferreira, D., Follows, M., Forget, G., Hill, H., Jahn, O., Klymak, J. and Losch, M., 2018. MITgcm documentation. Release checkpoint67a-12-gbf23121, 19.
14. Bakker, M., Post, V.+., Langevin, C.D., Hughes, J.D., White, J.T., Starn, J.J. and Fienen, M.N., 2016. Scripting MODFLOW model development using Python and FloPy. Groundwater, 54(5), pp.733-739.
15. Jiang, C., Hu, Z., Liu, Y., Mourelatos, Z.P., Gorsich, D. and Jayakumar, P., 2020. A sequential calibration and validation framework for model uncertainty quantification and reduction. Computer Methods in Applied Mechanics and Engineering, 368, p.113172.
16. Xiao, H., Wu, J.L., Wang, J.X., Sun, R. and Roy, C.J., 2016. Quantifying and reducing model-form uncertainties in Reynolds-averaged Navier–Stokes simulations: A data-driven, physics-informed Bayesian approach. Journal of Computational Physics, 324, pp.115-136.
17. Lelandais, B., Oudot, M.P. and Combemale, B., 2018, October. Fostering metamodels and grammars within a dedicated environment for HPC: the NabLab environment (tool demo). In Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (pp. 200-204).
18. Bezanson, J., Edelman, A., Karpinski, S. and Shah, V.B., 2017. Julia: A fresh approach to numerical computing. SIAM review, 59(1), pp.65-98.
19. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J. and Van Der Walt, S.J., 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature methods, 17(3).

Dorian Leroy is a PhD student doing an international PhD between the JKU Linz (Austria) and the University of Rennes 1 (France), currently in the DiverSE team in Rennes. His research interests lie in the field of Software Language Engineering and include metaprogramming approaches and generic V&V facilities. Contact him at dorian.leroy@inria.fr.

June Sallou is a PhD Student in Software Engineering and Hydrology at University of Rennes 1. Her research interests include Scientific Modelling, Approximate Computing and Environmental Science. Contact her at june.benvegnu-sallou@univ-rennes1.fr.

Johann Bourcier is an Associate Professor of Software Engineering at University of Rennes 1. His research interests in Software Engineering include Self-Adaptive Systems, Model-Driven Engineering, and Distributed and Heterogeneous Software Environments. Contact him at johann.bourcier@irisa.fr.

Benoit Combemale is a Full Professor of Software Engineering at University of Rennes 1. His research interests in Software Engineering include Software Language Engineering, Model-Driven Engineering, and Software Validation & Verification. Contact him at benoit.combemale@inria.fr.