# Integration Strategies for Computational Science & Engineering Software

Roscoe A. Bartlett
Department of Optimization and Uncertainty Qualification
Sandia National Laboratories
PO Box 5800, Albuquerque, NM 87185-1318
http://www.cs.sandia.gov/~rabartl
rabartl@sandia.gov

## Abstract

*In order to make significant progress in solving challenging problems in Computational Science & Engineering (CS&E), we need to integrate a large amount of software written by different groups of experts. Modern Lean/Agile methodologies would seem to provide a good foundation for research-driven development of complex CS&E software. Here, we describe issues related to the integration of CS&E software and propose different integration processes tailored to the special challenges in CS&E. We also describe practical experience with some of these tailored integration strategies related to Trilinos and some of its important application customers at Sandia National Labs.*

## 1. Introduction

One of the major bottlenecks that limits the use of state-of-the-art algorithms from various disciplines of CS&E is software engineering and integration. We can no longer expect a single group of developers to create and maintain cutting-edge CS&E software which includes the best in meshing [8], discretizations [1], adaptivity [21], advanced (embedded) numerical algorithms [3], and new computer architectures [13]. Software written by experts from a variety of different specialties and organizations needs to be integrated together in order to apply the full state-of-the-art in CS&E. We need to consider how to adapt Lean/Agile software engineering methods [6, 7, 9, 11, 15, 18, 20] to tailor them to the CS&E environment.

The scope of the software that needs to be integrated into a single complex CS&E application is too large to realistically be developed under a single blanket of full continuous integration (CI) [9]. The CS&E software partitioning model discussed in this paper involves one or more applications (APPs) and a number of different third-party libraries (TPLs), forming Customer/Supplier relationships [10, Chapter 14]. An APP typically represents a complete capability that is suited to solve an important problem. Each TPL represents a different type of state-of-the-art CS&E capability used by the APP. It is assumed that the APP and each TPL are developed under their own separate CI processes.

Several different software integration strategies for CS&E software are described as well as a proposed variation on CI that may be well suited to many especially challenging CS&E environments. Each of these approaches is outlined and some experience with different integration strategies at Sandia National Labs is described. Each of these integration strategies will be appropriate in different situations and various criteria should be considered when selecting a specific software integration strategy.

The trend in CS&E is to develop more and more complex algorithms and software so the problem of software engineering and integration will only become more critical in the coming years. We need to inject stronger software engineering discipline tailored to the modern CS&E environment if we are going to be able to integrate and apply the best that CS&E has to offer.

## 2. Overview of the CS&E software engineering environment

Here, some of the special issues that make large-scale CS&E software development and integration challenging are described. While the development of CS&E software in other environments is also important, this paper deals more specifically with CS&E development environments which include both research and applied activities, such as is common in a national labs setting like Sandia National Labs (SNL). Such CS&E environments are complex, especially with regard to software integration issues, for a number of reasons.

a) CS&E is a mix of production applications and algorithms research. Conducting cutting-edge research that impacts important real applications is challenging. With algorithm research you want to have unfettered access to change the software at will in order to quickly try new ideas. However, production development requires more careful controls to insure that important capabilities are preserved with every change. It is very difficult to integrate a mix of software with different maturity levels. However, modern Lean/Agile methods provide a means to develop high-quality research-driven software from the very beginning of its life cycle and therefore allow it to be integrated with production software throughout.

b) CS&E practitioners have a mix of backgrounds in science, physics, applied math, computer science, and other disciplines. This large variability in specialized backgrounds makes collaborative software development difficult. Each CS&E discipline often has a complex set of specialized theory and jargon and it can be hard to come up with common models that can be used to form the basis for integrated software. The ideas from Domain Driven Design (DDD) [10] need to be carefully applied to CS&E to help overcome these integration and maintenance difficulties.

c) Great variability exists between CS&E practitioners in knowledge and interest of basic software engineering issues like coding standards and development practices. For new algorithms to have real impact in CS&E, they must be implemented in high-quality software and be used to solve problems people care about. However, it is common for CS&E practitioners to be passionate about the particular discipline they specialize in, but have very little interest in important software engineering issues. Yet, in many CS&E organizations, these individuals are expected to be able to write software that will be used to solve real problems but there is a natural tendency for many of these individuals to create poor-quality software. Low quality software is difficult to keep integrated and severely damages the entire CS&E software development process.

d) CS&E involves a variety of complex algorithms. Not even the best single PhD can understand all of the different areas well eno¡ugh to write software for state-of-the-art algorithms for each area. This goes well beyond the basic computer science knowledge. Therefore, only a handful of specialized PhD developers can create and modify a complex CS&E algorithm in their discipline. The complexity of these algorithms also presents software integration challenges because it can be difficult to construct interoperable interfaces for such algorithms.

e) The CS&E environment places a strong emphasis on performance involving floating-point computations. CS&E computations are often performed on very expensive massively parallel computers and can take a long time to run. Therefore, computation performance is often a critical factor in CS&E software. Floating-point computations are the cornerstone of CS&E software. Even with formal floating-point standards like IEEE 754 in place, the strong need for high flop rates, along with differences in computer architectures and system software, negates the possibility of strictly defining the binary output from such programs [12]. Therefore, one typically expects to get different binary answers when porting the software to different platforms and even on the same platform when changing compiler optimization options. This presents great challenges in developing strong portable test suites and in performing fundamental software verification. Floating-point creates some difficult software integration challenges, especially when combined with other factors.

f) Complex nonlinear models are commonly used in CS&E and present a number of challenges in the development and maintenance of CS&E software. For example, a set of nonlinear equations can have more than one solution [16] and algorithms that solve these equations can find any one of these solutions or no solution at all. Nonlinear models can also posses bifurcations [5, 22], non-convexities [16], ill-conditioning [16] and other phenomenon that can make the behavior of numerical algorithms on these problems quite unpredictable. In some cases, these phenomenon can result in large changes in the behavior and output from algorithms involving these models. Therefore, the inclusion of complex nonlinear models and functions, coupled with floating-point computations, present some unique software engineering and integration challenges for CS&E software.

g) Close collaboration between CS&E practitioners from different disciplines is needed to solve hard problems. Often, practitioners from several different disciplines must work together on a single integrated code base in order to develop software and to work through issues that arise in hard problems. Oftentimes, significant development is needed in order to address the underlying issues. Therefore, each specialized CS&E developer needs to have access to a very recent version of their APP and TPL codes. This type of cross-discipline collaborative effort is massively easier if the up-to-date sources of the APP and relevant TPLs can be developed together in many cases. The need to have access to the integrated development versions of both the APP and the TPL presents a challenge for common models of software integration being advocated in the broader software engineering community.

The combination of some of the special properties of CS&E software described above make large-scale software integration very difficult. Some of the special software integration challenges are described in more detail in Section 4.

## 3. Overview of general software integration approaches

Before discussing the special software integration challenges for CS&E software in Section 4, first let's consider the current generation of Agile integration approaches. The foundation for modern Agile software integration methods is *Continuous Integration* (CI) [9]. CI comes in two main varieties; synchronous CI and asynchronous CI. *Synchronous CI* (SCI) requires developers to fully integrate and test their changes before each check-in. *Asynchronous CI* (ACI) involves developers doing much less thorough testing before each check-in. A CI server detects a check-in, proceeds to checkout, build, and run a more substantial test suite, and then informs developers if anything fails. SCI is the premiere CI method in terms of code stability, but ACI can scale to larger projects at the cost of greater code instability. In projects where ACI starts to produce failing builds too often, other CI-like methods may be considered [17].

At some point the size of a project will become too large to realistically apply any reasonable single CI method and other less-than-full CI methods must be considered. In these cases, the best approach is to partition the code base into distinct pieces with carefully designed interfaces and then to define appropriate less-than-full CI methods to keep the software integrated on a reasonable (but not continuous) schedule. Eric Evans in [10, Chapter 14] describes a number of different code partitioning and staged integration strategies. The strategy that is most applicable to the type of CS&E software environment being considered in this paper is the Customer/Supplier relationship where the APP and TPL play the roles of the customer and supplier, respectively.

The key to the success of the Customer/Supplier relationship, as stated in [10, Chapter 14], is:

> Jointly develop automated acceptance tests that will validate the interface expected. Add these tests to the upstream team's test suite, to be run as part of its continuous integration. This testing will free the upstream team to make changes without fear of side effects downstream.

The foundation of Customer/Supplier therefore is the automated acceptance test suite which is co-developed through a collaboration between the customer and the supplier which becomes part of the supplier's own test suite. While a co-developed acceptance test suite can be very effective in helping to preserve the interface specification between the customer (APP) and supplier (TPL) codes, it is questionable to what extent such a test suite can effectively substitute in place for the customer's own full test suite in the context of CS&E software. It is clear that every supplier code should contain integrated tests that attempt to fully validate the software against the customer's requirements. However, as stated in Section 2 and argued more strongly in the next section, CS&E software presents a special set of challenges that makes it very difficult, if not impossible, to write an affordable effective acceptance test suite that is independent of the APP code's own test suite. Note that in the end, the only test suite that ultimately matters is the APP's. Customers and end users will be unimpressed if all of the TPL's acceptance and other tests pass with flying colors but a significant number of the customer's own tests fail after an upgrade of the TPL. The primary purpose of the TPL's acceptance and other tests is to try to reduce the risk and likelihood that changes to the TPL will break the APP, but they are in no way substitutes for the APP's own test suite, especially in CS&E software.

## 4. Special software integration challenges in CS&E software

CS&E software is more fundamentally difficult to keep integrated than in almost any other domain. Some of the contributing factors for this were outlined in Section 2. The use of a Customer/Supplier co-developed acceptance test suite, which is the foundation for the decoupled Customer/Supplier relationship described in Section 3, will be fundamentally less effective in advanced CS&E environments. While all of the issues discussed in Section 2 contribute to this, it is the issues related to a) complex algorithms, b) complex nonlinear models and functions, and c) non-unique floating-point computations that change between compilers and platforms that are the most significant. These three issues conspire together to make it very difficult to write good tests for CS&E software and then to keep them validated as changes are made.

The influence of non-unique floating-point computations in the presence of these other factors has lead CS&E developers to declare a special type of failing test; the *diff*ing test. A test is declared to *diff* when the code seems to run correctly but some numerical error check exceeds the allowed tolerance. Many CS&E developers argue that a diffing test is somehow less serious than an otherwise failing test. However, there is very little practical difference between a diffing test and a failing test and any casual dealing with such issues can compromise the very foundation of CS&E software verification. A diffing test must be scrutinized and fixed in the same way as any other failing test.

To demonstrate these problems inherent in CS&E software, consider a complex transient simulation based on a model derived from a complex set discretized partial differential equations (PDEs). Many of these models exhibit all of the complex behavior mentioned in Section 2. In these types of problems, seemingly small changes in the structure of the algorithms amplified by variations in floating-point computations can cause large changes in the behavior of the algorithms and result in large changes in the final output which may cause even well designed tests to diff. In many cases, changes to complex CS&E algorithms that actually improve solution performance may cause diffs. Every such diff that occurs after the integration of a new TPL release must be carefully addressed. It is not uncommon for an upgrade of a TPL to cause dozens or even hundreds of diffing tests which then take an extreme amount of effort to resolve.

Because of the complex nature of the CS&E environment describe above, it is very costly and/or ineffective to develop an acceptance test suite that can successfully take the place of the APP's own test suite. Therefore, the most effective and affordable acceptance test suite for a TPL is often the APP's own test suite.

## 5. Software integration approaches for CS&E software

The following subsections describe three different software integration approaches for CS&E software (outlined in Figures 1–3).

### 5.1. APP + TPL Release with Punctuated TPL Upgrades

The most common way that TPLs are integrated with APPs in the CS&E environment is to develop the APP against a static release of the TPL and then to perform punctuated transitions to new TPL releases, which is depicted in Figure 1. In this figure, the two flat horizontal lines labeled "TPL Head (Dev)" and "APP Head (Dev)" represent the main development (Dev) lines for a TPL and the APP, respectively. The curved lines coming off of the main development line are release branches and complete release cycles for APP and TPL are shown.

In this approach, little to no testing is done between the development versions of the APP and the TPL. After a TPL release is put out, the APP developers attempt to transition to the new release. If releases of the TPL are put out with sufficient frequency, then the cost of performing the transition and the risk of experiencing a regression can be relatively low. However, it is very common in CS&E environments for very long periods of time and large batches of changes to be included between major releases of a TPL. In some cases, as much as a year and thousands of changes can occur between major releases. This can result in very difficult transition periods when trying to upgrade for many of the reasons discussed in Section 4. In some instances, either the APP must accept one or more regressions in some existing capabilities (in order to get access to new capabilities in the new TPL release) or the upgrade to the new TPL release must be abandoned altogether.

Even with all of these problems, using the APP + TPL Release with Punctuated TPL Upgrades approach is perfectly adequate in many situations. The need for more frequent integration testing in many other situations, however, brings us to the next integration approach.

### 5.2. APP + TPL Release and Dev Daily Integration

The high costs and risks associated with the APP + TPL Release with Punctuated TPL Upgrades approach described in Section 5.1 can be largely mitigated by putting a process in place to keep APP Dev up to date with TPL Dev on a regular basis. Figure 2 shows the timeline of this APP + TPL Release and Dev Daily Integration process [2]. APP developers still develop against a static TPL release but nightly testing and reporting are used to also keep APP Dev + TPL Dev integrated. Testing is extended to the new TPL release branch when it is created, and then the switch is made to the new TPL release after the release is finalized.

The primary advantages of this approach are a) all changes to the TPL are tested with the APP's current test suite every 24 hours, b) transitions to new TPL releases involve little effort, and c) there is less risk of experiencing regressions after each TPL upgrade. However, there are still some significant shortcomings with this approach which are a) extra computing resources are needed to test the APP against two or three different versions of the TPL at the same time, and b) testing of APP Dev + TPL Dev is often scaled back due to lack of resources (which in turn increases the changes of the APP experiencing a significant regression after the next TPL release).

Even with the shortcomings in the APP + TPL Release and Dev Daily Integration process described above, the improvements over the APP + TPL Release with Punctuated TPL Upgrades process are significant (see Section 6). If reducing the costs and risks in upgrading to new TPL releases were the only issue, then the daily integration process described above is fairly adequate. However, the primary shortcoming of this process is that it limits the level of detailed software collaboration that can be achieved between the APP and the TPL. It does not adequately support the ambitious collaborations between APP and TPL developers needed to produce the most effective CS&E research and software. To address these issues, an even closer level of integration and development process is needed which is described in the next section.

### 5.3. APP + TPL Almost Continuous Integration

The APP + TPL Release and Dev Daily Integration process addresses a number of important CS&E software inte-

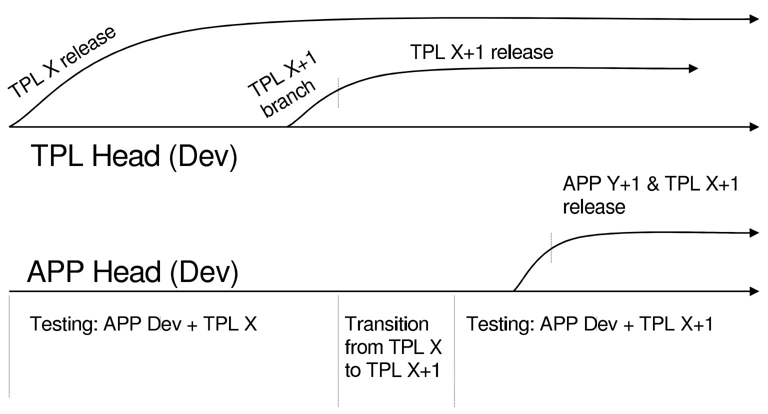

**Figure 1. Timeline for the APP + TPL Release with Punctuated TPL Upgrades process**

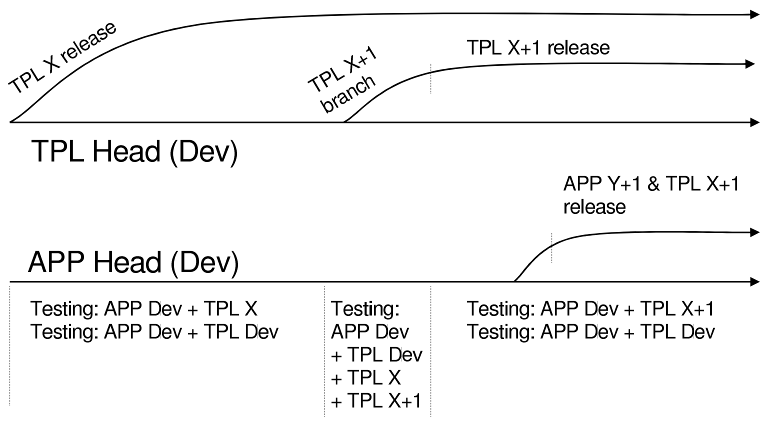
TPL X release
TPL X+1 branch
TPL X+1 release
TPL Head (Dev)
APP Y+1 & TPL X+1 release
APP Head (Dev)
Testing: APP Dev + TPL X
Testing: APP Dev + TPL Dev
Testing: APP Dev + TPL Dev + TPL X + TPL X+1
Testing: APP Dev + TPL X+1
Testing: APP Dev + TPL Dev

**Figure 2. ¡ Timeline for the APP + TPL Release and Dev Daily Integration process**

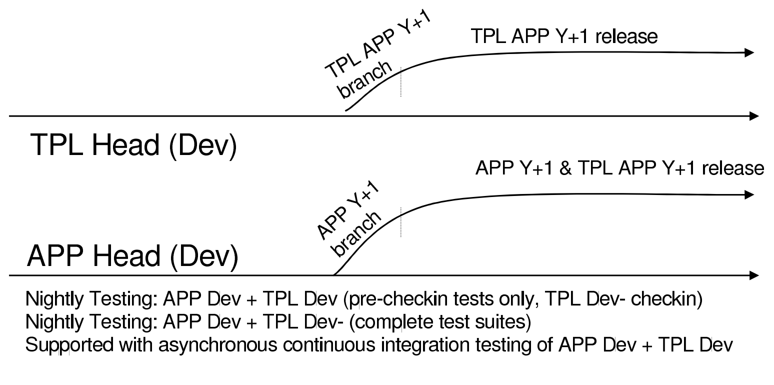
TPL APP Y+1 branch
TPL APP Y+1 release
TPL Head (Dev)
APP Y+1 & TPL APP Y+1 release
APP Y+1 branch
APP Head (Dev)
Nightly Testing: APP Dev + TPL Dev (pre-checkin tests only, TPL Dev- checkin)
Nightly Testing: APP Dev + TPL Dev- (complete test suites)
Supported with asynchronous continuous integration testing of APP Dev + TPL Dev

**Figure 3. Timeline for the APP + TPL Almost Continuous Integration process**

gration problems. As long as the TPL only provides a useful but not pivotal role in the APP, this process is quite adequate. What is not well supported, however, are a number of important more collaborative use cases where the TPL plays a more significant role in the APP. For example, if the TPL is used to define an important architectural aspect of the APP, then changes to the TPL will not be able to impact main-line APP developers until after the next major TPL release. Also, there are examples where it is desirable to be able to carefully refactor the APP and move some parts of it into the TPL so that it can be reused in other APPs. These and other more challenging collaborations need a closer form of software integration. However, the scope of the software involved and the separation of the APP and TPL development teams may be such that doing full CI is still impractical.

If full CI between APP Dev and TPL Dev is impractical, how close can we get to full CI while still maintaining a clear and important separation and also allowing us to address even the most challenging collaborative use cases? Before we look into answering this question, let's first consider some important requirements. We want to allow the majority of the APP and TPL developers to continue to work independently from each other. We need to preserve the important property of SCI where the APP developers can have high confidence that the code they check out from their version control (VC) repositories will build and pass the pre-check-in test suite. In addition, we need to allow a smaller subset of more expert APP and TPL developers to be able to co-develop APP Dev and TPL Dev together and make arbitrary changes.

Here, a less than full CI approach called Almost Continuous Integration is proposed to address the requirements stated above. Almost Continuous Integration is a strengthening of the Customer/Supplier relationship between the APP and the TPL where the full acceptance test suite for the TPL becomes the APP's actual test suite. APP Dev is developed against a very recent version of TPL Dev referred to as TPL Dev-. The APP development team keeps its own VC repository storing TPL Dev- which is a snapshot of TPL Dev from a few minutes to a few days old. Regular APP developers checkout and build against TPL Dev-. The code in the APP's TPL Dev- and APP Dev VC repositories is always guaranteed to build and pass the SCI pre-check-in test suite and TPL Dev- is always updated in a way that preserves this. First, a nightly testing process automatically checks out TPL Dev from the main TPL VC repository along with APP Dev from its own VC repository and then trys to build and run the pre-check-in test suite. If this passes, the updates are commited to the TPL Dev- VC repository. After this, the APP's more rigorous test suites are run using APP Dev + TPL Dev-. However, if APP Dev + TPL Dev failes for any reason, the commit to the TPL Dev- VC repository is skipped and notifications are sent out.

The other important situation where TPL Dev- gets updated is when APP and TPL co-development take place (depicted in Figure 4). Here, sources are checked out from both the APP's and the TPL's VC repositories. The two TPL versions from the TPL Dev- and the main TPL Dev VC repositories are merged into a single TPL working directory. The co-developer is then free to make whatever changes are necessary to the APP and TPL sources and test suites. Changes and enhancements across the entire TPL and APP can be made, and software can freely and incrementally be moved between the APP and TPL code bases. Since TPL Dev and APP Dev are both being changed, both pre-check-in test suites must be run and pass before checking in. The changes in the TPL are checked into both the APP-owned TPL Dev- and main TPL Dev VC repositories. The additions and improvements in APP Dev + TPL Dev- are then immediately available to other APP developers.

The only logical way to handle releases of the APP and the TPL is to put out APP-specific TPL releases to coincide with the APP releases (depicted in Figure 3). Putting out multiple frequent TPL releases is perfectly consistent with Agile best practices [7, 9, 18].

The numerous advantages to the Almost Continuous Integration process proposed here are that a) all changes to TPL Dev are typically tested against the APP's test suites every 24 hours, b) there is a lower probability of experiencing a regression after a TPL release than with any of the other approaches, c) code can freely move between the APP and TPL code bases in an incremental way, and d) the creation of new capabilities in the TPL and APP can be deployed to the rest of the APP development team quickly and safely.

However, there are also some disadvantages to the Almost Continuous Integration process. First, the co-development of APP Dev and TPL Dev is complicated by having to maintain two separate VC repositories for TPL Dev and TPL Dev- and getting a single TPL working directory to be able to point to both. Another disadvantage of this process is that regular APP developers will have to recompile the TPL code more frequently then they might want. If this is a serious problem, then the Almost Continuous Integration automated nightly updating process can be scaled back from updating the APP-owned TPL Dev- VC repository from every night to instead every few nights or less frequently depending on current circumstances. Even if APP-owned TPL Dev- VC repository was only updated once a week or even once a month, this is still significantly better than the APP + TPL Release and Dev Daily Integration process described in Section 5.2. However, testing of APP Dev + TPL Dev would be performed every night in order to catch problems as soon as possible.

While this APP + TPL Almost Continuous Integration

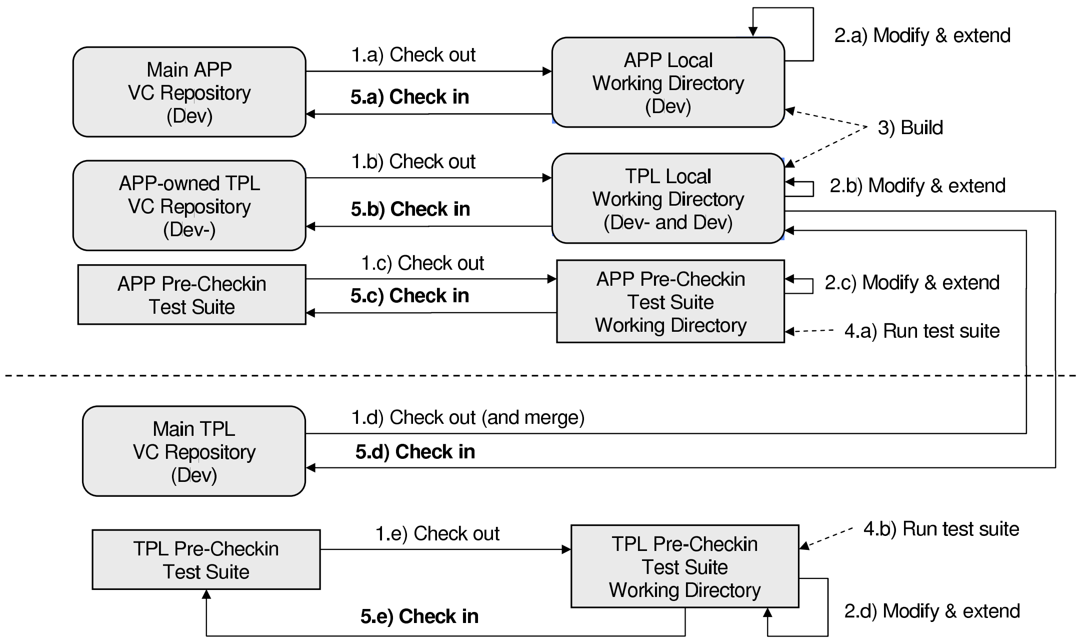
Main APP VC Repository (Dev)
1.a) Check out
5.a) Check in
APP Local Working Directory (Dev)
2.a) Modify & extend
3) Build
APP-owned TPL VC Repository (Dev-)
1.b) Check out
5.b) Check in
TPL Local Working Directory (Dev- and Dev)
2.b) Modify & extend
APP Pre-Checkin Test Suite
1.c) Check out
5.c) Check in
APP Pre-Checkin Test Suite Working Directory
2.c) Modify & extend
4.a) Run test suite
Main TPL VC Repository (Dev)
1.d) Check out (and merge)
5.d) Check in
TPL Pre-Checkin Test Suite
1.e) Check out
TPL Pre-Checkin Test Suite Working Directory
4.b) Run test suite
2.d) Modify & extend
5.e) Check in

**Figure 4. APP + TPL Almost Continuous Integration Co-Development process**

process has yet to be implemented in our CS&E environment at SNL, we have more than two years worth of experience with APP + TPL Release and Dev Daily Integration that indicates it will work very well.

## 6. CS&E software integration experience at Sandia National Labs

While the majority of this paper has been written in general terms, the ideas and proposed processes discussed are founded on significant practical experience with CS&E software integration at Sandia National Labs (SNL) and other related institutions.

The majority of these experiences are related to the Trilinos TPL code [14]. Trilinos is an ever expanding collection of separate packages that implement a variety of advanced research-driven CS&E algorithms. Trilinos has grown from just three packages and a single developer in 1998 to over 45 packages and more than 30 part-time developers at the end of 2008. Trilinos has a number of important customer APPs inside and outside of SNL. Ten or more distinct APP projects within SNL use some of the Trilinos packages. Here, integration experiences with three different internal SNL APP codes that use Trilinos are discussed; Alegra, Charon, and SIERRA. The Alegra APP code implements a number of important PDE discretization methods including high-energy shock hydrodynamics [19]. The Charon APP code implements semiconductor, reacting flow, and magneto-hydrodynamics (HMD) PDE discretization methods [3]. The SIERRA APP code represents a collection of PDE discretization codes and is larger and more complex than the other SNL APP codes [5]. The reason the Alegra, Charon, and SIERRA APP codes are specifically discussed is that they each have a long history with Trilinos integration, each have experimented with APP + TPL Release and Dev Daily Integration to some extent, and each are good candidates for APP + TPL Almost Continuous Integration.

Charon has the longest history with APP + TPL Release and Dev Daily Integration with Trilinos which started at the beginning of the ASC 2007 Vertical Integration Milestone Project [3]. Charon never went through the full release cycle shown in Figure 2 but is noteworthy because a) it was the pioneer APP where this integration method was first developed giving more than than two years of experience with the method, and b) it is also likely to be the first APP code that adopts the APP + TPL Almost Continuous Integration process [4].

The Alegra APP code is noteworthy because it adopted the APP + TPL Release and Dev Daily Integration process with Trilinos several months before the 9.0 release of Trilinos. The Alegra development team reported that the maintenance of Alegra Dev + Trilinos Dev was fairly easy and the upgrade to Trilinos 9.0 was stress-free and uneventful. They also estimated that the overall cost of maintaining Alegra Dev + Trilinos Dev and performing the later upgrade to Trilinos 9.0 was less than for any of the punctuated upgrades for new Trilinos releases done in the past.

The SIERRA APP code is significant to this discussion for several reasons. First, SIERRA is the largest (in lines of code and developer effort) and most complex (in every way) than all other Trilinos customer codes. SIERRA also started the APP + TPL Release and Dev Daily Integration process with Trilinos Dev between the Trilinos 8.0 and 9.0 releases. The punctuated upgrade of SIERRA for Trilinos 8.0 was the most painful of any prior upgrade and resulted in undue stress for both SIERRA and Trilinos developers. One of the SIERRA APP codes would not even build for several days, and their most important solver was broken for even longer. In contrast, the SIERRA upgrade to Trilinos 9.0 went smoother than any prior Trilinos upgrade which has helped pave the way for closer collaborations between the SIERRA and Trilinos development communities. Some of the specific goals for SIERRA and Trilinos collaboration require a closer level of software integration than is currently workable with the APP + TPL Release and Dev Daily Integration process. Therefore, SIERRA is another strong candidate for APP + TPL Almost Continuous Integration.

## 7. Conclusions

In order to make significant progress in solving challenging problems in Computational Science & Engineering (CS&E), we need to integrate a large amount of software written by different groups of experts. The scope and complexity of this software is more than can be addressed under any single blanket of continuous integration (CI). The CS&E environment presents a number of unique software integration challenges and the current generation of Agile software integration approaches needs to be adapted for CS&E. The Customer/Supplier relationship [10, Chapter 14] which forms the basis for decoupling Agile software into different CI software collections is significantly less effective in CS&E environments.

This paper describes two different improved adaptations of Agile integration approaches tailored to CS&E. The APP + TPL Release and Dev Daily Integration process has been shown to be very successful in reducing the costs and risks associated with upgrading TPL releases. However, it does not adequately address more challenging collaborative use cases. The APP + TPL Almost Continuous Integration process is proposed as a way to address even the closest APP and TPL collaborations while still maintaining the essential partitioning between the APP and TPL developers and code bases.

Modern Lean/Agile software engineering methodologies must be adopted by the CS&E community if significant progress is to continue. However, these methods must be carefully adapted in order to address the special challenges in CS&E. The issues of software quality, design, testing, and integration are among the most important and significant changes in the CS&E development community will be needed to effectively address them.

## References

[1] D. Arnold, P. Bochev, R. Lehoucq, and R. Nicolaides. *Compatible Spatial Discretizations*. Mathematics and its Applications. IMA, 2005.

[2] R. Bartlett. Daily integration and testing of the development versions of applications and Trilinos. Technical Report SAND2007-7040, Sandia National Laboratories, 2007.

[3] R. Bartlett and et. al. ASC vertical integration milestone. Technical Report SAND2007-5839, Sandia National Laboratories, 2007.

[4] R. Bartlett and et. al. Maintenence process for appliction and third party library integration methods. Technical Report In prepairation, Sandia National Laboratories, 2007.

[5] K. Bathe, editor. *Computational Fluid and Solid Mechanics*. Elsevier, 2001.

[6] K. Beck. *Test Driven Development*. Addison Wesley, 2003.

[7] K. Beck. *Extreme Programming (Second Edition)*. Addison Wesley, 2005.

[8] S. Benzley and et. al. Conformal refinement and coarsening of unstructured hexahedral meshes. December 2005.

[9] P. Duvall and et. al. *Continuous Integration*. Addison Wesley, 2007.

[10] E. Evans. *Domain-Driven Design*. Addison Wesley, 2004.

[11] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.

[12] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, March 1991.

[13] M. Heroux. Design issues for numerical libraries on scalable multicore architectures. *J. Phys.*, 2008.

[14] M. Heroux and et. al. An overview of the Trilinos project. *ACM TOMS*, 2005.

[15] R. Martin. *Agile Software Development (Principles, Patterns, and Practices)*. Prentice Hall, 2003.

[16] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, 1999.

[17] D. Poole. Multi-stage continuous integration. *Dr. Dobb's Journal*, December 2008.

[18] M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development*. Addison Wesley, 2007.

[19] A. Robinson and C. Garasi. *Computer Physics Communications*, 164:408–413, 2004.

[20] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.

[21] J. Stewart and H. Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elem. Anal. Des.*, 40(12):1599–1617, 2004.

[22] H. Troger and A. Steindl. *Nonlinear stability and bifurcation theory: an introduction for engineers and applied scientists*. Springer, 1991.