Collaborative Software Development in Support of Fast Adaptive AeroSpace Tools (FAAST)

William L. Kleb*, Eric J. Nielsen†, Peter A. Gnoffo‡, Michael A. Park†, and William A. Wood*
NASA Langley Research Center, Hampton, Virginia 23681

Corresponding author: Bil.Kleb@NASA.Gov
phone: +1 757 864 4364, fax: +1 757 864 8670

A collaborative software development approach is described. The software product is an adaptation of proven computational capabilities combined with new capabilities to form the Agency's next generation aerothermodynamic and aerodynamic analysis and design tools. To efficiently produce a cohesive, robust, and extensible software suite, the approach uses agile software development techniques; specifically, project retrospectives, the Scrum status meeting format, and a subset of Extreme Programming's coding practices are employed. Examples are provided which demonstrate the substantial benefits derived from employing these practices. Also included is a discussion of issues encountered when porting legacy Fortran 77 code to Fortran 95 and a Fortran 95 coding standard.

Introduction

The objective of the Fast Adaptive AeroSpace Tools (FAAST) program at NASA Langley Research Center is to develop the next generation of aerospace analysis and design tools.[1] The four primary elements in this effort are CAD-to-Grid Methods, High Energy Flow Solver Synthesis (HEFSS), Optimally Convergent Algorithms, and Efficient Adjoint Design Methods. This paper primarily focuses on the software development practices adopted by the HEFSS and design elements of FAAST.

Over the past two decades, Langley's Aerothermodynamics Branch has provided extensive computational support for NASA's space program. Contributions have included aerodynamic and aerothermodynamic predictions across the hypersonic regime for planetary missions such as Mars Pathfinder and access to space projects such as X-33, X-34, and X-37. The primary tool used to provide these data has been the LAURA solver,[2] developed by Gnoffo. In addition to external hypersonic flows, Langley's Hypersonic Airbreathing Propulsion Branch has made significant contributions to NASA's hypersonic propulsion projects with the VULCAN solver,[3] developed by White. Together, these two packages represent the state of the art at Langley in high-energy, reacting-gas chemistry computational tools.

While success has been achieved using the structured grid tools LAURA and VULCAN, there are inherent topology limitations on geometric configurations for which structured-grid discrete domains can be efficiently produced. Alternatively, the field of unstructured-grid methods has matured rapidly in recent years. With this approach, a wide range of geometric configurations can be efficiently modeled for analysis. Langley tools such as FUN2D/3D[4,5] and USM3D[6] have been validated with experimental data and with their structured-grid counterparts for a wide array of perfect-gas Reynolds-averaged Navier Stokes applications, ranging from incompressible to transonic and supersonic flows.

The goal of the HEFSS project is to combine the capabilities and strengths of the reacting-gas physical models in LAURA and VULCAN and the unstructured-grid discretizations of codes like FUN3D and USM3D to produce the next-generation computational tool for analysis and design, while employing software development techniques that en-

*Aerospace Engineer, Aerothermodynamics Branch, Aerodynamics, Aerothermodynamics, & Acoustics Competency, member AIAA.

†Aerospace Engineer, Computational Methods Branch, Aerodynamics, Aerothermodynamics, & Acoustics Competency, member AIAA.

‡Aerospace Engineer, Aerothermodynamics Branch, Aerodynamics, Aerothermodynamics, & Acoustics Competency, AIAA fellow.

able a robust, extensible, and portable final product. This software development effort is similar to ONERA's elsA project.[7]

The development of computational fluid dynamic (CFD) application codes[a] at Langley and at the former, co-located Institute for Computer Applications in Science and Engineering (ICASE) has consisted of one or two people working sharply focused applications or algorithms. Even if more people contribute to the development of a code, there is typically only one person who contributes the bulk of the code and serves as gatekeeper for any changes. Manifestations of this paradigm are shown in Table 1. In all these examples, the code architect will cite others who have made important contributions to the code; nevertheless, they are typically managed like a cottage industry.

In cases where the high-level application is identical, algorithm details may differ because of code robustness considerations for specialized applications or because new algorithms must eventually evolve to "application" status. Such evolution has often been easier to accommodate by extending the initial, simple test versions of the algorithm rather than by integrating them into an existing application code. However, over the last decade, both the field of CFD and computational capability have largely outstripped the ability of a single developer to make a significant contribution. The scope and complexity of a modern application now require several experts to work collaboratively.

Software engineering processes which accommodate teams of tens of hundreds of programmers working with a relatively well-defined set of requirements (e.g., a satellite tracking system) were considered,[b] but it was found that they are simply not appropriate for a small, research environment. On the contrary, the emerging agile software development movement[c] is well suited to the uncertain requirements and small teams typically present in a research environment. The agile movement views software development as an empirical process rather than the defined process which software engineering attempts to govern.[8] To manage the empirical process, agile methods incorporate rapid feedback mechanisms to enable constant steering and place a renewed emphasis on the heart of software development—software craftsmanship.[9]

[a] An "application" code is defined as one which can compute aerodynamics and/or aerothermodynamics of three-dimensional flows, including appropriate physical models and boundary conditions for geometrically complex configurations.

[b] See www.sei.cmu.edu/cmm/ for example.

[c] See www.agilealliance.org.

Regardless of the software development process chosen, making the switch from a one-code, one-developer paradigm to a team-based approach is a large culture change under any circumstances. However, the ambitious goals of the HEFSS project provided a strong motivation to look past skepticism and overcome resistance to change since it required a group of developers (initially 12 people ranging between 25 to 100 percent work-level), with diverse areas of expertise, to collaborate on a single piece of software. Within the 18 months of the project, HEFSS had promised to demonstrate successful synthesis of the structured-grid physical models on a cylinder case using an unstructured discretization. In addition, an existing unstructured-grid code was to be selected to serve as the baseline for the HEFSS effort, and its functionality was to be maintained within the HEFSS code base. To compound matters, there were no software development experts available to serve on the team. This critical gap was filled by consultant-led workshops, a visiting lecturer series, a support contractor, and the good fortune of having two team members with CFD expertise aggressively pursue software development best practices appropriate for our team.[10,11]

The purpose of this paper is to document how the HEFSS team adapted and incorporated agile software development practices to develop the next generation CFD application software. No claims are made that the correct process decisions were made or that the current processes have fully matured. And since there is no control team with identical talents and objectives, it is difficult to objectively gauge the performance of the HEFSS team other than to observe that the project is ongoing, morale is high, its practices have been adopted by other teams, it was included in a group achievement award, and the local software engineering process group is using it as a model. The experience and lessons learned are humbly offered as a case study, which may be useful to others with similar background and goals.

The first several sections outline the baseline code selection process, justify the programming language chosen, and outline how the legacy code base was ported and restructured to take advantage of the new language features. Following this background material, modularity and data encapsulation design issues are presented as well as details about how the high-energy physics modules were incorporated into the baseline solver. Next, the software development section documents the practices that allow a team software development environment to thrive. This discussion is followed by a section highlighting several experience reports of research products created

Table 1 CFD code, architect, and application domain.

CFL3D	Rumsey/Biedron	Structured-grid (SG) aerodynamics
LAURA	Gnoffo	SG hypersonic aerothermodynamics
VULCAN	White	SG hypersonic propulsion
TLNS3D	Vatsa	SG aerodynamics
OVERFLOW	Buning	Overset SG aerodynamics
USM3D	Frink	Unstructured-grid (UG) aerodynamics
NSU3D	Mavriplis	UG aerodynamics
FUN3D	Anderson	UG aerodynamics and design
FELISA	Peraire	UG hypersonic aerodynamics

in this environment and finally, some concluding remarks.

Baseline Code Selection

Three Langley unstructured codes, USM3D, FUN3D, and FELISA,[12] were considered as the initial template for the HEFSS code. FELISA, an inviscid, unstructured flow solver, already has considerable success in the hypersonic domain. It also has equilibrium and thermochemical nonequilibrium gas models. While the addition of thermochemical nonequilibrium source terms, thermodynamic models, and transport models was perceived to be straightforward, considerable effort would have been required to introduce the viscous terms, the viscous flux Jacobians, and an implicit solution scheme. Both USM3D and FUN3D are highly successful codes for computing viscous flow on unstructured grids within the subsonic to low supersonic speed regimes. Ultimately, FUN3D was selected because it is more robust in the hypersonic domain, which is apparently attributable to its combination of Roe Flux Difference Splitting, flux reconstruction, and associated limiters. In addition, its discretizations are similar to LAURA, and the discrete adjoint capability for perfect gas design[13,14,15] and grid adaptation[16,17,18,19,20,21] was judged particularly appealing for future hypersonic design and grid adaptation. A successful retrofitting of FUN2D with thermochemical nonequilibrium models confirmed the viability of this approach.

Programming Language

Most of the CFD codes developed at Langley are written in FORTRAN 77 and often rely on nonportable extensions such as vendor-specific functions or links with C code. For the current project, the team sought a single, unifying standard language under which to develop new code. After surveying the available programming languages and deciding that a mixed-language code base would increase complexity too much, FORTRAN 95 was selected for the new suite of codes. FORTRAN 95 promises the numerical performance of FORTRAN 77 with the advanced features of other languages, such as dynamic memory allocation, derived types, recursion, and modules. This choice also allows a relatively straightforward conversion of a substantial legacy code base written in FORTRAN 77.

The selection of FORTRAN 95 was tempered by the commitment to deliver a hypersonic flow simulation with thermochemical nonequilibrium on a geometrically simple configuration within 18 months. Adoption of a programming language significantly different from FORTRAN would have required a learning period for the majority of the team members, who were already proficient with FORTRAN 77. The time required to bring team members up to speed in a new language, plus the time required for conversion of legacy FORTRAN 77 to a language outside the FORTRAN family, was judged too costly, relative to the potential benefit offered by any other language.

FORTRAN 95 training was tailored to team needs in a two-part workshop. Dan Nagle, from Purple Sage Computing Solutions,[d] spent a day with the team learning the HEFSS code objectives and the architecture of the legacy code. Using this material, he prepared a two-day course which highlighted FORTRAN 95 features suited to the HEFSS project.

Auxiliary scripting for controlling code compilation, templating, and testing is performed with Ruby[22,23] and Make.[e] Ruby is an open source, object-oriented, threaded-scripting language with cross-platform support, while Make is an open source compilation tool.

Porting and Restructuring Legacy Code

To lay a solid foundation for the new suite of solvers, FUN3D and the physical models from LAURA and VULCAN were ported from a mixture of C and FORTRAN 77 to FORTRAN 95. Porting FORTRAN 77 code to FORTRAN 95 was initially thought

[d] users.erols.com/dnagle/
[e] www.gnu.org/software/make/make.html

to be a simple process that could be accommodated by using a combination of homegrown scripts and a commercial software package, FORESYS™.[f] FORESYS™ was helpful when `implicit none` was requested because it would automatically declare all variables used in the routine. It also provided instructive diagnostics for various classes of errors during the conversion process and when replacing common blocks by modules. However, it invariably reformatted lines and destroyed symmetric forms of equations that had been carefully introduced by earlier authors, and it repositioned or silently eliminated comments. Eventually, Ruby and Perl scripts were crafted to handle tedious, error-prone operations such as code indentation and the conversion of continuation symbols without losing the comments and other structured formatting. The remainder of the conversion was done manually.

As the team had a chance to study the legacy structure, it became clear that the old arrangements of common blocks and subroutines were counter to the modularity and extensibility the team was trying to create. So, during the port to FORTRAN 95, common routines and functions were extracted and placed in a single, shared library directory, while data structures such as boundary conditions, grid metrics, and solution quantities were generalized to handle an arbitrary number of equations and were encapsulated in derived types.

The use of derived types provides additional flexibility over FORTRAN 77; however, early versions of FORTRAN 95 compilers often displayed a significant performance penalty when these constructs were used in the computationally intensive regions of the solver.[g] Consequently, the restructuring effort often required reworking these core routines to recover performance comparable to the legacy solver.

This transformation took nearly a year and was not without difficulties, but it was definitely a worthwhile effort because it gave team members hands-on experience with a code most had never seen before, instead of merely accepting the results of an automatic conversion. The conversion process also gave the team an opportunity to create and tailor a coding standard[h] suited to their style and knowledge. In addition, the total lines of source code had been reduced by some 40 percent, in itself a significant benefit from the standpoint of code maintenance.

[f] FORESYS™ is a trademark of Connexite S.A., for more information see www.simulog.fr/is/2fore1.htm.

[g] See Appendix B on page 17 for current results.

[h] See Appendix A on page 16.

Modularity and Encapsulation

Modularization, along with abstraction, information hiding, and encapsulation, are also means used to enhance code maintainability and bring the additional promises of code reuse, reduced complexity, extensibility, and orthogonality.[i] Abstraction is the process of picking out common features of objects or procedures and replacing them with a single, more general function. Information hiding reduces complexity by hiding details of an object or function so the developer can focus on the object without worry about the hidden details. Encapsulation, or combining elements to create a larger entity, is one mechanism to achieve this.

The FORTRAN 95 constructs of modules, interface statements, public and private declarations, and derived types were employed to implement these ideas. FORTRAN 95 modules are similar to the class construct in object-oriented languages, while derived types are akin to structures. Modules were designed to abstract types of operations, e.g., file input/output, memory allocation, interprocessor communication, execution timing, linear algebra, and so on. Many modules employ a generic interface statement that automatically detects the type, kind, and rank of the calling arguments at compile time and matches them to an appropriate low-level routine, which allows them to be largely independent of any particular flow solver since data is only exchanged through well-defined interfaces. Many of these FORTRAN 95 interface statements are produced automatically in the build process by a Ruby script which emulates the template system available in C++. In the remainder of this section, specific examples are given to demonstrate the benefits of modularization and data encapsulation.

Memory allocation

Array memory allocation is handled by a single interface statement in a module that automatically detects the type, kind, and rank of the argument and calls the appropriate low-level routine for the allocation and initialization. This abstraction streamlines memory allocation requests throughout the code since memory tracking and diagnostics can be placed and maintained in a single location.

Parallel Communication

Originally, the baseline solver relied on a shared-memory implementation specific to SGI® hardware and was not portable to the increasingly popular cluster-based, distributed-memory computing plat-

[i] In this case orthogonal is used in the sense of mutually independent or well separated.

forms. Moreover, the communication operations were dispersed throughout the solver, and any modifications to the communication model needed to be made in numerous locations throughout the code. In the current work, the message passing interface (MPI) standard was selected. Interprocessor communication has been abstracted from all but the lowest levels of the source code and is now encapsulated in a single module.

With this centralized approach to MPI communication, it is now trivial to make sweeping changes to the parallel aspects of the code, including completely removing it to produce a sequential version of the code. This abstraction also benefited the team when the high-energy, reacting-gas portion of the code was parallelized successfully on the first attempt. Normally, a developer would expect to spend considerable time debugging interprocessor communication.

Boundary Conditions

Another area in which modularity and data encapsulation have provided a significant benefit is in the treatment of boundary conditions. The baseline FUN3D solver was extremely deficient in its ability to handle a wide range of boundary conditions. The user was restricted to inviscid, inflow/outflow, and viscous boundary types. Information required for these boundary types was contained in hard-coded data structures specific to each condition and were dispersed throughout the code. This design had become extremely limiting in recent applications and was clearly not sufficient for extension to high-energy flows, where a large array of boundary condition types are required.

Using FORTRAN 95 derived types to encapsulate boundary condition information, the baseline solver was completely refactored to allow the straightforward addition of new boundary types. For any given boundary condition, all necessary data are contained in a boundary condition type. An array of these derived types then constitutes all boundaries in a given problem. For boundary conditions requiring additional physical data, a link to an additional data structure specific to that boundary condition is encapsulated. Derived types also allow the additional enrichment of the data structure without modifying argument lists. In this manner, any number of different boundary groups can be efficiently handled at the higher levels of the solver and unrolled for use as needed.

It should be noted that this data structure also allows for a natural handling of cost functions based on boundary data required for the design and grid adaptation capabilities within FAAST. Objective functions composed of viscous and/or pressure contributions can easily be specified on any subset or combination of boundary groups such that a specific flow feature or region of the domain can be targeted. For example, if it is determined that a strong shock on the outboard section of a wing is responsible for a severe wave drag penalty, a cost function can easily be formulated based solely on the contribution of that boundary group to the total drag. This method represents a substantial improvement over the baseline capabilities, where all boundary groups necessarily contributed to a given cost function.

Gas Physics

Modules, interfaces, and derived types are used extensively for the gas phase physics modules, which include thermodynamics, transport properties, thermal relaxation, and chemical kinetics. The thermodynamics module contains the initial interface from the flow solver to gas phase physics. The transport property module interfaces with the flow solver and the thermodynamics module to define molecular viscosity, conductivity, and species diffusivities. The thermal relaxation module is engaged when populations of excited states (rotational, vibrational, and electronic modes) cannot be defined by a single temperature. This module provides the source terms that define energy exchange among the available, thermally distinct modes. The chemical kinetics module provides source terms for the species continuity equations that define the rate of production or destruction of species.

In conclusion, it should also be noted that because the HEFSS project started with a large legacy code base and modularity and data encapsulation are elusive goals, which are really only earned through experience, code architecture changes are ongoing. In addition, there are drawbacks to modularization that must be considered. For example, it was originally anticipated that compilers could optimize high-level constructs like derived types as if they were written using their lower-level counterparts. However, as Appendix B on page 17 reveals, such is not always the case in practice.

Collaborative Software Development

CFD software development at Langley has traditionally been performed in a rather unconstrained, self-governed environment. As mentioned earlier, most codes have typically been developed by one, or perhaps two researchers. This paradigm has worked relatively well and has produced software packages widely used by industry and academia.

Unfortunately, such software development strategies often result in codes that are complex and burdensome to maintain, and frequently subsequent working groups produce distinct versions of the code which are often incompatible with each other and previously released versions. Moreover, cohesiveness and portability are typically lost, as additional researchers contribute to the code, using their own coding style and practices.

In contrast to this ad hoc approach to code development, the HEFSS team sought to incorporate the software industry's best practices, not only because of the challenges of working as a cohesive team, but also to find methods which would extend the life cycle of the new code. Everyone on the team had experienced the pain of adding new capability to a large, existing code which was developed in an ad hoc manner. Even a seemingly innocuous bug fix was unnerving because there was no repeatable method to discover whether the fix would break existing capability in some subtle manner.

A survey of industry best practices for software development was conducted, which included sponsoring a local ICASE lecture series entitled "Modern Programming Practices."[j] Meanwhile, two pathfinder projects were conducted to gain hands-on experience. Detailed discussion and extensive reference lists are available in References 10 and 11.

As described earlier, the emerging body of agile software development methodologies were determined to have the best fit with the inconstant nature of a scientific research environment. Specifically, Extreme Programming (XP)[24] appeared to be the most mature, although at the time, documentation was limited to a few websites.[k] In addition, recent experience with ISO 9001 edicts tended to steer the team away from defined process management techniques implicit in methodologies like the Capability Maturity Model®[25] and its associated Team Software Process℠.[26]

The collection of collaborative software development practices described herein evolved from weekly meetings in which the challenges and possible solutions were discussed. Issues discussed cover fresh-start versus retro-fit versus restructuring of existing code; language selection; coding standards; modularization and maintainability versus efficiency; acceptance testing; source code management etiquette;[l] and documentation. As the HEFSS team initially struggled with and then embraced new software development practices, other teams (CAD-to-Grid, Design Optimization) within the FAAST project adopted many of the same practices.

Specific software development techniques are discussed in the following sections, namely: XP, project retrospectives, status meetings, other communication mechanisms, and documentation.

Extreme Programming

XP is founded on four values: communication, simplicity, feedback, and courage. It was designed to keep the right communications flowing by employing many practices that cannot be done without communicating. XP also gambles that it is better to do a simple thing today and pay a little more tomorrow for any necessary changes than to do a more complicated thing today that may never be used; that is, in this universe one cannot "save time." Meanwhile, XP's feedback mechanisms cover many time scales since optimism is an occupational hazard of programming and feedback is the treatment. Finally, courage enables one to escape local optima.

Built from this value system, XP consists of 12 practices shown in Table 2 on the next page. Also shown in the table is the level to that the HEFSS team has adopted each practice. The ensuing sections serve to briefly describe each practice and also to describe a practice in the context of the HEFSS team. Adjacent to the start of each section are quotations from Reference 24.

Sustainable Pace

Productivity does not increase with hours worked; tired programmers are less productive than well-rested ones.

Formally known as "40-hour week," the sustainable pace practice probably ranks the highest on the common sense scale, but it is also the most frequently violated by managers and developers alike. Since the majority of the research conducted with the HEFSS project is years from commercial use, compulsory overtime is simply not part of the working environment.

Metaphor

Guide all development with a simple shared story of how the whole system works.

Employing a system metaphor which all participants can understand facilitates communication both within the code and within the team. Since all the team members are familiar with CFD jargon, the naive metaphor is used.

Coding Standards

Programmers write all code in accordance with rules emphasizing communication through the code.

Coding standards are usually dreaded and met with resistance because they are seen as adding a superfluous burden. After a brief discussion of the genesis of HEFSS's coding standard, several reasons are provided to demonstrate why a coding standard is not only necessary but actually quite beneficial for

[j] See www.icase.edu/series/MPP/

[k] www.c2.com and www.extremeprogramming.org.

[l] Source code management etiquette—when source code should and may be committed to a common repository

Table 2 Current level of XP adoption.

Practice	Adoption	Comments
Sustainable pace	Full	No compulsory overtime.
Metaphor	Full	Using naive metaphor, i.e., CFD jargon.
Coding standards	Full	See Appendix A on page 16.
Collective ownership	Full	Anyone can change any piece of code.
Continuous integration	Full	Automated build and test on three computer architectures.
Small releases	Partial	A portion of code base is currently export restricted; seeking to relieve this constraint.
Test-driven development	Partial	FORTRAN 90 unit test framework not widely used; however, Ruby codes are typically created using TDD.
Refactoring	Partial	Performed, but not mercilessly, due to lack of unit test coverage.
Simple design	Partial	Upfront, complex design is hard to resist, especially without strong test-driven development and refactoring.
Pair programming	Partial	Practiced, but not exclusively.
On-site customer	Partial	No outside customer is providing a business perspective, currently self serving as customer for research products at hand.
Planning game	None	Have yet to invoke project management side of XP.

a team software development project.

During the transition of legacy code from FORTRAN 77 to FORTRAN 95, a rough guess at a coding standard was created and used by the entire team. Based on this experience, a more detailed revision was created. (See Appendix A on page 16.) One duty of the full-time contractor assigned to the team is to enforce the coding standard as new content is committed to the repository. This function is slated to be replaced by an automated agent that parses the source code.

Given a thoughtfully crafted coding standard, improved source code readability is a natural benefit through consistent indentation, alignment, naming, and commenting conventions. However, the coding standard must be appropriately tailored to the programming language. For example, FORTRAN 95 permits declaring an array variable and later dimensioning it through a separate statement. This multi-line variable declaration can be hard to follow and can create confusion, thus prompting a line in the coding standard to place all attributes of the declaration on a single line, if possible. Another example is that the variable names of arguments in the calling and called routines do not have to match. However, retaining the same names for both improves global comprehension of the code and makes code-generated documentation more coherent.

A coding standard also serves as a sentinel against the use of vendor-specific language extensions or depreciated elements of the language that do not lend themselves to portability across various platforms. For example, FORTRAN 95 does not contain a complete set of intrinsic functions for accessing system-level utilities or timing, but many compiler vendors offer extensions like `system()` and `etime()`, which are tempting but create portability headaches.

Collective Ownership

Anyone can change any code anywhere in the system at any time.

The ideal situation for team software development occurs when a pair of developers looks at a given piece of code and does not feel the need to change the indentation, and so forth, and furthermore cannot recall whether they wrote the code in the first place. No single developer claims code ownership, yet all share responsibility; all source code is eligible for changes by any team member. Using a coding standard is absolutely essential to reach this goal.

Collective code ownership was a completely foreign concept to team members prior to this project. Initial acceptance of this philosophy came about because the original developer of the FUN2D/3D code was no longer at Langley, and the current "code steward" did not feel comfortable claiming the code as "his." Both the software development practices mentioned above and the tools the team uses for effective collaboration have cemented the idea of collective code ownership to the extent that members feel comfortable changing the code without asking permission of another developer.

Due to the team-oriented nature of the project and the amount of source code involved, a widely used, source code management system is used, the Concurrent Versions System (CVS).[m] CVS oversees a central repository of the source code and allows each team member to concurrently develop and modify sections as needed. Any changes or additions to a local working copy can then be committed back to the repository, whereby they will be available to the

[m] www.cvshome.org

entire team.

CVS maintains complete documentation of any changes made during the course of code development, and previously modified or deleted code can be resurrected at any time by any member of the team. In addition, the system allows team members to work on platforms located virtually anywhere. The use of a software management tool allows for nearly seamless integration of a number of widely varying research projects and eliminates the need for multiple branches of a code.[n]

Integrate and build the system many times a day, every time a task is completed.

Continuous Integration

In a team environment that has many developers who all contribute to a code base on a daily basis, integrating those changes into a common code base quickly becomes a major undertaking unless new code is integrated and tested as soon as practical, preferably within a few hours.

Continuous integration avoids diverging or fragmented development efforts, in which developers are not communicating with each other about what can be shared or reused. Simply stated, *everyone* needs to work with the latest version of the code base. Making changes to obsolete code causes integration headaches.

Originally, developers manually ran the HEFSS test suite during code modification, but not all developers consistently ran the test suite before checking their code modifications into the repository, so an automated process was sought. At first the Unix-based `cron` utility was used to check out a fresh version of the CVS repository, to compile the suite of codes, and to run regression tests on three different architectures and compilers every night. However, the Extreme Programming community soon reminded the HEFSS team that "[daily builds] are for winning-challenged people who can't integrate every 5 to 15 minutes and run all the tests at every integration," and they went to a true continuous integration mode of operation on dedicated machines.

The continuous integration process restarts the build and test process after each successful set of tests. Test results are automatically logged on a web server, and failures are e-mailed to all developers listing all CVS commits that were performed since the last successful build. With this system, errors are detected within a couple hours, and the integration failure e-mail provides a strong source of peer pressure on developers to run a range of tests before committing changes.[o]

[n]This CVS controlled LaTeX document was jointly composed by the team using such an approach.

[o]See www.martinfowler.com/articles/continuousIntegration.html for more information.

Small Releases

Put a simple system into production quickly, then release new versions on a very short cycle.

Feedback is the core idea behind the small releases practice. Get the software out there and learn from it. Strive to make the transition from pure software development to software maintenance as quickly as possible. Small releases are enabled by other practices like simple design, automated testing, and continuous integration.

The source code management system described previously enables the team to automatically create releases by merely "tagging" snapshots of the repository for which all the tests pass successfully during the continuous integration cycle. So routinely, the team is typically making several releases throughout any given day. This snapshot feature also facilitates the management of releases to outside users by providing accurate technical support tailored specifically to the exact source code snapshot released to a given party. Unfortunately, the HEFSS code currently has some restrictions on its external distribution; however, it is being used in house by several people.[27]

Test-Driven Development

Any program feature without an automated test simply doesn't exist.

Since the time to fix a software defect (aka "bug") scales exponentially with the time lag between introduction and detection,[28] it is extremely advantageous to trap defects as early as possible during development.

Previously known as merely "Testing," this practice has blossomed into a whole field in itself.[29] Test-driven development within XP has two components, one centered around developers and the other centered around customers, or end-users. Developers write unit tests so that their confidence in the code can become part of the code itself, while customers write acceptance tests so that their confidence in the code's capabilities can also become part of the code. These automated tests allow confidence in the code to grow over time, allowing the code to become more capable of accepting change.

Unit tests are intended to verify small quanta of functionality within a code and should be automated and run to completion in fractions of a second. The unit tests serve as a development guide by specifying the desired capability, interfaces, and expected output of a functional unit. Unit tests also serve as mobility enablers during code architecture shifts to ensure a safe path was taken. Mobility allows code to be easily reused and to have functionality extended while safely maintaining current functionality. Note that in most cases, there will be more lines of unit

test code than actual production code.

Acceptance tests check the interactions between code elements that unit tests cannot cover and document the existence of a particular code feature. Preferably, customers write acceptance tests.

Since the HEFSS code contains active research in many different disciplines that coexist in the same framework, work in one field can introduce errors in others through the common framework. These errors can go unnoticed if the code, in part and in whole, is not verified in a repeatable manner. One well-known approach to finding defects and ensuring that the code produces repeatable, verified answers is through automated testing.

For example, an unforeseen interaction with module A is introduced by modifying code in module B. If the problem in module A goes undetected for a month, it may be difficult to link the problem to an interaction with module B or to other code modifications made during that month. If the problem in module A is detected in minutes by an automated testing framework, the interaction of module A and module B can be clearly identified before other code modifications cloud the picture.

The current project began with legacy code that did not contain a single unit test. Because retrofitting an exhaustive set of unit tests to the existing legacy code was deemed too expensive, the original intent was to introduce unit tests as new code was added and old code was refactored. To date, however, unit testing has not been widely adopted by the team despite the creation of a unit testing framework for FORTRAN 95.[p] Currently, unit tests only cover a very small percentage of the code base. However, significant unit testing coverage is being built into Ruby-based wrappers used for testing and grid adaptation. Additionally, some of the low-level FORTRAN library routines are becoming test-infected, for example, character-to-number conversion routines and linear algebra routines.

The acceptance tests for the HEFSS code are a suite of over 240 regression tests performed by a series of Makefiles. These regression tests simply compare the convergence history of residual, force, and moment calculations (or other output appropriate to the code under test) to previously recorded executions to machine precision (not just 2–3 digits). These results are referred to as "golden files." These test fixtures ensure that the current code gives the same discrete answer as the original golden file. Makefiles were initially selected to perform these tests because the tests were seen as a natural extension to code compilation.[q] The compile operations were incorporated into the tests, so the tests are always performed with an executable file produced from the current source files. Test cases can be run on an individual basis or as an entire suite.

The current set of acceptance tests for HEFSS was added incrementally to first cover the legacy functionality of FUN3D and then new functionality, as it was added to the suite. The Makefiles that perform the tests have become complex, hard to maintain, and are being replaced in an incremental fashion with unit-tested Ruby. This unit-tested Ruby framework should be much easier to maintain and allow more flexibility. The Ruby framework can be reused to link a number of the codes together to perform complex functions such as design optimization and grid adaptation, in addition to testing.

Refactoring

Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.

To extend a code's viable lifetime and strive for the simplest design that will work, developers need lots of practice modifying the design, so that when the time comes to change the system, they will not be afraid to try it. Constant refactoring is absolutely essential to keeping the cost-of-change curve from growing exponentially as time increases. Reference 30 teaches developers how to refactor and why.

Automated testing, as discussed earlier, is absolutely essential to refactoring. Without a safety net of tests, subtle shifts in the code's fundamental architecture toward a more agile, clean, and understandable design is extremely difficult and frustrating. Testing allows developers to modify code that they did not write so that the original developer can be sure that modified routines still perform the original purpose *correctly*, if the appropriate unit tests pass. This process leads to an environment in which the tests are paramount and the code can be easily modified to add new functionality, improve speed, or become more readable.

Due in part to the lack of extensive unit testing in the HEFSS code, many refactorings are delayed, creating a backlog of work. Occasionally, the team will tackle some of these tasks, but so far the backlog continues to grow. A renewed effort at promoting the benefits of test-first programming is being made within the team by drawing attention to the inefficiencies inherent in the "Code-n-Fix" style of programming.

[p] To facilitate both the writing and running of unit tests for FORTRAN 95 source code, a testing framework called F95UNIT has been developed using Ruby. F95UNIT has a model similar to the unit-testing frameworks for other languages, e.g., JUnit, PyUnit, Ruby test/unit.

[q] If the code is modified and needs to be recompiled, it should also be tested.

The system should be designed as simply as possible at any given moment; extra complexity is removed as soon as it is discovered.

Simple Design

Simple design is defined by two ideas: One is the YAGNI principal, otherwise known as, "you aren't gonna need it," and the other is a chant, "do the simplest thing that could possibly work." These principals should be internalized and provide instinctive reactions to "gold plating" or other ideas that do not seem to fit the current task. A simple design should not contain ideas that are not used yet but that are expected to be used in the future. However, one should pay attention to the word "expected." If you are somehow assured of the future, and that a given idea will be necessary, design with it in mind, but do not implement it now because you will best know how to add it when the time comes.

As with refactoring, the lack of unit test coverage within HEFSS code makes this practice difficult to follow completely. For many developers, it is also typically contrary to years of prior practice; regardless, the team can now at least recognize complexity and several major strides have been made to reduce existing manifestations.

All production code is written with two programmers at one machine.

Pair Programming

The initial reaction to the idea of two people working on the same task at the same computer at the same time is usually negative. However, this reaction is typically caused by painful experiences associated with "pair debugging" or simply misunderstanding the true nature of pair programming itself. Pair programming is not one person programming while another person watches. It is more akin to an animated conversation, facilitated by a white board, where one participant might grab the marker from the other and make a change while the first is still talking. Pair programming should be highly dynamic, and the participants should be able to switch "driver" and "navigator" roles at any point. Besides making programming more fun, pair programming provides an extensive host of benefits, such as streamlining communication, propagating knowledge, and continuous code reviews. Pair programming also greatly enhances collective code ownership. For a detailed discussion of the art of pair programming, see Reference 31.

Within the HEFSS team, frequent pair programming is highly encouraged but not mandated. It is used for all aspects of code development, for example, debugging, teaching, refactoring, and adding new features. Intimately involving a number of researchers at the lowest levels of code development ensures a relatively high truck number.[r] Traditional CFD codes at Langley are developed by individuals or small teams and most of the resulting code base has a truck number of 1 or perhaps 2, whereas the current collaborative team approach yields a value near 10.

On-Site Customer

Include a real, live user on the team, available full-time to answer questions.

This XP practice is intended to remove the communication barriers present in a typical contracted piece of software where a slew of requirements and specifications are defined upfront and then the "code monkeys" are let loose to grind out the required piece of software. The pitfalls with this sort of contract negotiation are many, the least of which is that the customers seldom know what they want before they see a working prototype. By placing an end user with the team, XP is nearly guaranteed of delivering a relevant, useful piece of software.

As discussed in Reference 11, the scientific research environment often creates a situation in which the developers are their own customers. This scenario requires diligent role playing to keep technical and business needs separated. Currently, the HEFSS team members largely act as their own customers, with only very minor input from project stakeholders.

The Planning Game

Quickly determine the scope of the next release by combining business priorities and technical estimates; as reality overtakes the plan, update the plan.

XP uses a four-dimensional space to plan and measure progress: time, cost, quality, and scope. Scope is typically ignored by many project planning mechanisms, but it plays a central role in XP. The planning game has two levels: iteration planning and release planning. The basic premise of the planning game is that business people determine scope, priority, composition of releases, and dates of releases, while technical people provide estimates, design consequences, the process, and detailed scheduling.

As shown in Table 2 on page 7, the HEFSS team has not yet begun using this practice. However, full-cost accounting practices now being put into place may force this final XP practice to be invoked.

Project Retrospectives

Sometimes referred to as XP's "thirteenth practice," project retrospectives[32] are important components of tailoring a process to a given situation. Every few months, the team takes time to reflect on past events and accomplishments. The goal is not faultfinding, but instead the goal is to learn how to do better in the future. During these sessions, the team begins with a discussion guided by the following three questions: what has gone well? what could

[r]The truck number is the size of the smallest set of people in a project such that, if *all* of them got hit by a truck, the project would be in trouble. See c2.com/cgi/wiki?TruckNumber for further discussion.

be improved? and with what new techniques or tools should the team investigate? Currently these sessions are not as formal or wide-reaching as some of the formats presented in Reference 32.

Scrum Status Meetings

A daily, stand-up meeting is normally associated with XP, but it is not explicitly called out as a practice or given much structure, except that nobody can sit during the meeting, it should be short, and it should happen every day before developers start pair programming. The HEFSS team has adopted a similar, but more structured status meeting format from another agile methodology, Scrum.[8]

A Scrum status meeting is held daily by an appointed "Scrum Master" and lasts no longer than 15 minutes. The meeting has an open attendance policy, but only team members are allowed to talk. The team members, in turn, succinctly report three things: what they *did* since the last meeting, what they will *do* by the next meeting, and what got *in the way* (impediments). Additional discussion during a Scrum is strictly limited to clarification-related questions and to note topics that will be discussed at a later time by interested parties. The Scrum master plays the role of gatekeeper and takes notes. Later, the Scrum master compares performance with past commitments and follows up on situations that appear to be stalled. Most importantly, the Scrum master is responsible for removing impediments.

Scrum status meetings have several benefits from a management perspective. They offer a quick and easy mechanism to collect data for status reports and yield an immediate sense of whether a team is in trouble. By using Scrums to their benefit, management can avoid what Peopleware[33] claims is the ultimate management sin: wasting people's time.

Since the HEFSS team is currently dispersed throughout the local campus and most developers are not full time, the Scrum status meeting is only held weekly. In addition, the team also allots some time afterward to address any topics which may have arisen during the Scrum. This post-Scrum gathering is governed by Open Space's Law of Two Feet,[s] which states that if during the course of any gathering, persons find themselves in a situation in which they are neither learning nor contributing, they must use their two feet and go to some more productive space.

Other Communication Mechanisms

Since communication and cooperation are essential to the success of the effort, several additional tools are employed in addition to the communication mechanisms implicit in XP. The first is a Majordomo-based electronic mailing list which, serves to facilitate communication among team members that are distributed across the local campus. In addition to the E-mail list and weekly meetings, the team also uses a web-based collaborative tool known as a Wiki.[t] A Wiki allows users to freely create and edit web page content using any web browser. Wikis have a simple text syntax for creating web page elements, and they dynamically create a new web page when they encounter a CamelCased word (a mixed-case word containing at least two capitals). The team uses the Wiki for a number of purposes. For example, the testing status page is contained in the Wiki so that adding new data to the page can be done by anyone. The Wiki is also used to share data for emerging test cases that have yet to be incorporated into the automated testing system, and it also serves as a repository for otherwise tacit knowledge, for example, CompilerNotes, AvoidingSshPasswords, CreatingNewTestCases.

Documentation

Documentation for the HEFSS code takes many forms. While currently the HEFSS code itself lacks a formal users' manual,[u] it does have a more exacting form of documentation, a large set of regression test cases. Each test case directory contains everything needed to run a given type of case and can usually be readily adapted to a new type of case.

Meanwhile, developers have three tools available for browsing the HEFSS code base. Code browsing can take many forms and be done for various reasons; consolidating them into a single tool has so far proven to be an elusive goal.

The simplest tool is a web-based rendering of the CVS repository, generated on-the-fly by the open source VIEWCVS[v] tool. This approach is based on the CVS repository's file directory structure and thus lacks the ability to navigate the source by using internal structure. However, it is the only tool that readily provides access to prior versions of the source code.

A second tool, developed by a support service contractor using C++, parses the source code and generates web-based output by using a commercial

[s] See www.openspaceworld.com/users_guide.htm for more discussion.

[t] www.wiki.org

[u] The users' manual is being written.

[v] viewcvs.sourceforge.net

tool, UNDERSTAND FOR FORTRAN,[w] which extracts calling tree graphs and code statistics. The C++ code also creates tables of variable declarations and renders comments associated with routines that are placed according to the coding standard. The web pages generated by this tool include source code listings that have been formatted with line numbers and are keyword-colored to enhance readability.

A third code-browsing opportunity leverages the open source code documentation system, RDoc,[x] which was originally intended for documenting Ruby source. A short extension for this system was written to parse and format FORTRAN 95[y] and has subsequently been accepted into the RDoc distribution. The RDoc system extracts a graph of the code source based on files, modules, and routines. From these data, it can generate frame-based web pages, XML, or Windows help files that can be used to navigate the calling structure.

Research Products

To illustrate some specific examples of lessons learned during the current effort, several research focuses are discussed briefly below. The team's experience has been largely a positive one; however, it is clear that properties of modularity and extensibility are earned through experience and not designed into a system upfront.

Time-Accurate Simulations

In support of both passive and active flow control research at Langley, the perfect-gas capabilities in the solver have been extended to higher order temporal accuracy. The validity of the approach has been verified through numerical experiments in which an order property consistent with a second-order scheme has been demonstrated for turbulent flows. With a trivial amount of effort, the modifications required to obtain these results in the perfect-gas realm were extended to include reacting-gas simulations. Current work is focused on evaluating third- and fourth-order time-integration schemes for perfect-gas flows,[34] which should also be readily extendable to more complicated physical models as needed.

Incorporating Multiple Element Types

Initially, the HEFSS solver made sole use of tetrahedral element types to discretize a given domain. However, the ability to accommodate additional element types such as prisms, hexahedra, and pyramids provides greater flexibility to match a given element type to a particular flow topology, and the extension to include such elements in all aspects of the package is currently ongoing. This effort represents one of, if not the, most substantial modifications to the software to date since it extends the fundamental data structure used throughout the code base. The pre-/post-processor and solvers, as well as all of their associated linearizations for optimization and adaptation, require considerable modification at the most fundamental levels. This undertaking has revealed many areas in which additional refactoring is still required before an acceptable level of modularity is achieved.

Two-Dimensional Capability

A major advantage of pursuing mixed-element discretizations is the ability to recover a truly two-dimensional solution capability, which can be achieved through the use of prismatic and hexahedral elements in the spanwise direction, such that flux balances need only be performed in the plane of symmetry. Axisymmetric flows can also be readily accommodated by adding source terms. The benefits of such an approach are substantial, in that a separate code need not be maintained for such problems, a longtime burden for the original FUN2D/3D developers. In addition, all algorithms and physical models available in the three-dimensional path are immediately available for two-dimensional solutions, which allows basic research to be carried out on less costly two-dimensional problems. When computations are extended to three dimensions, the inconsistencies normally associated with switching between two separate solvers are no longer an issue, and the results are not contaminated by differences in discretizations or solution methods.

Multigrid Algorithms

A major thrust of the FAAST project is aimed at achieving textbook multigrid efficiency (TME), an effort that could drastically reduce solution times for complex problems.[35] Since the baseline unstructured solver used as the foundation for the current work did not include options for multigrid acceleration, much work has focused on implementing such a capability.

The use of an agglomeration multigrid algorithm relies on an edge-based discretization of the governing equations; this requirement precludes the ability to compute solutions to the full Navier-Stokes equations on mixed-element grids. For this reason, a geometric non-nested multigrid approach has been initially chosen for the HEFSS solver. Operations such as coarse-grid partitioning and intergrid

[w] www.scitools.com/uf.html
[x] rdoc.sourceforge.net
[y] This extension was accomplished with only 120 lines of code.

transfers in a complex domain-decomposed environment have been developed, and a simple FMG/FAS multigrid algorithm has been implemented. Although this capability has been coded primarily with perfect-gas applications in mind, the scheme has been implemented such that users performing reacting-gas computations will also be able to make immediate use of this research without the need to duplicate extensive low-level code development typically associated with geometric multigrid on domain-decomposed unstructured meshes.

One component necessary to achieve TME is a line-implicit solver to overcome stiffness associated with high-aspect ratio grid elements. The ability to form lines suitable for implicit relaxation, to obtain an appropriate partitioning, and to perform an exact inversion along each line has been developed and is applicable to any set of physical equations being solved.[15]

Incorporating High-Energy Physics

The thermochemical nonequilibrium models in Hefss are identical to those in Laura, but their implementation is substantially different. Laura made extensive use of precompiler directives that allocated memory and defined the code path according to a diverse set of options. This compilation strategy evolved from an absence of dynamic memory allocation capability in Fortran when Laura was originally coded and because of a desire to completely eliminate any model-dependent conditional statements within loops that could compromise vector efficiency. Any change in the gas model required a recompilation of the source code. Laura employs a script to guide a user through the various permutations and combinations of options, but the process is burdensome to a user conducting parametric studies. In contrast, Hefss only needs to be compiled once on any platform, regardless of the desired physics model options.

Model parameters in Laura are initialized in block data routines; these routines have been replaced by formatted data files that use conventional formatted reads and namelists in the Hefss solver. Model parameters that are unlikely to be changed by the user (thermodynamic curve fit constants, species molecular weights, and heats of formation) are assembled in one set of data files. Gas model options that are likely to be changed by the user on a frequent basis, such as the chemical composition of the gases entering the domain or the thermochemical model, are assembled in a separate file. This separation minimizes the amount of setup required to perform a given analysis.

Adjoint Solver and Sensitivity Analysis

As important as the software practices in this effort are to the development of new analysis capabilities, they are absolutely critical to the success of the design element under FAAST. In References 13, 14, 15, a discrete adjoint capability has been developed for the solver. This effort represents the only capability of its kind and relies on several hundred thousand lines of exact hand-differentiated linearizations of the preprocessor, flow solver, and mesh movement codes with respect to both the dependent variables and the grid coordinates. For free-stream conditions of Mach 0.84, a 3.06 degree angle of attack, and a Reynolds number of 5 M, sensitivity derivatives of the lift and drag coefficients, with respect to several shape design variables for fully turbulent flow over an Onera M6 wing,[36] that were computed by using the discrete adjoint formulation, are shown in Table 3 on the next page. The adjoint results are in excellent agreement with those obtained using a complex-variable approach[37] with a step size of 1×10^{-30}. This accuracy can easily be compromised by a single error anywhere in the source code. With a dozen researchers modifying code on a daily basis, the use of continuous integration and automated testing is critical in maintaining such accuracy. Just as residual and force convergence histories are monitored to machine accuracy for the flow solver on several architectures, similar quantities are constantly tested for the adjoint solver and gradient evaluation codes. This constant testing ensures that discrete consistency between the analysis and design tools is always maintained, regardless of the modifications being implemented in other parts of the software.

Similar to the continuous integration and testing performed for the hand-differentiated code, a Ruby code has been developed similar to the effort described in Reference 38 to automatically convert the codes in the Hefss suite to a complex-variable formulation. This capability can immediately recover a forward mode of differentiation for the entire solver at any time, with no user intervention. This procedure is also continuously tested.

Design Optimization

Approximation and Model Management Optimization (AMMO) techniques[39,40,41] have been recently added to the Hefss software set. AMMO is a methodology aimed at maximizing the use of low-fidelity models in iterative procedures with occasional but systematic recourse to higher-fidelity models for monitoring the progress of the algorithm. In current demonstrations, AMMO has exhibited

Table 3 Comparison of discrete adjoint and complex variable design variable derivatives for coefficients of lift and drag for fully turbulent flow over an Onera M6 wing.

	Camber	Thickness	Twist	Shear	
C_L	0.956208938269467	-0.384940321071468	-0.010625997076936	-0.005505627646872	discrete
	0.956208938269046	-0.384940321071742	-0.010625997076937	-0.005505627647001	complex
C_D	0.027595818243822	0.035539494383655	-0.000939653505699	-0.000389373578383	discrete
	0.027595818243811	0.035539494383619	-0.000939653505699	-0.000389373578412	complex

from three to five-fold savings in terms of high-fidelity simulations on aerodynamic optimization of 3D wings and multi-element airfoils, where simplified physics models (e.g., Euler) computed on coarse grids serve as low-fidelity models, while more accurate models (e.g., Navier-Stokes) computed on finer grids serve as high-fidelity models. AMMO was the first approach for using variable-fidelity models analytically guaranteed to converge to high-fidelity answers.

Because AMMO relies on using a variety of models in a single optimization run, maintaining continuous integration and consistency with the entire software set is especially crucial for obtaining stable optimization results. However, designing a testing strategy for optimization presents an interesting challenge since optimization algorithms requires reasonably well converged analyses and is, therefore, expensive. Procedures for automated testing of optimization software is currently under development.

Output Error Correction and Grid Adaptation

One of the thrusts of the FAAST program is to develop a mathematically rigorous methodology to adapt a grid discretization to directly improve the calculation of an output function. An adjoint-based error correction and adaptation scheme has produced excellent results in 2D.[19] This scheme is being extended to 3D and incorporated into Hefss.[21] This error correction and adaptation scheme requires the calculation of flow and adjoint residuals on embedded grids with interpolated solutions. The modularity of the Hefss reconstruction, flux, and adjoint routines facilitated this calculation.

The interpolation of the solution onto the embedded grid requires the calculation of least-squares gradients. This gradient routine was readily shared between the flow and adjoint codes. The element-based interpolation scheme was developed test-first with the F95unit framework. The code to compute the flow and adjoint residuals consists of only a small driver routine; the remainder of the code is reused from the flow and adjoint solvers. The anisotropic adaptation metric is calculated with code that was also developed test-first using the F95unit framework.

Concluding Remarks

While the FAAST team has learned a great deal from the arena of commercial software development, it is important to remember that Langley's primary goal is to advance the state of the art rather than deliver commercial software products. However, it has been the experience of the team that the synthesis and extension of the Center's computational fluid dynamic capabilities has greatly enhanced the ability to perform research. Development of a unified framework for computational simulation enables researchers to examine a number of widely varying research disciplines and to apply new technology in a more straightforward and encompassing fashion. The capability laid out in this effort currently supports a broad range of research projects and applications, including general unstructured-grid algorithms, high-energy flows, mixed-element computations, error estimation, grid adaptation, design optimization, time-accurate schemes, turbulence modeling, and multigrid algorithms.

As advances are made in each area, they *immediately* become part of the mainstream capability and are readily available to other researchers and users. Some team members have expressed concern about using such a dynamic software tool for work on specific research projects. However, these concerns are mitigated by developing and continuously invoking a suite of automated test cases.

One truly remarkable aspect of this project is that a gaggle of developers which typically shuddered at any mention of the word "process" gelled into a *team* of developers using a fairly rigorous, pervasive software process that they enjoy.

Acknowledgments

The authors would like to recognize the rest of the FAAST team that made contributions to this paper: Natalia Alexandrov, Harold Atkins, Karen Bibb, Robert Biedron, Mark Carpenter, Dana Hammond, William Jones, Elizabeth Lee-Rausch, Tom Roberts, James Thomas, Veer Vatsa, Sally Viken, and Jeffery White.

The authors would also like to thank Charles Miller of the Aerothermodynamics Branch at NASA

Langley Research Center, Hampton, Virginia, for enduring multiple manuscript reviews of this work.

Colophon

This paper was typeset in Donald Knuth's 10pt Computer Modern Font using the *free*, multi-platform LaTeX typesetting system and Kleb's aiaa bundle.[42] The following packages were also used: array, dcolumn, tabularx, multirow, xspace, varioref, fancyvrb, url, and textcomp.

References

[1]Thomas, J. L., Alexandrov, N., Alter, S. J., Atkins, H. L., Bey, K. S., Bibb, K. L., Biedron, R. T., Carpenter, M. H., Cheatwood, F. M., Drummond, P. J., Gnoffo, P. A., Jones, W. T., Kleb, W. L., Lee-Rausch, E. M., Merski, N. R., Mineck, R. E., Nielsen, E. J., Park, M. A., Pirzadeh, S. Z., Roberts, T. W., Samareh, J. A., Swanson, R. C., Vatsa, V. N., Weilmuenster, K. J., White, J. A., Wood, W. A., and Yip, L. P., "Opportunities for Breakthroughs in Large-Scale Computational Simulation and Design," NASA/TM 2002-211747, June 2002.

[2]Cheatwood, F. M. and Gnoffo, P., *User's Manual for the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA)*, NASA TM-4674, 1996.

[3]White, J. A. and Morrison, J. H., "A Pseudo-Temporal Multi-Grid Relaxation Scheme for Solving the Parabolized Navier-Stokes Equations," AIAA Paper 99-3360, June 1999.

[4]Anderson, W. K. and Bonhaus, D. L., "An Implicit Upwind Algorithm for Computing Turbulent Flow on Unstructured Grids," *Computers & Fluids*, Vol. 23, No. 1, Jan. 1994, pp. 1–21.

[5]Anderson, W. K., Rausch, R. D., and Bonhaus, D. L., "Implicit/Multigrid Algorithms for Incompressible Turbulent Flows on Unstructured Grids," *Journal of Computational Physics*, Vol. 128, No. 2, 1996, pp. 391–408.

[6]Frink, N., "Tetrahedral Unstructured Navier-Stokes Method for Turbulent Flow," *AIAA Journal*, Vol. 36, No. 11, Nov. 1998, pp. 1975–1982.

[7]Cambier, L. and Gazaix, M., "elsA: An Efficient Object-Oriented Solution to CFD Complexity," AIAA Paper 2002-0108, Jan. 2002.

[8]Schwaber, K. and Beedle, M., *Agile Software Development with Scrum*, Prentice Hall, Oct. 2001, See also `http://www.controlchaos.com` last accessed 16 June 2003.

[9]McBreen, P., *Software Craftsmanship: The New Imperative*, Addison-Wesley, 2002.

[10]Wood, W. A. and Kleb, W. L., "Extreme Programming in a Research Environment," *Extreme Programming and Agile Methods—XP/Agile Universe 2002*, edited by D. Wells and L. Williams, Vol. 2418 of *Lecture Notes in Computer Science*, Springer-Verlag, Chicago, IL, Aug. 2002, pp. 89–99.

[11]Wood, W. A. and Kleb, W. L., "Exploring XP for Scientific Research," *IEEE Software*, Vol. 20, No. 3, May 2003, pp. 30–36.

[12]Bibb, K. L., Peraire, J., and Riley, C. J., "Hypersonic Flow Computations on Unstructured Meshes," AIAA Paper 97-0625, Jan. 1997.

[13]Nielsen, E. J., *Aerodynamic Design Sensitivities on an Unstructured Mesh Using the Navier-Stokes Equations and a Discrete Adjoint Formulation*, Ph.D. thesis, Virginia Polytechnic Institute and State University, 1998.

[14]Nielsen, E. J. and Anderson, W. K., "Recent Improvements in Aerodynamic Design Optimization on Unstructured Meshes," *AIAA Journal*, Vol. 40, No. 6, June 2002, pp. 1–9, See also AIAA Paper 01-0596.

[15]Nielsen, E. J., Lu, J., Park, M. A., and Darmofal, D. L., "An Exact Dual Adjoint Solution Method for Turbulent Flows on Unstructured Grids," AIAA Paper 2003-0272, Jan. 2003.

[16]Venditti, D. A. and Darmofal, D. L., "Adjoint Error Estimation and Grid Adaptation for Functional Outputs: Application to Quasi-One-Dimensional Flow," *Journal of Computational Physics*, Vol. 164, 2000, pp. 204–227, See also AIAA Paper 99-3292.

[17]Venditti, D. A. and Darmofal, D. L., "Grid Adaptation for Functional Outputs: Application to Two-Dimensional Inviscid Flows," *Journal of Computational Physics*, Vol. 176, 2002, pp. 40–69, See also AIAA Paper 2000-2244.

[18]Venditti, D. A., *Grid Adaptation for Functional Outputs of Compressible Flow Simulations*, Ph.D. thesis, Massachusetts Institute of Technology, 2002.

[19]Venditti, D. A. and Darmofal, D. L., "Anisotropic Grid Adaptation for Functional Outputs: Application to Two-Dimensional Viscous Flows," *Journal of Computational Physics*, Vol. 187, 2003, pp. 22–46.

[20]Park, M. A., "Adjoint-Based, Three-Dimensional Error Prediction and Grid Adaptation," AIAA Paper 2002-3286, June 2002.

[21]Park, M. A., "Three–Dimensional Turbulent RANS Adjoint–Based Error Correction," AIAA Paper 2003-3849, June 2003.

[22]Matsumoto, Y., *Ruby in a Nutshell*, O'Reilly, Sebastopol, CA, 2002.

[23]Thomas, D. and Hunt, A., *Programming Ruby: The Pragmatic Programmer's Guide*, Addison-Wesley, 2001.

[24]Beck, K., *Extreme Programming Explained*, Addison-Wesley, 2000.

[25]Paulk, M. C., Weber, C. V., and Curtis, W., *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, 1995.

[26]Humphrey, W. S. and Lovelace, M., *Introduction to the Team Software Process*, Addison-Wesley, 1999.

[27]Lee-Rausch, E. M., Buning, P. G., Morrison, J. H., Park, M. A., Rivers, S. M., Rumsey, C. L., and Mavriplis, D., "CFD Sensitivity Analysis of a Drag Prediction Workshop Wing/Body Transport Configuration," AIAA Paper 2003-3400, June 2003.

[28]Boehm, B. W., *Software Engineering Economics*, Prentice Hall, 1st ed., Oct. 1981.

[29]Beck, K., *Test Driven Development: By Example*, Addison-Wesley, 2002.

[30]Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[31]Williams, L. and Kessler, R., *Pair Programming Illuminated*, Addison-Wesley, 2003.

[32]Kerth, N., *Project Retrospectives: A Handbook for Team Reviews*, Dorset House, 2002.

[33]DeMarco, T. and Lister, T., *Peopleware: Productive Projects and Teams*, Dorset House, 2nd ed., 1999.

[34]Carpenter, M. H., Viken, S. A., and Nielsen, E. J., "The Efficiency of High Order Temporal Schemes," AIAA Paper 2003-0086, Jan. 2003.

[35]Thomas, J. L., Diskin, B., and Brandt, A., "Textbook Multigrid Efficiency for Fluid Simulations," *Annual Review of Fluid Mechanics*, Vol. 35, 2003, pp. 317–340.

[36]Schmitt, V. and Charpin, F., "Pressure Distributions on the ONERA-M6 Wing at Transonic Mach Numbers," AGARD AR-138, May 1979.

[37]Anderson, W. K., Newman, J. C., Whitfield, D. L., and Nielsen, E. J., "Sensitivity Analysis for the Navier-Stokes Equations on Unstructured Meshes Using Complex Variables," *AIAA Journal*, Vol. 39, No. 1, Jan. 2001, pp. 56–63, See also AIAA Paper 99-3294.

[38]Martins, J. R. R. A., Kroo, I. M., and Alonso, J. J., "An Automated Method for Sensitivity Analysis Using Complex Variables," AIAA Paper 2000-0689, Jan. 2000.

[39]Alexandrov, N. M. and Lewis, R. M., "A Trust Region Framework for Managing Approximation Models in Engineering Optimization," Tech. Rep. AIAA Papers 96-4101 and 96-4102, Sept. 1996.

[40]Alexandrov, N. M., Nielsen, E. J., Lewis, R. M., and Anderson, W. K., "First-order model management with variable-fidelity physics applied to multi-element airfoil optimization," AIAA Paper 2000-4886, Sept. 2000.

[41]Alexandrov, N. M., Lewis, R. M., Gumbert, C. R., Green, L. L., and Newman, P. A., "Approximation and Model Management in Aerodynamic Optimization with Variable-Fidelity Models," *Journal of Aircraft*, Vol. 38, No. 6, November–December 2001, pp. 1093–1101.

[42]Kleb, W. L., "aiaa—A LaTeX Class and BibTeX Style for AIAA Conference Papers and Journal Submission/Simulation," Electronic Documentation, April 1999, Version 2.5.

[43]Alexandrov, N., Atkins, H. L., Bibb, K. L., Biedron, R. T., Gnoffo, P. A., Hammond, D. P., Jones, W. T., Kleb, W. L., Lee-Rausch, E. M., , Nielsen, E. J., Park, M. A., Raman, V. V., Roberts, T. W., Thomas, J. L., Vatsa, V. N., Viken, S. A., White, J. A., and Wood, W. A., "Team Software Development for Aerothermodynamic and Aerodynamic Analysis and Design," NASA/TM 2003-212421.

Appendix A
Coding Standard

Note: parenthetical numbers refer to line numbers in the sample program which follows.

Style

- Free format with no character past column 80
- Indentation: begin in first column and recursively indent all subsequent blocks by two spaces.
- Start all comments within body of code in first column [42].
- Use all lowercase characters; however, mixed-case may be used in comments and strings.
- Align continuation ampersands within code blocks [77].
- No tab characters
- Name `ends` [85].

Comments

- For cryptic variable names, state description using by a comment line immediately preceding declaration or on end of the declaration line [62].
- For subroutines, functions, and modules, insert a contiguous comment block immediately preceding declaration containing a brief overview followed by an optional detailed description [42].

Variable Declarations

- Do not use Fortran intrinsic function names.
- Avoid multi-line variable declarations.
- Declare `intent` on all dummy arguments [63].
- Declare the kind for all reals, including literal constants, using a kind definition module.
- Declare `dimension` attribute for all non-scalars [63].
- Line up attributes within variable declaration blocks.
- Any scalars used to define extent must be declared prior to use [60].
- Declare a variable name only once in a scope, including `use module` statements.

Module Headers

- Declare `implicit none` [35].
- Include a public character parameter containing the CVS `$Id$` tag [37].
- Include a `private` statement and explicitly declare public attributes.

Subroutines and Functions

- The first executable line should be `continue` [69].
- Use the `only` attribute on all `use` statements [58].
- Keep `use` statements local, i.e., not in the module header.
- Group all dummy argument declarations first, followed by local variable declarations.
- All subroutines and functions must be contained within a module.
- Any pointer passed to a subroutine or function must be allocated by at least size 1 to avoid null or undefined pointers.

Control Constructs

- Name control constructs (e.g., `do`, `if`, `case`) which span a significant number of lines or form nested code blocks.
- No numbered do-loops.
- Name loops that contain `cycle` or `exit` statements.
- Use `cycle` or `exit` rather than `goto`.
- Use case statements with case defaults rather than if-constructs wherever possible.
- Use F90-style relational symbols, e.g., `>=` rather than `.ge.` [73].

Miscellaneous

- In the interest of efficient execution, consider avoiding:
  - assumed-shape arrays
  - derived types in low-level computationally intensive numerics
  - `use` modules for large segments of data
- Remove unused variables.
- Do not use common blocks or includes.

Illustrative Example

```
! Define kinds to use for reals in one place

module kind_defs

  implicit none

  character (len=*), parameter :: kind_defs_cvs_id = &
    '$Id: cs_example.f90,v 1.5 2002/08/13 02:37:59 kleb Exp $'

  integer, parameter :: sp=selected_real_kind(P=6)  ! single precision
  integer, parameter :: dp=selected_real_kind(P=15) ! double precision

end module kind_defs

! A token module for demonstration purposes

module some_other_module

  implicit none

  character (len=*), parameter :: some_other_module_cvs_id = &
    '$Id: cs_example.f90,v 1.5 2002/08/13 02:37:59 kleb Exp $'

  integer, parameter :: some_variable = 1

end module some_other_module

! A collection of transformations which includes
! stretches, rotations, and shearing.  This comment
! block will be associated with the module declaration
! immediately following.

module transformations

  implicit none

  character (len=*), parameter :: transformations_module_cvs_id = &
    '$Id: cs_example.f90,v 1.5 2002/08/13 02:37:59 kleb Exp $'

contains

! Computes a stretching transformation.
!
! This stretching is accomplished by moving
! things around and going into a lot of other details
! which would be described here and possibly even
! another "paragraph" following this.
!
! This contingous comment block will be associated with the
! subroutine or function declaraion immediately following.
! It is intended to contain an initial section which gives
! a one or two sentence overview followed by one or more
! "paragraphs" which give a more detailed description.

  subroutine stretch ( points, x, y, z )

    use kind_defs
    use some_other_module, only: some_variable

    integer,                      intent(in)  :: points

!   component to be transformed
    real(dp), dimension(points), intent(in)  :: x, y
    real(dp), dimension(points), intent(out) :: z ! transformation result

    external positive
    integer :: i

    continue

    i = 0

    if ( x(1) > 0.0_dp ) then
      call positive ( points, x, y, z )
    else
      do i = 1, points
          z(i) = x(i)*x(i) + 1.5_dp * ( real(i) + x(i) )**i &
               + ( y(i) * real(i) ) * ( x(i)**i + 2.0_dp )  &
               + 2.5_dp * real(i) + 148.2_dp * some_variable
      enddo
    endif

  end subroutine stretch

end module transformations
```

Appendix B

Fortran 95

The rationale for some elements of the coding standard presented in the previous section are discussed in this section.

Best Practices

The use of `implicit none` minimizes the possibility of variable type errors. An example of a type error is when the implicit FORTRAN integer typing scheme creates integers for variable names beginning with the letters "i" through "n" when the user had intended a real variable. This unintended declaration type is avoided because `implicit none` requires every variable to be declared explicitly.

The use of `only`* prevents unintended changes to values of other variables in the inherited modules. The `only` statement also facilitates finding the module that provides the inherited variable. To further restrict access to variables or subroutines in modules, a `private` statement is to be placed at the top of the module. An exclusive and explicit list of `public` entities is therefore required to share module data and methods outside the module. This exclusivity prevents unintended variable modifications.

Use of equality comparison with reals should be avoided because small, round-off errors may be present. The difference between the two variables is compared to an intrinsic function like `tiny()` to provide a more reliable comparison.

In general, the use of the `select case` conditional construct is more efficient than using an `if-elseif` construct since `if-elseif` might require several condition evaluations, while the `select case` only contains one condition evaluation. The `select case` construct is analogous to the depreciated computed `goto`.† Also `select case` constructs convey control logic in clearer fashion and allow for cleaner error handling through the `default case`.

Performance Considerations of Fortran 95

Throughout the FORTRAN 95 restructuring of the FUN3D solver, several efficiency issues pertaining to advanced coding constructs were uncovered. Features such as derived types and modules are extremely attractive for communicating data; however, it was found that current FORTRAN 95 compilers often failed to produce performance comparable to that of conventional FORTRAN 77 constructs such as passing data through calling argument lists.

Data Sharing With Modules

An intermediate restructuring of FUN3D relied almost exclusively on the use of FORTRAN 95 modules. By eliminating virtually every argument list in the solver, an exceptionally clean code was obtained.

*For example, `use aModule, only : aVariable`

†See groups.google.com/groups?threadm=9o7uhi%24pus%241%40eising.k-net.dk for further discussion.

Table B1 Compilers used in performance study.

Vendor	Options	Release	O/S	Platform
Absoft™	`-O3 -cpu:p6`	8.0-1	Linux® 2.4.18	Intel® P3
Compaq®	`-arch ev67 -fast -O4 -tune ev67`	X1.1.1-1684	Linux® 2.4.2	Alpha EV67
HP®	`-O3`	2.4	HP-UX® B.10.20	HP® 9000
IBM®	`-O5`	7	AIX® 3	IBM® 7044
Intel®	`-O3 -ipo -wK`	7.1-008	Linux® 2.4.18	Intel® P3
Lahey-Fujitsu	`--o2 --nwarn -static --nsav --ntrace --nchk -x -`	6.20a	Linux® 2.4.18	Intel® P3
NAG®	`-O4 -Wc,-malign-double -ieee=full -unsharedf95`	4.2	Linux® 2.4.18	Intel® P3
NA Software	`-fast`	2.2-1	Linux® 2.4.18	Intel® P3
PGI®	`-fast`	4.1-1	Linux® 2.4.18	Intel® P3
SGI®	`-O2`	7.3.1.2m	IRIX® 6.5	SGI® R10000
Sun℠	`-fast`	6.2-2	SunOS™ 5.8	Sun℠ Blade1000

Table B2 Unformatted disk I/O using 20M integers and 20M reals.

Compiler	Assumed size	Module	Derived type	Assumed shape
Absoft™	1.00	1.00	1.03	1.04
Compaq®	1.00	0.98	6.47	0.99
IBM®	1.00	1.03	1.01	1.03
Intel®	1.00	1.16	1.14	1.05
Lahey/Fujitsu	1.00	6.05	5.99	1.04
NAG®	1.00	1.02	1.22	1.03
NA Software	1.00	0.99	1.08	1.01
PGI®	1.00	0.98	1.00	0.98
SGI®	1.00	31.91	31.37	34.80
Sun℠	1.00	0.98	1.02	0.98

Table B3 Compute work using 20M integers and 20M reals.

Compiler	Assumed size	Module	Derived type	Assumed shape
Absoft™	1.00	1.16	1.84	1.20
Compaq®	1.00	1.40	1.47	1.38
IBM®	1.00	2.76	2.76	2.76
Intel®	1.00	0.97	0.98	0.95
Lahey/Fujitsu	1.00	1.07	1.07	1.02
NAG®	Aborted			
NA Software	1.00	0.95	1.13	0.92
PGI®	1.00	1.96	1.96	0.94
SGI®	1.00	1.10	1.10	1.07
Sun℠	1.00	1.42	1.40	1.07

However, in subsequent testing, this implementation was shown to be several times slower in execution speed than the legacy C/Fortran 77 solver. Upon closer inspection, it was found that the use of modules to communicate large segments of data can be extremely inefficient. To illustrate this degradation in performance, the test code included in Reference[43] has been executed on a range of platforms and compilers as listed in Table B1. Here, data is communicated with a file I/O routine, as well as a routine that performs a large amount of arbitrary floating-point manipulations. In addition to an array `A` passed through a traditional argument list interface, an identical array `B` is also passed to and from the subroutines through the use of a Fortran 95 module. For this test, the extent of the arrays is 20 M, a value on the order of that encountered in typical aerodynamic simulations. The results are normalized on the data obtained using the argument list model. As can be seen in Table B2, use of the module construct can incur severe penalties for unformatted disk I/O. The module in-

terface is over thirty times slower than the data transferred via a conventional argument list on an SGI®. For floating-point arithmetic, the module interface exhibits run times on the order of 20 percent higher than the computations using data brought in through an argument list, as shown in Table B3 on the facing page. Due to this performance degradation, the module construct is employed sparingly in the HEFSS solver as a means to share large data structures. Only small amounts of data such as free-stream quantities, algorithmic parameters, and turbulence modeling constants are shared through modules.

Derived Types

The baseline C/FORTRAN 77 solver was also refactored to make extensive use of the FORTRAN 95 derived type construct. The derived type is very attractive in the sense that a number of related quantities can be encapsulated in a single variable, yielding relatively short argument lists throughout the code. Using this paradigm, variables related to the computational grid are stored in a `grid` type; solution-related variables are located in a `soln` type, and so forth. When a low-level routine requires a fundamental piece of data such as the coordinates of a grid point i, the information can be extracted as `grid%x(i)`, `grid%y(i)`, and `grid%z(i)`. Arrays of derived types are also supported under FORTRAN 95, making the implementation of algorithms such as multigrid and multiple instances of quantities, such as boundary groups, straightforward.

As in the case of modules, it was found that the use of derived types can also incur severe execution penalties. As shown in the last column of Tables B2 and B3 on the preceding page, a similar test to the one described previously has been performed on an array `C` transferred as the component of a derived type variable. It can be seen in Tables B2 and B3 on the facing page that this coding idiom can yield execution times more than thirty times slower for unformatted disk I/O and nearly a factor of three slower for floating-point operations over the argument list model.

The current HEFSS solver uses derived types to encapsulate much of its data structures; however, the components of these types required by low-level routines are extracted at the calling level and are received as conventional scalars and arrays in the I/O- and compute-intensive portions of the code. This model allows simple argument lists at the higher levels of the code, while maintaining the performance of the baseline solver. From a developer's point of view, derived types are one of the more useful enhancements of FORTRAN 95 over FORTRAN 77. They allow the developer to string together variables in meaningful groups and treat them as a single entity when desired. The HEFSS code uses a number of derived types. For example, the grid derived type contains all the information needed for the specification of the discretized mesh—x,y,z values for each point in space, cell volumes, cell-face normals and areas, connectivity information, and so on. Any of this information is available with the simple construct `grid%variable`, e.g., `grid%x`. Derived types may also be concatenated, extending their usefulness. For example, the grid derived type in the HEFSS code encompasses a boundary condition derived type that contains all the necessary data to impose boundary conditions—the physical condition (e.g., solid wall), the locations of points on the boundary, surface normals, and so forth. In addition, the definition of the derived type may be extended at a future date without affecting existing code. For example, adding a cell-face velocity for moving grid applications would involve a one-line addition to the type definition and would be completely transparent to sections of code not requiring this information.

Assumed-Shape Arrays

As shown in Tables B2 and B3 on the preceding page, some compilers treat arguments passed via assumed-shape arrays as poorly as they did derived types. Assumed-shape arrays can be noncontiguous, and thus interfacing to old FORTRAN 77 routines may require data to be copied to form a contiguous data block. These data copies can cause a large increase in the total memory required to compute a flow solution for some compilers as compared to others.

Memory Copies

Occasionally it is desirable to bring variables into a routine via argument lists rather than modules, as demonstrated in Tables B2 and B3 on the facing page. However, unexpected behavior was detected on certain platform/compiler combinations when argument lists were combined with low-level module use. In these instances, the variables in the modules were not synchronized with the argument list variables. This synchronization issue was resolved when argument lists were used consistently throughout the subroutines that needed access to the data. It was eventually surmised that this problem was due to memory copies made by some compilers during a subroutine call. When that data copy was modified, it was no longer synchronized with the original data stored in the module and accessed with `use`. Also,

on return from the subroutine, the local copy of the data was used to overwrite the data stored in the module, possibly erasing any modifications of the original data while the copy existed. This behavior appears to be very compiler and application specific and very difficult to detect and instrument.

Compilation Errors, Warnings, and Information

The various compilers listed in Table B3 on page 18 generally have different sets of constructs that deem errors or produce a warning or other information. Some of the compilers are generally more lenient or particular than others when it comes to the constructs that are accepted as valid code for compilation. The FORTRAN 95 code base has benefited from exposure to a large number of different compilers. The coding standard contains guidelines for promoting portability. This portability experience was gained by exposure to multiple compilers, which makes it important to build and test on many different architectures/compilers, and which also results in a code base that is very portable.

Compiler Maturity

In addition to the problems discussed with performance, errors have been found in a number of compilers. Some versions of the compilers have contained errors that have prevented them from successfully compiling HEFSS. Also, compiled code will sometimes suffer run-time errors that are specific to the compiler or its version. Some compiler vendors have been very quick to respond to compiler bug reports, and others have ignored our requests for resolution of these errors.