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 when the size and composition of the code makes it impossible to make progress without them. 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 software for software engineering practices in CSE codes lags behind that in the software industry. There are many reasons for it, but the primary 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 largest 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 more limited usefulness in the CSE lifecycle, only a select subset of those are used. 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 engineering

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.

And finally the institutional challenges 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 com-

municating 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 interaction to develop, neither of which are possible in the academic institutions where these codes are typically organically grown. The model of human resource there is that of post-docs and students with no software professionals who can retain the institutional knowledge about the code. This, and the fact that the developer of a complex physics module is almost definitely going to move on 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 equivalnt of having equations without accompanying explanation in the text. 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 the 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 di-

rectory 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 form, and surrounding each of the subdomains with halo cells copied over from the surrounding neighborin subdomains. To the physics whole domain is not distinguishable from a sub-domain. It is also important not to let any of the physics own the state variables. They are owned by the infrastructure, which decomposes the domain into blocks. A further separation of concern takes place within the units handling the infrastructure, that of isolating the parallel aspects from 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. This does impose bulk synchronous communication model on the code, and may need to be modified.

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 nearest cell, an LE leaves its block when its location no longer overlaps with the block. Migration to the 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 meeting all the performance critical data management needs of the code so far.

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 production projects. This particular need arose because at a point there were four different streams of production simulations going on simulataneously in three different scientific domains. All projects needed some stable code from the trunk, but also needed some development flexibility in their own branches. And it was understood that very little of the code added to the branches will make its way back into the repository. This would have been easy with the distributed repositories that have become available relatively recently, but it is difficult in SVN. This was accomplished by turning the trunk into essentially a merge area, with a separate *production* branch becoming the base for forward code merges to the individual projects. The path was tagged-trunk =¿ production =¿ projects =¿ merge into trunk =¿ tag trunk when stabilized. Note that the production branch was never allowed a backward merge to avoid the possible inadvertent breaking of code for one project by another one.

1.4.3 Policies

1.5 Generalization

: this section will discuss those aspects of FLASH solutions that are generalizable

---

1.6 Additional Future Considerations

: How the software, design and policies might need to change in Future.

Bibliography