Contents

Chapter 1

Software Process for Multiphysics Multicomponent Codes

1.1 Introduction

The computational science and engineering (CSE) communities have a mixed record of using software engineering and adopting good software practices. Many codes adopt software practices only when making progress without them becomes impossible because of the size and composition of the code In rarer instances code projects start with an awareness of the importance of software process and build it in from the beginning. As more codes have crossed the threshold of being manageable without software engineering they have increasingly been adopting software processes derived from outside the scientific domain. The driving force behind adoption is usually the realization that without using software engineering practices, the development, verification and maintenance of code becomes intractable. State-of-the-art for software engineering practices in CSE codes lags behind that in the commercial software space. There are many reasons for it, but an important one is that not all software engineering practices can be adopted by the CSE code developers.

many software best practices are not well-suited for CSE codes without modification and/or customization, in particular to multiphysics multicomponents codes that run on the large HPC platforms. Sometimes the inherent physics of scientific applications require different software methodologies, at others a premium is placed on performance rather than code architecture. Still others are more sociological and due to the type of institutions where such codes are developed. The challenges for scientific applications range from their architecture to the process for their maintenance and growth. The standard practices adopted by the CSE codes include repositories for code version control, modular code design, licensing process, regular testing, documentation, release and distribution policies and contribution policies. Less frequently used practices include code-review and code-depracation. Agile developmenet methods tend to have limited usefulness in the CSE lifecycle, only a select subset of those have met with success. The degree of adoption and sophistication in using these practices varies among teams. Many of the reasons behind less penetration are discussed in section **??**. Almost all the software engineer-

ing practices mentioned above have to be modified and customized for CSE software. The next two sections outline the challenges that are either unique to, or are more dominant in this domain than elsewhere.

1.2 Domain Challenges

%item diverse algorithms - different data layout needs

Multiphysics codes, by their definition, have more than one mathematical model that they are solving. A typical code combines 3-4 diverse models, the more extreme ones may employ as many as a dozen. In a rare calculation all models work with the same discretization using similar algorithmic approach (for instance stencil computations in explicit PDE solvers). More common is to have models with diverse discretizations and algorithms. Each operator has its own preferred data layout and movement, and it usually differs from those needed by the other operators. Normally these challenges can be mitigated through encapsulation and well defined API's. The outer wrapper layers of the operators can carry out data transformations as needed. There are two factors against taking this approach in CSE codes: (1) physics is not always friendly to encapsulation, and (2) the codes are performance sensitive and wholesale data movement significantly degrades performance.

The CSE simulation codes model the physical world which does not have neat modularization. Various phenonema have tightly coupled dependencies that are hard to break. These dependencies and tight couplings also translate into their mathematical models and it becomes hard to eliminate lateral interactions among code modules implementing the models. An attempt to force encapsulations by hoisting up the lateral interactions to the API level can explode the size of the API. And if not done carefully this can also lead to extra data movement. The module designs, therefore, have to be cognizant of potential lateral interactions and make allowances for them. Similarly the data structures have to take into account the diverse demands placed on them by different operators and carefully consider the trade-offs during software design. Considerations such as these are not common in software outside of CSE.

In a CSE software design, separation of concerns is of utmost importance. Orthogonalizing expertise requirements into different code components allows developers to focus on what they know best. Another natural fallout of this approach is that different dimensions of complexities in the algorithm space are handled separately. The numerical algorithms associated with physics operators are complex because of accuracy and stability concerns, and require mathematical expertise. They are not logically as complex. Whereas machinery for managing the discretizations and interoperability among code components is likely to be less complex numerically, but could be very complex logically. A third axis of concern is parallelization, which brings in some fea-

tures that are unique to CSE codes, such as domain decomposition, aspects of synchronization and dependencies, and performance impact of the design choices. With appropriate separation of concerns not only do these aspects of software development not interfere with one another, they help make the development tractable.

Multiphysics multiscale codes are almost unique in the software world in their need to tightly integrate third party software, which comes in the form of numerical libraries. Because multiphysics codes combine expertise from many domains, the numerical solvers they use also require diverse applied mathematics expertise. It is nearly impossible for any one team to assemble all the necessary expertise. The only form in which such expertise is encoded is in the form of numerical libraries or other third party software. The developers are faced with a choice between integrating third party software, or develop their own technology which is likely to yield inferior quality of solution.

Testing of CSE software needs to reflect the layered complexity that the codes themselves have. The first line of attack is the unit tests where possible. However, as mentioned in chapter **??**, some dependencies are impossible to break in mathematical software, and where such dependences exist a unit test cannot be devised. Testing relies upon no-change tests where minimum possible combination of units are used within the dependence constraints. In effect these minimally combined tests play the same role in the testing regime that unit tests do. Multicomponent CSE software also relies upon integrated and system level testing. This is because the components can be permuted and combined in many different ways, and all configurations have to work within the accuracy and stability constraints.

Another unique aspect of multiphysics CSE software is its need for performance portability. HPC machines are expensive and rare resource, they need to be used efficiently. In order to do that codes should ideally be optimized for each machine. However, yypical lifecycle of a multiphysics software spans many generations of HPC platform lifecycle, which is about 3-4 years. Depending upon the size of the code optimization for a specific target platform can take a significant fraction of the platform lifecycle. During the optimization phase, the code is not available for science. Thus a large fraction of scientists' time is lost in porting and optimizing the code over and over. Another dimension of this problem is that even within the same generation the platforms differ from one another. So machine specific optimization ties a code to one machine. These factors make platform-specific optimizations unattractive. Instead, HPC CSE codes consider the trade-offs and opt to design their software using constructs that perform modestly well across a range of platforms.

1.3 Institutional Challenges

Many adaptations in the software engineering described in the previous section pertained to software design and testing. In particular they spoke to challenges to modularity, performance and unit-testing because of the intertwined nature of the problems being tackled by these codes. However, all challenges faced by the CSE codes are not only because of the nature of problems they solve. Many challenges are specific to the kind of organizations and the research communities where the codes are developed. The most crippling and pervasive challenge faced by CSE codes in general and multiphysics codes in particular is that they rarely get funding for development of software infrastructure. There is evidence that when software is designed well it pays huge dividends in scientific productivity, collected from the miniscule number of projects that secured such funding for software infrastructure design. Even with the evidence the scientific establishment remains unconvinced about the criticality of investment in software engineering. The funding is carved out of scientific goal oriented funding which has its own priorities and time line which end up short-changing the software engineering.

The scientific output is measured in terms of publications which in turn depend upon the data produced by the simulations. Therefore in a project driven purely by scientific objectives, the short-term science goals can lead to compromise on the quality of software design. Quick-and-dirty often triumphs over long term planning. The cost of future lost productivity is not appreciated until it is too late. By the time design deficiencies are realized usually the software has grown too large to remove the deficiencies in any easy way. Software engineering is forcibly imposed on the code, which it at best a band-aid solution. This is another reason why many of the software practices are not embraced by the CSE community.

Another instituitional challenge faced by SE for CSE is the training. Multiphysics codes require a broad range of expertise in domain science from their developers, software engineering is an added requirement. The developers are not trained in SE, many learn them on the job through reading or talking to colleagues. The practices are applied as they understand them, usually picking only what is of most importance for their own development. This can be both good and bad. Good because it sifts out the unnecessary aspects of the SE practice, and bad because it is not always true that the sifted out aspects were really not necessary. It just means that the person adopting the practice did not understand how to use them, or their importance.

Institutional challenges also arise from scarcity of resources and stability. The domain and numerical algorithmic expertise is rarely replicated in a team developing the multiphysics CSE application. Even otherwise, deep expertise in the domain may be needed to model the phenomenon right, and that kind of expertise is hard to come by. Then there is the challenge to communicating

the model to the software engineer if there is one on the team, or to other members of the team with other expertise. It requires at least a few developers in the team who can act as interpreters for the various domain expertise and are able to integrate them. Such abilities take a great deal of time and effort to develop, neither of which are possible in the academic institutions where these codes are typically organically grown. The available human resources in these institutions are post-docs and students who move on, so there is no retention of institutional knowledge about the code. A few projects that do see the need for software professionals struggle to find ways of funding them or to provide a path to professional growth.

The above institutional challenges also provide a clue about why any set software development methodology is hard, and often even undesirable, to adopt in such projects. For example, the principles behind the agile manifesto apply, but not all the formalized process does. These codes are developed by interdisciplinary teams where interactions and collaborations are preferred over process. The code development and its use for science go on in parallel, so the requirements change and there is quick feedback when they do. For the same reason, the code needs to be in working condition almost all the time. However, scarcity of resources does not allow the professional roles in the agile process to be played out efficiently. There is no clear separation between the developer and the client, many developers of hte code are the scientists who use for their research. Because software development goes hand in hand with research and exploration of the algorithms it is impossible to do either within fixed time-frames. This constraint effectively eliminates using sprints. The waterfall model is even less useful because it is impossible to do a full specification ahead of time. The code has to grow and alter organically as the scientific understanding grows, the effect of using technologies are digested and requirements change.

The need for deep expertise, and the fact that the developer of a complex physics module is almost definitely going to leave with possibly no replacement, documentation of various kind takes on a crucial role. It becomes necessary to document the algorithm, the implementation choices, and the range of operation. The general practice of "writing code that does not need inline documentation" does not apply. To an expert in the field, who has comprehensive understanding of the underlying math, such a code might be accessible without inline documentation. But to all others, and there are many who have reasons to look at the code, it would be equivalent to having equations without accompanying explanation. For longevity and extensibility, a scientific code must have inline documentation explaining the implementation logic, and the reasons behind the choices made.

1.4 Case Study: The FLASH Code

1.4.1 Code Design

From the outset FLASH was required to have composability because the simulations of interest needed capabilities in different permutations and combinations. For example, most simulations needed compressible hydrodynamics, but with different equations of state. Some needed to include self-gravity while others did not. An obvious solution was to use object-oriented programming model with common API's and specializations to account for the different models. However, the physics capabilities were mostly legacy with F77 implementations. Rewriting the code in an object oriented language was not an option. A compromise was found by exploiting the unix directory structure for inheritence, where, for a code unit the top level directory defined the API and the subdirectories contained the multiple alternative implementations of the API. Meta-information about the role of a particular directory level in the object oriented framework was encoded in a very limited domain-specific language (configuration DSL). The meta-information also included state and runtime variables requirements, dependences on other code units etc. A "setup tool" parsed this information to configure a consistent "application". The setup tool also interpreted the configuration DSL to implement inheritence using the directory structure. For more details about FLASH's object oriented framework see [**?**, **?**].

FLASH design is aware of the need for separation of concerns and achieves it by separating the infrastural components from physics. The abstraction that permits this approach is very well known in CSE, that of decomposing a physical domain into rectangular blocks surrounded by halo cells copied over from the surrounding neighboring blocks. To a physics operator whole domain is not distinguishable from a box. Another necessary aspect of the abstraction is not to let any of the physics own the state variables. They are owned by the infrastructure that decomposes the domain into blocks. A further separation of concern takes place within the units handling the infrastructure, that of isolating parallelism from the bulk of the code. Parallel operations such as ghost cell fill, refluxing or regridding have minimal interleaving with state update in the blocks from application of physics operators. To distance the solvers from their parallel constructs, the required parallel operations provide an API with corresponding functions implemented as a subunit. The implementation of numerical algorithms for physics operators is sequential, interspersed with access to the parallel API as needed.

Minimization of data movement is achieved by letting the state be completely owned by the infrastructure modules. The dominant infrastructure module is the *Eulerian* mesh, owned and managed by the *Grid* unit. The physics modules query the *Grid* unit for the bounds and extent of the block they are operating on, and get a pointer to the physical data. This arrangement

works in most cases, but gets tricky where the data access pattern does not conform to the underlying mesh. An example is any physics dealing with Lagrangian entities (LE's). They need a different data structure, and the movement of data has nothing in common with the way the data moves on the mesh. The added difficulty is that the entities do need to interact with the mesh, so physical proximity of the corresponding mesh cell is important in distributing the LE's. This is one of the examples of unavoidable lateral interaction between modules. In order to advance, LE's need to get some field quantities from the mesh and then determine their new locations internally. In some applications they have to apply near- and far-field forces, and in some applications they have to pass some information along to the mesh. And after advacing in time they may need to be redistributed. FLASH solves this conundrum through keeping the LE data structure extremely simple, and using argument passing by reference in the API's. The LE's are attached to the block in the mesh that has the overlapping cell, an LE leaves its block when its location no longer overlaps with the block. Migration to a new block is an independent operation from everything else that goes on with the LE's In FLASH parlance this is the Lagrangian framework (see [**?**] for more details). The combination of *Eulerian* and *Lagrangian* frameworks that interoperate well with one another has succeeded in largely meeting the performance critical data management needs of the code.

1.4.2 Software Process

The software process of FLASH has evolved organically with the growth of the code. For instance, in the first version there was no clear design document, the second version had a loosely implied design guidance, whereas the third version documented the whole design process. The third version also published the developer's guide which is a straight adaptation from the design document. Because of multiple developers with different production targets, versioning repository was introduced early in the code life cycle. The repository used has been SVN since 2003, though its branching system has been used in some very unorthodox ways to meet peculiar needs of the Flash Center. Unlike most software projects where branches are kept for somewhat isolated development purposes, FLASH uses branches also to manage multiple ongoing projects. This particular need arose when there were four different streams of physics capabilities being added to the code. All projects needed some code from the trunk, but the code being added was mostly exclusive to the individual project. It was important that the branches stay more or less in sync with the trunk and that the new code be tested regularly. This was accomplished by turning the trunk into essentially a merge area, with a schedule of merge from individual branches, and an intermediate branch for forward merge. The path was tagged-trunk =¿ forward-branch =¿ projects =¿ merge into trunk =¿ tag trunk when stabilized. Note that the forward branch was never allowed a backward merge to avoid the possible inadvertent breaking of code for one

project by another one. For the same reason the project branches never did a forward merge directly from the trunk.

Testing is another area where the standard practices do not adequately meet the needs of the code. Many multiphysics codes have legacy components in them that are written in early versions of Fortran. Contrary to popular belief, a great deal of new development continues in Fortran because it still is the best HPC language in which to express mathematical algorithms. All of solver code in FLASH is F90, so the general unit test harnesses aren't available for use. Small scale unit tests can only be devised for infrastructural code because all the physics has to interact with the mesh. Also, because regular testing became a part of FLASH development process long before formal incorporation of software engineering practices in the process, FLASH's designation of tests only loosely follows the standard definitions. So a unit test in FLASH can rely on other parts of the code, as long as the feature being tested is isolated. For example testing for correct filling of halo cells uses a lot of AMR code that has little to do with the halo filling, but it is termed unit test in FLASH parlance because it exclusively tests a single limited functionality. The dominant form of regular testing is integration testing, where more than one code capability is combined to configure an executable. The results of the run are compared against pre-approved results to verify that changes are withing a specified acceptable range. Because of a large space of possible valid and useful combinations selected of tests becomes a challenging task. FLASH's methodology for test design and selection is described in detail in [**?**], and follows the matrix method described in chapter **??**

FLASH's documentation takes a comprehensive approach with having a user's guide, developer's guide, robodoc API, inline documentation, and online resources. Each type of documentation serves a different purpose and is indispensable to the developers and users of the code. There are scripts in place that look for violations of coding standards and documentation requirements. User's guide documents the mathematical formulation, algorithms used and instructions on using various code components. The user's guide also includes examples of relevant applications explaining the use of each code module. The developer's guide specifies the design principles and coding standards with an extensive example of the module architecture. Each function in the API is required to have a robodoc header explaining the input/output, function and special features of the function. Except for the third party software, every non-trivial function in the code is required to have sufficient inline documentation that a non-expert can understand and maintain the code.

FLASH effectively has two versions of release - internal, which is close to the agile model, and general, which is no more than twice a year. The internal release amounts to tagging a stable version in the repository for the internal users of the code. This is signal to the users that a forward merge into their production branch is not going to break the code. The general releases have a more rigorous process which makes them infrequent. The general releases undergo some amount of code pruning, check for compliance with coding and

documentation standards and stringent requirements from the testing process. They are expensive in terms of developers resources. The dual model ensures that the quality of code and documentation are maintained without unduly straining the team resources, while near continuous code improvement is still possible for ongoing projects.

1.4.3 Policies

In any project, policies regarding attributions, contributions and licensing matter. In CSE arena intellectual property rights, and interdisciplinary interactions are additional policy areas that are equally important. Some of these policy requirements are a direct consequence of the cathedral model [] of development that majority of CSE publicly distributed software follow. Many arguments are forwarded for dominance of the cathedral model in this domain, the most compelling one relates to maintaining the quality of software. Recollect that the developers in this domain are typically not trained in software engineering, and software quality control varies greatly between individuals and/or groups of developers. Because of tight, and sometimes lateral, coupling between functionalities of code modules a lower quality component introduced into the code base can have disproportionate impact on the overall reliability of output produced by the code. Strong gate-keeping is desirable, and that implies having policies in place for accepting contributions. FLASH again differentiates between internal and external contributors in this regard. The internal contributors are required to meet the quality requirements such as coding standards, documentation, and code verification in all of their development. Internal audit processes minimize the possibility of poorly written and tested code from getting into a release. The internal audit also goes through a periodic pruning to ensure that bad or redudant code gets eliminated.

The external contributors are required to work with a member of the internal team to include their code in the released version. The minimum set required from them is: (1) code that meets coding standards, has been used or will be used for results reported in peer-reviewed publication, (2) at least one test that can be included in the test-suite for nightly testing, (3) documentation for user's guide, robodoc documentation for any API functions and inline documentation explaining the flow of the control, and finally (4) a commitment to answer questions on users mailing list. The contributors can negotiate the terms of release, a code section can be excluded from the release for a mutually agreed period of time to enable the contributor to complete their research and publish their work before it the code becomes public. This policy permits the potential contributors to be freed from the necessity of maintaining their code independently, while still retaining control over their software until the agreed upon release time. As a useful side effect their code remains in sync with the developments in the main branch between releases.

There is another model of external contribution to FLASH that is without any intervention from the core gate-keeping team. In this model anyone can

stage any FLASH compatible code on a site hosted by them. The code has no endorsement from the distributing entity, the Flash Center, which does not take any responsibility for its quality. The Flash Center maintains a list of externally hosted "as-is" code sites, the support for these code sections are entirely the responsibility of hosting site.

The attribution practices in CSE are somewhat ad-hoc. For many developers, the only metric of importance are the scientific publications that result from using the software. When a team is dominated by such developers proper attribution is not given enough importance or thought. Other teams also employ computing professionals whose career growth depends upon their software artifacts, and publications describing their algorithms and artifacts. FLASH falls into the latter category, but the attribution policy does not reflect meet this challenge adequately. All contributors' names are included in the author list for the user's guide, the release notes explicitly mention new external contributions and their contributors, if any, for that release. Internal contributors rely upon software related publications for their attribution. This policy has not always worked well, and one of worst side effects has been citations scewed in favor of early developers. Users of FLASH cite a paper published in 2000 [**?**] which does not include any of the later code contributors in its author list, who are, therefore, deprived of legitimate recognition for citations. Many major long running software projects have this problem, which is peculiar to the academic world where these codes reside and are used.

1.5 Generalization

: Not all of the solutions described in the earlier sections for CSE specific challenges are generalizable to all scientific software, but the vast majority of them are. This is borne out by the fact that at a workshop on community codes in 2012 [], all represented codes had nearly identical stories to tell about their motivation for adopting software engineering practices and the ones that they adopted. This was true irrespective of the science domains these codes served, the algorithms and discretization methods they used and communities they represented. Even their driving design principles were similar at the fundamental level though the details differed. The codes represented state-of-the-art in their respective communities in terms of both model and algorithmic research incorporated and the software engineering practices. Note that these are the codes that have stood the test of time and won the respect in their respective communities. They are widely used and supported, and have more credibility for producing reproducible reliable results than smaller individualistic efforts. Therefore, it is worthwhile to discuss those practices in this chapter. At a minimum they provide a snapshot of the state of large scale

computing and its dependence of software engineering in the era of relatively uniform computing platforms.

One practice that is universally adopted by all community codes and other large scale codes is versioning repositories. That is worthy of mention here because even this practice has not penetrated the whole computational science community. There are many small projects that still do not use versioning, though their number is steadily decreasing. Other common practice is that of licensing for public use and most codes are freely available to download along with their source. Testing is also universal, though the extent and methodologies for testing vary greatly. A general verification and validation regime is still relatively rare, though regression testing is more common. Unit tests are less common than integration tests and bounded-change tests. Almost all codes have user level documentation and user support practices in place. They also have well defined code contribution policies.

Another feature that stands out is the broader design philisophy of all multiphysics codes. Every code exercises separation of concerns between mathematical and structural parts and between sequential and parallel parts. In almost all cases this separation is dictated by the need to reduce complexity for efforts needing specific expertise. Also, all the codes have basic backbone frameworks which orchestrate the data movement and ownership. This is usually driven by the need for maintenance and flexibility. And where it is realized well it provides extensibility - the ability to add more physics and therefore greater capabilities and fidelity in the models being computed. Majority of frameworks are component based with composability of some sort. This is because different models need different capability combinations. Most codes use self-describing IO libraries for their output to facilitate the use of generally available analysis and visualization tools.

The degree to which teams from vastly different scientific domains producing community codes have arrived at essentially similar solutions is remarkable. It points to a possibility that seemingly diverse problems can have a uniform solution if they are trying to achieve similar objectives. For the codes highlighted in this section, the objectives were capabilities, extensibility, composability, reliability, portability and maintainability. They were achieved through design choices concious of trade-offs, most often with raw performance that individual components or specific platforms were capable of. The lesson here is that similar objectives can yield a general solution even if there is great diversity in the details of the individual problem. It is not beyond the realm of possibility that similar generalized solution will emerge for the next generation software faced with heterogeneous computing described in the next section.

1.6 Additional Future Considerations

: One of the aspect of software design that is a unique requirement of the CSE domain is fast becoming its biggest challenge - performance portability. In the past machine architectures were fairly uniform across the board for large stretches of time. The first set of effective HPC machines in routine use for scientific computing were all vectors machines. They later gave way to parallel machines with *risc* processor as their main processing element. A code written for one machine of its time, if portable, would have reasonable performance on most contemporary machines. The abstract machine model to which the codes were programming was essentially the same for all machines of each era. It is true that wholesale changes had to occur in codes for transitioning from vector to risc-parallel machines. But it was a transition from one long-term stable paradigm to another long-term stable paradigm. And the codes were not as large as the multiphysics codes of today. So the transition took time, and the codes that adapted well to the prevailing machine model thrived for several years.

That landscape is about to change completely. Now there are machines in the pipeline that have deep enough architectural differences among them that one machine model cannot describe their behavior. Even within a machine heterogeneity of various kinds may exist. Because the codes are significantly larger than the last time such drastic changes had occurred in the computing platforms, the challenge is of a completely different magnitude. More importantly, some aspects of the challenges are not unique to the large multiphysics codes. Because the deep architectural changes are occurring at the level of nodes that will go into all platforms, the change is ubiquitous and will affect everyone. There portability in general and performance portability in particular is an issue for everyone. At this writing the impact of this paradigm shift is not fully understood. Means of combating this challenge are understoon even less. There is a general consensus that more programming abstractions are necessary not just for the extreme scale, but also for small scale computing. The unknown is which abstraction or combination of abstractions will deliver the solution. Many solutions have been proposed, for example [] (also see [**?**] for a more comprehensive and updated list). Of these, some have undergone more testing and exercise under realistic application instances than others. None of the approaches provide a good road map for a general solution that can broadly applicable in the ways that optimizing compilers and MPI had provided in the past.

Bibliography