Idea Paper: The Lifecycle of Software for Scientific Simulations

Anshu Dubey
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL 60439
Flash Center for Computational
University of Chicago
Chicago, IL 60637
Email:adubey@anl.gov

Lois C. McInnes
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL 60439
Email:curfman@mcs.anl.gov

***Abstract*—The software lifecycle is a well-researched topic that has produced many models to meet the needs of different types of software projects. However, one class of projects, software development for scientific computing, has received relatively little attention from lifecycle researchers. In particular, software for end-to-end computations for obtaining scientific results has received few lifecycle proposals and no formalization of a development model. An examination of development approaches employed by the teams implementing large multicomponent codes reveals a great deal of similarity in their strategies. This paper formalizes these related approaches into a lifecycle model for end-to-end scientific application software, featuring loose coupling between submodels for development of infrastructure and scientific capability. We also invite input from stakeholders to converge on a model that captures the complexity of this development processes and provides needed lifecycle guidance to the scientific software community.**

I. Introduction

The software lifecycle is a well researched topic with copious literature and many models that serve the specific needs of different type of projects. The objective behind the definition of lifecycle models is to decompose the software development process into distinct stages, where each stage can control its own quality and result in higher quality software overall. Examples include waterfall [14], spiral [3], rapid application development (RAD) [12], agile [1], and several more (see [16] for a list and general description of various software lifecycle models). One class of software development, that of scientific software, is not served well by any of these models. Elements from several of them are useful, with RAD and agile coming the closest, but by themselves none adequately meets the needs of scientific software development. The TriBITS [2] effort has produced an agile lifecycle model for research-driven software development from the perspective of software that downstream becomes a component in a larger software collection. This model is suitable for libraries and other software that implement research ideas. For example, the Blue Brain Project [8], [11] is adapting the TriBITS model to their computational needs. However, many projects exist in scientific domains where software is the means for conducting research instead of being the product or the objective of the research. End-to-end simulation codes fall into this category. They may use libraries and other third-party software as components, but their users have different expectations from the code. An examination of the development approaches of many existing multicomponent scientific codes reveals a great deal of similarity in lifecycle methodologies that predate TriBITS. In this "idea" paper we propose a formalized model based on these approaches that are already informally followed by many groups.

Codes for simulation and analysis are typically used either to understand some phenomenon or process through computational exploration or to help design physical experiments and instruments for similar investigations. These codes often tend to be interdisciplinary endeavors and need more lifecycle elements because they have to contend with many stages that have different levels of complexity. Among scientific simulation codes exists a class of codes that are multiphysics, multiscale, and multicomponent [10]. This set of codes has its own unique challenges, not only because of the complexity of the codes themselves, but also because such codes typically need high-performance computing (HPC) resources for simulation and analysis. The successful codes among these differentiate between *infrastructure*, a set of general-purpose services provided by the code's backbone, and *scientific capabilities*, models of the physical world. These two aspects of codes also have different lifecycle needs. The infrastructure or the framework can follow one of the standard development models. Once designed and developed, it typically has long-term stability. Scientific capability development is more challenging and is characteristic of simulation codes used for scientific discovery.

In this idea paper we outline the elements of various relevant lifecycle models and use them to create a software lifecycle model that meets the needs of software for scientific simulation and analysis. As previously mentioned, variations of the model are already in use in many projects without a formal theoretical basis [5], [4], [7], [13]. We differentiate between submodels for infrastructure and scientific capability development, and we loosely couple these submodels to form a complete end-to-end lifecycle model for an entire scientific simulation code. Our model also includes the concept of feedback from simulation planning, an integral part of the evolution of code requirements. For brevity, the remainder of the document uses the term *scientific software* to represent software for scientific simulation and analysis.

II. Scientific Capability Development Cycle

The first phase of a typical lifecycle model is requirement gathering—a complex undertaking for science research codes that deserves its own model. Scientific software is designed to model phenomena in the physical world, including physical, chemical, or biological processes or their combinations. Scientific experts interested in studying such phenomena formulate a mathematical model that captures the behavior of the system as they understand it. The model may not be complete or fully understood. Because most models do not lend themselves readily to analytical solutions, the mathematical model is discretized so that numerical methods may be applied to find one or more solutions.

Figure 1 shows a schematic of the process in scientific software development that is equivalent to requirement gathering. This example is taken from the development of a model based on partial differential equations. The stages roughly follow the process described above, where the first step is formulating the mathematical model. If the model is simple, the second stage of incorporating approximations may not be needed, although more often approximations are used. Sometimes approximations are introduced to make a model tractable, while in other cases approximations save computation time and may not make a substantive difference to the simulation outcome. In still other cases approximations are used because the corresponding part of the model may not be well understood. The next two stages are discretization and algorithm development. Existing implementations of the needed algorithms in third-party software or libraries may obviate the need for the algorithm development stage. In such situations convergence and stability analysis may not be needed. Once an implementation is available, the computational model should be validated against some calibrated observation. The calibration stage is where the range of valid operation of the model is tested against the implementation. While not every project requires a calibration stage, this phase is often critical.

Figure 1 shows feedback arrows from the validation stage to three of the earliest stages. The process of verification and validation may reflect a need to adjust approximations because some approximations may lead to too much compromise on model fidelity, or it may be found that more approximations can be made without substantive loss of fidelity. Similarly a numerical algorithm may prove to be inadequate or too expensive for the region of interest, or the implementation choices may make it suboptimal. Any of the these situations can lead to resetting the state of the development to the corresponding stage and resuming the cycle from that stage. Thus, scientific code developers work iteratively [15], and requirement specifications evolve throughout the cycle.

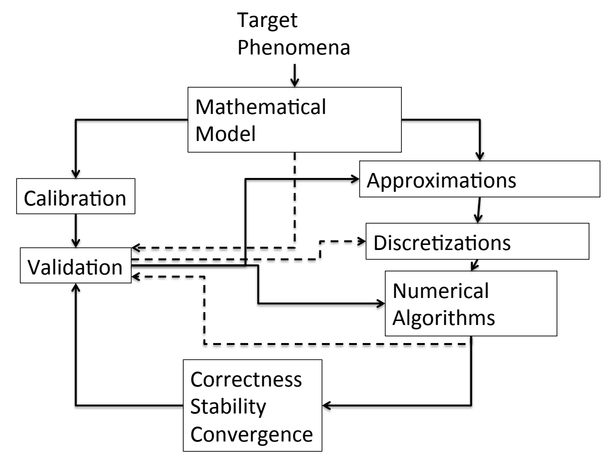

Fig. 1. Schematic of requirement gathering and prototype development in scientific simulation codes

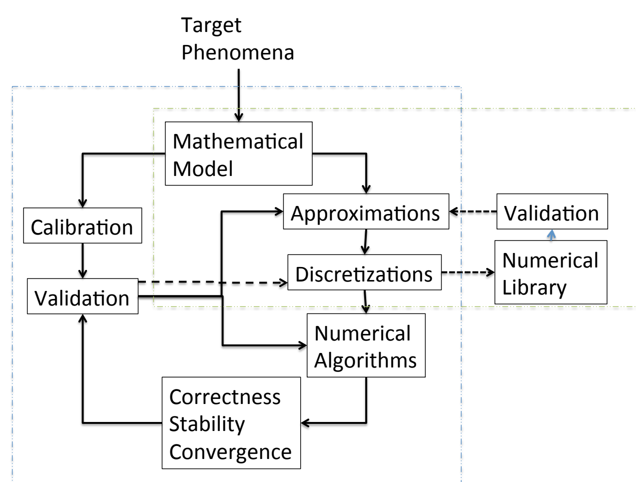

Fig. 2. Schematic of prototype development when the model needs more than one component. Here one component needs a new numerical algorithm, while the other component is using a library. Discretization is the same.

A change in discretization is rare for a component, and hence the arrow is dashed. The other dashed arrows represent the possibilities of bypassing some of the stages. For example, if calibration is not needed, the stage is bypassed for validation. Similarly, if a third-party or library-based numerical algorithm is used, convergence and stability analysis of the algorithm may be unnecessary. In multicomponent codes this kind of cycle may exist independently for different components, or some of the components may share a part of the cycle. Figure 2 shows an example of a partially shared cycle between two components with the same discretization. One component needs calibration and a new algorithm, while the other component can use a preexisting numerical algorithm

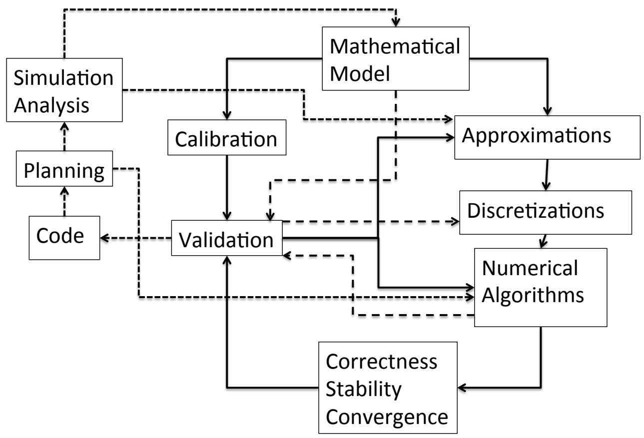

Fig. 3. Larger development cycle, which includes feedback into requirements from new learning in the domain or modifications demanded by resource planning for simulation

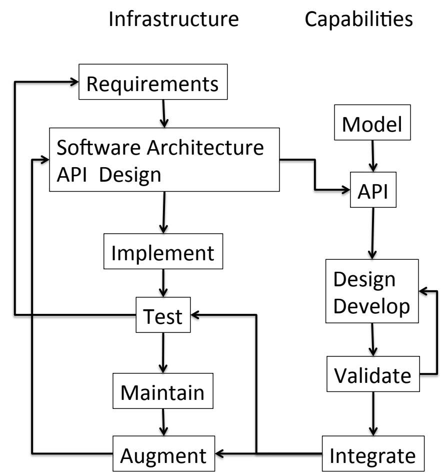

Fig. 4. Infrastructure development cycle and its coupling with the scientific capability development cycle.

and needs no calibration. Figure 3 shows the development cycle augmented with the needs of simulation planning and results of scientific discovery. See [6] for an example of code modifications to meet simulation campaign needs. Simulation planning involves making estimates of resources needed to complete a study, which can be based on past experience or pathfinder runs. Ideally parameters should be built into the code that can be used to customize the behavior of the code to meet the simulation needs. This process can sometimes dictate changes in the approximations and/or algorithms being used, as indicated by the presence of feedback arrows from planning to the corresponding two stages. Scientific discovery can lead to adjustment of the model itself, which can either reset the process or require incremental adjustment. In either case, the entire cycle is affected.

III. Infrastructure Development Cycle

Because the development of scientific software usually responds to an immediate scientific need, the codes typically are employed in production as soon as they have a minimal set of capabilities for some simulation. Similarly, the development of computational modules almost never stops through code lifecycles because new findings in science and math almost continuously place new demands on the codes. While the additions may be incremental when they incorporate new findings into existing features, the additions are often substantial when new capabilities are added. The need for new capabilities may arise from the need for greater model fidelity or from trying to simulate a more complex model. Sometimes a code designed for one scientific field may have enough in common with another field that capabilities may be added to enable its use for the new field.

Irrespective of the cause, coexistence of development and production/maintenance phases is a constant challenge to code teams. While numerical algorithms and solvers can be in a continually evolving state that reflects the advances in their respective fields, fundamentals such as discretization methods, I/O, and other similar supporting services generally are much more stable from the perspective of application scientists. While separate components for support services may not be needed for simple codes with one or two capabilities, separate support services are critical in codes with many capability components that must interoperate with each other and work as a whole. Extracting support services into common infrastructural components is the only way to make the development tractable for such codes. Note that common infrastructure does not imply that every capability must adhere to the same discretization or the same form of I/O or runtime support. Instead, all different manifestations together form the overall infrastructure or backbone of the code.

The lifecycle model for infrastructure development can be any of the prevalent models mentioned in Section I, depending on what is best suited for individual projects. Among scientific communities, the agile model is most popular because even infrastructure is not completely devoid of incremental changes. Not all agile methods apply, but the philosophy of the methodology is well suited to codes that are constantly evolving. The overall development model for such software needs to incorporate coupling between the two modes of development. One possible solution is shown in Figure 4.

In this coupled lifecycle model, a large part of the development cycle of each type (infrastructure and scientific capability) proceeds independently of the other. API design, integration, and integrated testing and maintenance are the coupling points between the two modes of development. We have added a stage for augmentation over and above the usual stages in the infrastructure model to account for incremental changes that are integral to all large scientific projects. The augmentation stage also makes provisions for any changes imposed on the infrastructure when a new scientific capability integrates into the existing code or an existing capability imposes some new constraints due to its own tweaking. Good software discipline requires that different components interact with one another through APIs; therefore that is another coupling point between the two development cycles. A new scientific capability may demand augmentation of the infrastructure API or recognition of the API of the new capability. Both kinds of modification to the API should proceed cooperatively.

IV. Discussion

This proposed development cycle is meant to serve as a starting point for a meaningful discussion about the unique needs of scientific software, in particular multiphysics, multi-component simulation software that runs on HPC resources. This lifecycle model does not capture some of the other aspects of software engineering that are unique to such software. For example, testing of scientific software needs to reflect the layered complexity of the codes. A single box "testing" in the model does not adequately capture that. Similarly, porting a code to new platforms can be a challenging undertaking requiring substantial development resources (see [9] for software engineering and productivity documents relevant to computational science and engineering). Because similar methodology has already found relatively wide use among scientific simulation and analysis code developers, we believe that this lifecycle model—featuring loose coupling between distinct models for infrastructure and scientific capability development—is a good place to start in order to converge on a lifecycle model for the high-performance scientific software community.

Acknowledgments

This work was supported by the U.S. Department of Energy Office of Science Office of Advanced Scientific Computing Research.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (Argonne). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.

References

[1] Agile methodology. http://agilemethodology.org/.

[2] R. A. Bartlett, M. A. Heroux, and J. M. Willenbring. Overview of the TriBITS lifecycle model: A lean/agile software lifecycle model for research-based computational science and engineering software. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–8. IEEE, 2012.

[3] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[4] CASC. SAMRAI structured adaptive mesh refinement application infrastructure. https://computation.llnl.gov/casc/SAMRAI/, December 2007. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory.

[5] A. Dubey, K. Antypas, M. Ganapathy, L. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[6] A. Dubey, A. Calder, C. Daley, R. Fisher, C. Graziani, G. Jordan, D. Lamb, L. Reid, D. M. Townsley, and K. Weide. Pragmatic optimizations for better scientific utilization of large supercomputers. *International Journal of High Performance Computing Applications*, 27(3):360–373, 2013.

[7] Enzo Collaboration, G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, and Y. Li. Enzo: An adaptive mesh refinement code for astrophysics. *ArXiv e-prints*, July 2013.

[8] M.-O. Gewaltig and R. Cannon. Current practice in software development for computational neuroscience and how to improve it. *PLoS Comput Biol*, 10(1):e1003376, 2014.

[9] IDEAS Productivity - Howto Documents. https://ideas-productivity.org/resources/howtos/.

[10] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, X. Jiao, K. Jordan, D. Kaushik, E. Kaxiras, A. Koniges, K. Lee, A. Lott, Q. Lu, J. Magerlein, R. Maxwell, M. McCourt, M. Mehl and Roger Pawlowski and Amanda Peters Randles, D. Reynolds, B. Rivière, U. Rüde, T. Scheibe, J. Shadid, B. Sheehan, M. Shephard, A. Siegel, B. Smith, X. Tang, C. Wilson, and B. Wohlmuth, Multiphysics Simulations: Challenges and opportunities. *International Journal of High Performance Computing Applications*, 27(1):4–83, 2013.

[11] H. Markram. The Blue Brain project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[12] J. Martin. *Rapid application development*, volume 8. Macmillan New York, 1991.

[13] J. D. Moulton, J. C. Meza, M. W. Buksas, M. Day, et al. High-level design of Amanzi, the multi-process high performance computing simulator. Technical report, ASCEM-HPC-2011-03-1, DOE-EM, Washington, DC, 2012.

[14] K. Petersen, C. Wohlin, and D. Baca. The waterfall model in large-scale development. In *International Conference on Product-Focused Software Process Improvement*, pages 386–400. Springer, 2009.

[15] J. Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10(4):517–536, 2005.

[16] Software development lifecycle tutorial. http://www.tutorialspoint.com/sdlc/.