Component-Based Development of CFD Software FUN3D

Xinyu Zhang*
Analytical Mechanics Associates, Inc., Hampton, VA 23666, USA

William T. Jones†, Stephen L. Wood‡, and Michael A. Park§
NASA Langley Research Center, Hampton, VA 23681, USA

FUN3D, a suite of Computational Fluid Dynamics simulation and design tools developed at the NASA Langley Research Center, has undergone continuous development since the late 1980s. It contains a large portion of legacy code. Extending it with new capabilities becomes increasingly difficult. To improve the extensibility and reusability, FUN3D is moving toward component-based development. New features, such as Stabilized Finite Elements, Yoga, and Sparse Linear Algebra Toolkit, are integrated into the system as components. Some existing features such as the Node-Centered Finite Volume Solver, are also being refactored to components. The integration of these components poses new requirements on the development workflow. In this paper, we describe the Continuous Integration of FUN3D to support component-based development, and discuss the tools used, the practices followed, and lessons learned during the transition from the traditional approach.

I. Introduction

FUN3D was born as a research code to study algorithms for unstructured-grid fluid dynamic simulations. Later it grew into a suite of Computational Fluid Dynamics (CFD) simulation and design tools. FUN3D development migrated from a one-code, one-developer paradigm to a team-based development decades ago. Agile software development practices have been adopted and incorporated [1, 2]. Team collaboration has made FUN3D a widely used CFD software tool that has supported many leading edge research projects. Over the years, the complexity of the code has increased significantly. The interaction of simulations from different disciplines, such as structures, acoustics, reacting flow, etc., is needed. Numerous solvers, mesh partitioners, and communication schemes are desired. This is a challenge not only faced by FUN3D but the CFD community as a whole. CFD vision 2030 has suggested that NASA should streamline and improve software development processes [3].

Studies have been carried out at NASA Langley on software design principles involving decoupled components and software interface design [4, 5]. These studies laid the foundation of the approach discussed here. The design of interfaces and components has been adopted in FUN3D development. Components are integrated through well-defined software interfaces that define an Application Programming Interface (API). New features are added to FUN3D, such as the Stabilized Finite Element (SFE) [6] solver, Yoga [7] overset mesh assembly, and Sparse Linear Algebra Toolkit (SLAT) [8] in the form of components that adhere to the software interfaces defined for the respective discipline.

Over the lifespan of the code, many researchers have contributed to the code base. During any point in time, there could be contributions from a group of developers. In component-based development, the team owns the components jointly. Most developers focus on particular components, and occasionally contribute to other components. A Continuous Integration (CI) workflow has been established and new tools have been developed to facilitate the development process. Future FUN3D releases will be based on this design. To avoid version skews, FUN3D is tested as an integrated system and releases all the components as a package. In the following, FUN3D as an integrated system will be referred to as FUN3D INTG, while the FUN3D component refers to the legacy code in the form of a component.

The outline of the paper is as follows. Section II discusses the component-based development approach. Section II.A introduces the component interfaces and section II.B describes the software life cycle. In section II.C, the version control used for component management is described followed by the details of CI in section II.D. Section II.E discusses the testing carried out at component and system levels. Section III summarizes the practices followed in the approach and lessons learned, and discusses its advantages and disadvantages. Concluding remarks are given in the last section.

*Engineer, Analytical Mechanics Associates, Inc., AIAA Member.
†Computer Engineer, Computational AeroSciences Branch, AIAA Associate Fellow.
‡Research Scientist, Computational AeroSciences Branch, AIAA Senior Member, stephen.l.wood@nasa.gov
§Research Scientist, Computational AeroSciences Branch, AIAA Associate Fellow.

Table 1 FUN3D INTG Components.

Component	Description
FUN3D	FUN3D component, a suite of CFD simulation codes
libcore	Utilities for memory allocation and MPI
SFE	Stabilized finite element solver
refine	3D mesh adaptation tool
SLAT	Sparse linear algebra toolkit
NFE	Utilities for grid based operations and preprocessing
Yoga	Overset grid assembler

Table 2 FUN3D Components.

Component	Description
libcore	Utilities for memory allocation and MPI
Sparskit Interface	Sparse Matrix Utility Package (Sparskit) interface implementation

II. Component-Based Development

Component-based development aims at building systems with reusable software units – components. Its main objectives are to increase productivity, save costs, and improve quality [9]. The idea to componentize software was first published at the NATO conference on software engineering in 1968. In the past two decades, component-based development has emerged as a distinguishable approach [10]. The definitions of components and their properties have been discussed by Broy, Szyperski, Meyer, and Lau [9]. The definition given by Szyperski [11] is widely adopted and fits the context here, "a component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Software can also be organized into modules that also have interfaces. However, they are not contractually defined and are highly interrelated. N modules have interactions grow to $\frac{N^2}{2}$. Kleb and Wood showed the complex interactions of the modules in the FUN3D flow solver [12]. FUN3D has long been modularized. For example, there are modules of turbulence models, grid movement, boundary conditions, a GPU library to solve perfect and generic gas flow (FLUDA) [13], high energy flow solver synthesis (PHYSICS MODULE), etc. However, these modules are not identified as components here. They can hardly be reused by third parties. Components are different from modules. Components are created by applying the Dependency Inversion Principle and are decoupled [5]. In Figures 1, 2, and 3, components have an icon on the upper right corner of their boxes.

In FUN3D INTG, a "driver" component communicates with all other components. Currently, part of the FUN3D component acts as the driver. It is being refactored to be solely a coordinator. A component can be a subsystem/composite of other components. The current components of FUN3D INTG are listed in Tables 1 and 2. The refactoring and development of these components is still an on-going process. As multiple components may depend on the same subcomponent, common subcomponents are elevated to the component level to ensure consistency and avoid version skew. Figure 1 is the current layout of FUN3D INTG. Both FUN3D and NFE have subcomponent *libcore*, see Figure 2. Each may use a different version of *libcore*. However, when integrated into FUN3D INTG, *libcore* is elevated to the component level and its suitable version is defined in the system. An advantage of the component-based approach is that components can be easily reused. Figure 3 shows other systems that are created by integrating some of the components. Node-Centered Finite Volume Solver (NCFV) is currently being refactored from the legacy finite-volume solver. The two systems, FUN3D NCFV and FUN3D SFE, focus on research into different types of nonlinear solvers. Furthermore, FUN3D INTG does not rely on any framework, and the components can be reused in other systems as well.

All components use GNU Autotools [14] to configure and build. Some components also build with CMake [15], but the integration build and distribution utilize the GNU Autotools subpackages. Documentation is also assembled at the system level. Each component can have its own manual in LaTeX [16], while FUN3D INTG creates a final manual for distribution by extracting chapters and sections from each component.

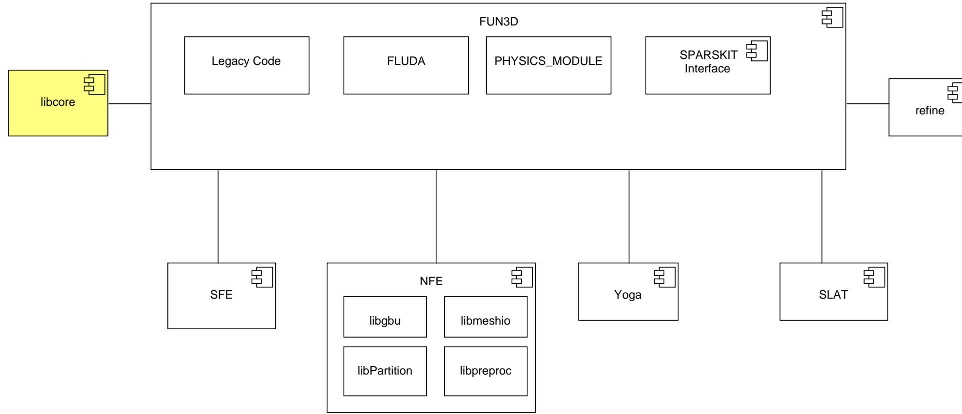


Fig. 1 FUN3D INTG System.

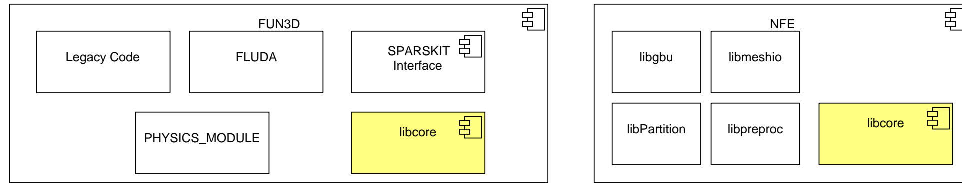
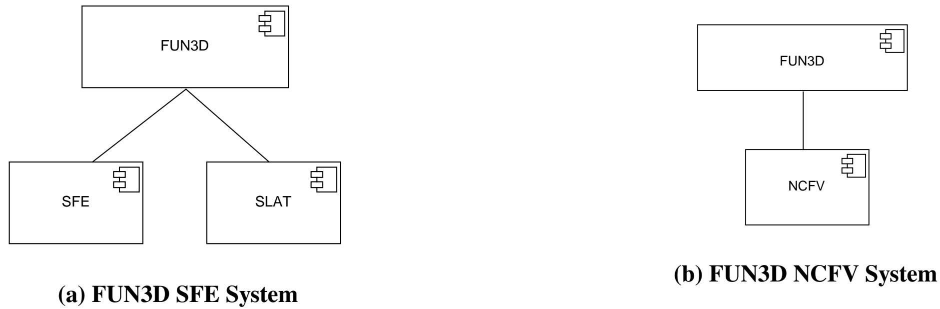


Fig. 2 FUN3D and NFE Components.



(a) FUN3D SFE System

(b) FUN3D NCFV System

Fig. 3 Two Systems with Some of the Components.

A. Component Interfaces

Components interact through interfaces. Well-defined interfaces are critical to the system integration. A suite of software interfaces has been developed and implemented by FUN3D INTG components. This includes parallel communication, problem definition, mesh access, linear solver execution, nonlinear iterative solver execution, and solution access. The interfaces are language independent and aim to support the largest spectrum of programming models. The use of interfaces places the dependency on the interface instead of the implementation in components. The interfaces are rigid and follow the open-closed principle [17] in that they are open to extension, but closed to modification. Strict adherence to the interface definition guarantees interoperability between components. Figure 4 shows the components with interfaces and Listing 1 is a snippet of the mesh interface. The mesh interface defines the methods to retrieve data from a partitioned mesh and obtain information about node residency. The full definition of the interfaces has been published and discussed in Ref. [4].

Listing 1 Mesh Interface

```
int64_t tinf_mesh_node_count(void* const mesh, int32_t* error);
int64_t tinf_mesh_resident_node_count(void* const mesh, int32_t* error);
int64_t tinf_mesh_element_count(void* const mesh, int32_t* error);
int64_t tinf_mesh_element_type_count(void* const mesh, const enum TINF_ELEMENT_TYPE type, int32_t
    * error);
int32_t tinf_mesh_nodes_coordinates(void* const mesh,
                                        const enum TINF_DATA_TYPE data_type,
                                        const int64_t start, const int64_t cnt,
                                        void* x, void* y, void* z);
enum TINF_ELEMENT_TYPE tinf_mesh_element_type(void* const mesh,
                                                  const int64_t element_id,
                                                  int32_t* error);
int32_t tinf_mesh_element_nodes(void* const mesh, const int64_t element_id,
                                    int64_t* element_nodes);
int64_t tinf_mesh_global_node_id(void* const mesh, const int64_t local_id,
                                     int32_t* error);
int64_t tinf_mesh_global_element_id(void* const mesh, const int64_t local_id,
                                        int32_t* error);
int64_t tinf_mesh_element_tag(void* const mesh, const int64_t element_id,
                                  int32_t* error);
int64_t tinf_mesh_element_owner(void* const mesh, const int64_t element_id,
                                    int32_t* error);
int64_t tinf_mesh_node_owner(void* const mesh, const int64_t node_id,
                                 int32_t* error);
int64_t tinf_mesh_partition_id(void* const mesh, int32_t* error);
```

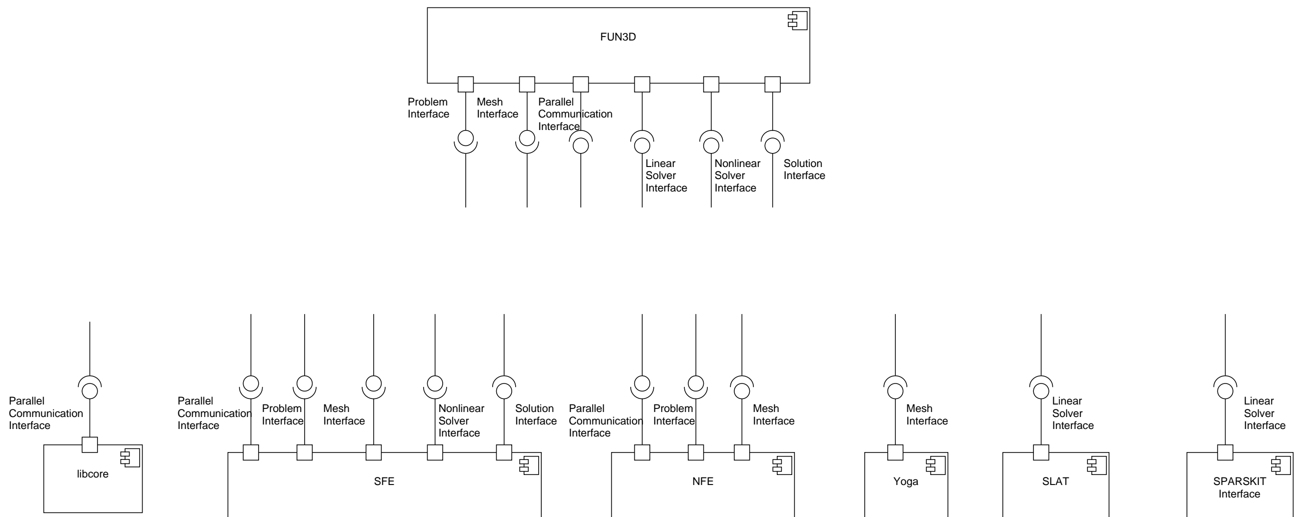


Fig. 4 FUN3D INTG Components and Interfaces.

B. Life Cycle

The FUN3D INTG process is bottom-up and better described as a W model. Lau et. al. discussed the model in Ref. [9]. Figure 5 illustrates the FUN3D INTG W model. It joins two V models of component and system life cycles. Domain knowledge is applied to the component design, implementation, and verification and validation. The component is then integrated into the system by system assembly. The W model allows more flexibility in the component life cycle. Developers can explore novel ideas and choose whether to expose the contents to the system. FUN3D supports cutting edge research and this benefit weighs heavily. Some software has a designated group on testing and integration. For example, in CREAVE-AV's annual release life cycle, the Quality Assurance Group carries out Product Acceptance Test (PAT) for a period of time before beta release [18, 19]. However, FUN3D does not have these resources and relies

mostly on the CI for testing. In addition, the W model also reflects the CI. As the new features are implemented in a component, they are first tested in the component level and then in the system integration. Once a component is initially integrated, the update is usually carried out by an automated process in the CI and requires little or no coding. Components can also be developed and tested in a different system before being integrated into the FUN3D INTG. For example, a system with FUN3D as a driver for SFE was created during the initial development of the SFE component and is still being used to provide a faster verification of the limited integration of only FUN3D and SFE. The NCFV component, which is being extracted and refactored from the FUN3D code is currently developed in another system comprised of just FUN3D and NCFV. It will be integrated into FUN3D INTG when it is mature.

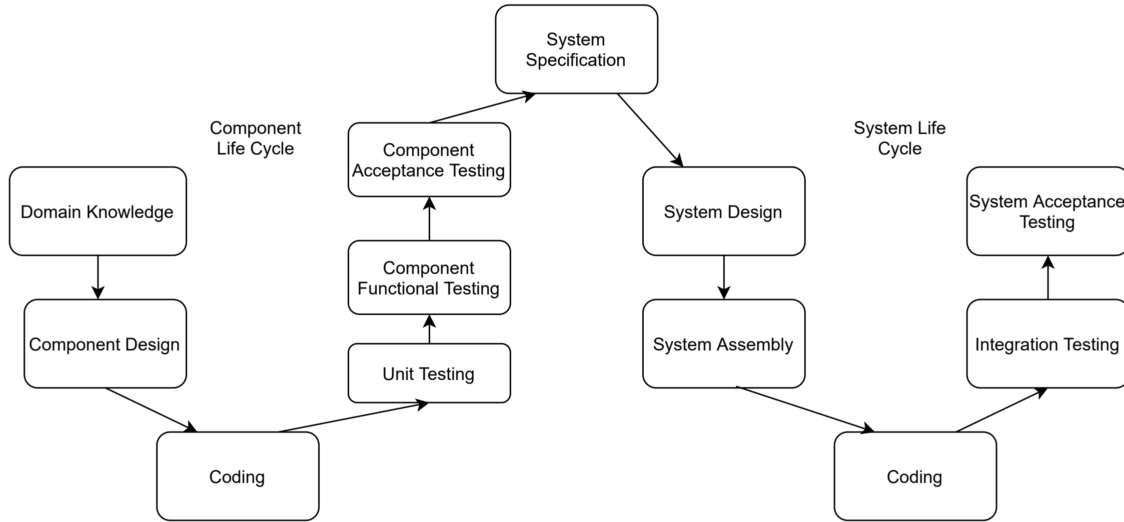


Fig. 5 The W Model of FUN3D INTG.

C. Version Control

Git [20] is used for version control. Although some businesses adopt a single repository model of source code management, it requires in-house tools to support the operations [21]. A monolithic Git repository is hard to maintain with the increased size. When using Git, it is usually preferred to have smaller repositories [21]. Git provides submodule tools so that a repository can be a subdirectory of another repository. Each component or system is a Git repository. The components are assembled into a system through submodules. A component is a submodule in Git, but a submodule may not be a component.

A Git submodule can track a branch and always point to its head. However, the update in a component may not pass the integration tests. So instead, a FUN3D INTG submodule points to a specific SHA. The SHA reference in a submodule is advanced with an automated commit triggered by the successful completion of a CI build pipeline. As a practice, only SHAs from a protected branch (usually the main branch) are allowed so the history is preserved, and the integrated components are ensured to have passed both the component tests and the integration tests. Other than FUN3D components, test cases and tools that are used for managing the build system and testing are also submodules. The software interface definitions are referenced as a submodule as well.

Most of the FUN3D component repositories have one protected branch, the main branch, and run a simple workflow. However, some components use a second protected branch designated here as the develop branch, see Figure 6. Feature branches are merged into the develop branch after passing a subset of merge request tests. More rigorous tests run on the develop branch before it is merged into the main branch. Merging from the develop into the main branch is a one-to-one merge. It does not have the potential issue of functional conflicts, so the *develop* tests do not need to run on the main branch again. The process is automated, and the main branch is always release ready. This has been applied to components that include lengthy tests such as FUN3D and SFE.

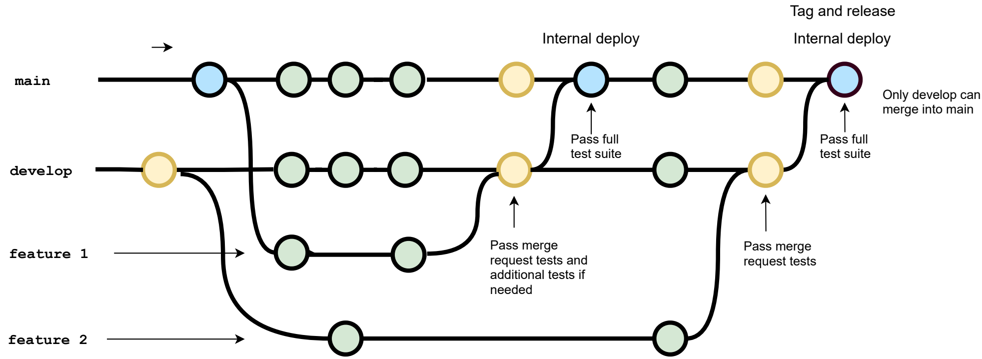


Fig. 6 Git Branches.

D. Continuous Integration

FUN3D INTG uses GitLab-CI. Comparing previous experience with the Jenkins CI tool, developers have found running CI and managing the Git repositories in the same tool is easier to navigate and causes less confusion in tracking the results. With the advancement of GitLab-CI, there is more flexibility on CI design for different use cases.

Component Pipelines

Each component may have a different workflow that fits its own development style. The workflow of the FUN3D component, currently still a major part in FUN3D INTG, will be discussed here. The FUN3D component has two protected branches, main and develop. A feature branch is merged into the develop branch once it has passed a set of merge request tests. The merge request tests contain unit tests, regression tests, and selected tests that have higher failure frequencies with source code changes. Once merged into develop, it will trigger the full test suite to run on the develop branch merge commit that takes hours to finish and includes performance and acceptance testing. A full test suite pipeline contains more than 50 jobs. When a failure occurs, the committer of the merge will be notified by email. A list of users who watch the develop branch status will also receive a notification. It is possible for the develop branch to have failed tests relative to the full pipeline. Any subsequent merge into develop will inherit the problem(s). While there may be some collaboration among developers to resolve the issue(s), the committer from the source of the problem usually takes the responsibility to make the fix. Not until all of the tests pass, does the develop branch with the commit merge into the main branch. At that point, built executables are deployed for internal users. The committer will also be notified that the code change is in the main branch.

The following use cases of how developers use the GitLab-CI are considered. There are a large number of tests, so debugging a failure can be challenging. It is helpful to provide developers various ways to run a specific or a set of tests.

1) Submit a merge request and run a subset of merge request tests. This is the standard way of code merging. The subset of tests includes unit tests and regression tests that cover failures that typically accompany a code change.
2) Submit a merge request, run the set of merge request tests and some additional tests, or the full test suite. The additional tests are for covering code changes that are known to specific obscure features, but not included in 1) due to infrequent changes or longer testing time. The tests can be added when opening any merge request or editing an existing merge request.
3) Run tests on a feature branch without opening a merge request. This assists developers to debug issues found in one or a set of tests, as the developers may work in a development environment different from the testing environment and/or may not have direct access to the testing environment.

Automated Integration Pipelines

In the CI, integrating components into a system involves multiple Git repositories in our design. Since merging into the main branch of a system requires passing the integration tests, the following use cases are considered when setting up the CI pipelines. The components in Git are submodules of the system repository. The process described here applies to updating submodules, which also include build tools, test cases, etc. Tools based on the GitLab API were created to allow the processes to be fully automated.

1) A component's main branch update triggers the downstream system update. A feature branch is created in the

system and a merge request is submitted automatically. The triggering component watches the process and sends the feedback to its committer about the status of system update. This is applied to components that have infrequent update yet need the integration in the system to be carried out in a timely manner. For example, the update of SLAT triggers its update in the FUN3D SFE system, and the update of FLUDA triggers an update of FUN3D.

2) A component triggers the tests in the downstream system, but does not update the system. A feature branch is created in the system and a merge request is submitted automatically. The triggering component watches the process and sends the feedback to its committer on the status of system tests. The system will close the merge request upon finishing without updating. This is to provide a way to understand the influence on the downstream system.
3) The system updates its components. It checks for updates of the components in their own repositories, creates a feature branch, and opens a merge request. The update is merged once the tests pass. Users on a list will receive the notification of the status. This applies to the update of FUN3D INTG. The automated update runs nightly to advance the SHAs of the submodules. The CI tools allow the system to update all or selected submodules, so the update of some submodules can be temporarily frozen. Figure 7 shows the CI pipelines for this process.

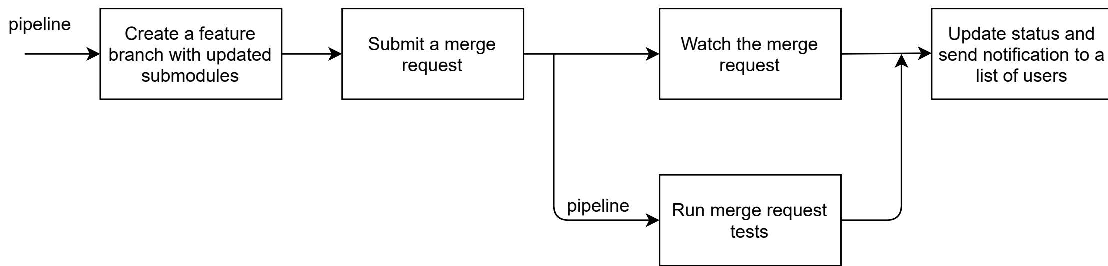


Fig. 7 FUN3D INTG CI Pipelines.

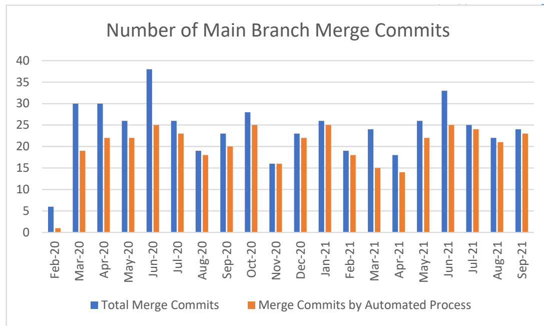
Manual Integration for Coupled Component Updates

The automated update works well with uncoupled submodule changes. For coupled changes, a manual update is needed. A feature branch is created in each component to be updated, and a feature branch in the system is created to point to the respective feature branches in the components. Merge requests are submitted in the components and the system. Once all tests pass, the component feature branches will be merged into their main branches. The system feature branch will then be updated to point to the components' SHAs of their main branches, and then merged into the system's main branch. Most of the merges do not need to go through this process. Figure 8 shows the total number of merge commits on the main branch. The majority of the merges are automated.

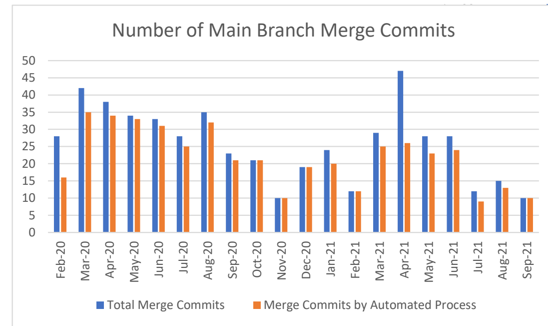
E. Component and Integration Testing

In the component-based approach, each component takes the responsibility of testing its own functionalities. Unit, regression, performance, and acceptance tests are carried out at the component level and run intensively. Metrics, such as code coverage, source lines of code, and compilation timings are also tracked in component testing. Integration tests are carried out in the system level and run less frequently. Most of the testing is carried out on a midrange cluster, the K-cluster at NASA Langley, with dedicated resources. Limited tests are also run at the NASA Advanced Supercomputing facility. The automated tests run on Intel, AMD, and IBM processors as well as on NVIDIA GPUs. Testing on other architectures is periodically done manually by developers due to the lack of access to these resources for the CI.

Unit testing is a way to test the smallest piece of code that can be logically isolated such as a function or a subroutine. Unit tests are encapsulated and do not use external resources, so it is always done in the component level. Languages used in each component may vary. For example, the FUN3D component is mainly in Fortran. SFE and Yoga are in C++. *refine* is in C. The unit testing tools used in components can also vary. Funit is used for Fortran unit test, and Catch2 is applied to SFE and Yoga unit test, while *refine* has it own unit testing framework coded in C. The component-based approach gives the flexibility for each component to select its testing tool. Many code coverage tools are available. The FUN3D INTG components use test coverage by the Intel Code Coverage Tool and LCOV. Both tools generate HTML as well as text reports. When the unit testing runs in the CI pipelines, the compile options are added to the configuration,



(a) FUN3D INTG



(b) FUN3D SFE

Fig. 8 Number of Merge Commits on Main Branch.

and the report is ready once the job finishes.

Regression testing is used to verify the code change does not impact the existing features. Each component has its own regression test suite. These regression tests usually compare the outputs with golden files, which are recorded files with the expected outputs. They are usually matched to machine zero. In some situations such as running on GPUs, specified tolerances are used. Some tests are pulled over from components to the integration testing to ensure that unintended changes do not propagate to the system level. While regression testing checks that no new bugs have been introduced, it does not check if the results are correct. Acceptance testing is also run at the component and system levels to check an implementation's correctness. For the solver type components, these tests require more time as converged results are compared.

Since CFD software usually requires a large amount of computing resources, performance testing is important. Any degradation in the performance can lead to a great increase in the cost when using the software. Performance is tested on various architectures and with various build options. It occurs at both component and system levels.

Some components can easily be isolated and tested alone such as SLAT, *refine*, NFE, and *libcore*. However, some components are difficult to test without other components. For example, the SFE and NCFV solvers. They can hardly function without the implementation of mesh partition and parallel communication. Even for components that can be isolated, a limited testing of integration with other components may also be desired at the component level. This can provide some insight into potential issues of the integration before moving to the full integration. Fixing problems in an early stage usually costs less than addressing them later. Currently there are two modes for FUN3D INTG components to include other components in the testing: using the deployed binaries from other components or creating a system with limited components or alternative components.

- At the component level, a component does not have the knowledge of other components. To use another component in testing, the component can get previously deployed binaries of the dependent component. It may not be consistent with the version of the component that will actually be used in the system level of integration. The focus is to look for influence and potential issues to the system integration.
- Creating a system with limited components or with alternative components helps test the components that are difficult to isolate. The FUN3D SFE system has been used for this purpose. However, due to the FUN3D component's size, the turnaround is slow. A recent effort uses light-weight, faster components to serve as alternatives to the FUN3D component. Alternate preprocessor and visualizer components have been integrated with SFE to test new features of SFE.

III. Practices and Lessons Learned

Component-based development requires collaborations among developers in different groups. Adopting best practices helps in delivering high-quality software. This includes practices in component design, software development, and operations.

In FUN3D INTG, each component is a Git repository. The collection of components is not fixed, and as the code development moves on, the components may be reorganized, combined, or split. For example, the Yoga component was initially brought in with multiple components for message passing and for visualization that Yoga depends on. As these are not reused by other components, they were later grouped into the Yoga component as a subsystem.

There should be no cycles in the component dependency graph, either explicit or implicit. A cycle can cause issues when building the system. This is strictly followed. The Dependency Inversion Principle is applied to establish the interface between components, so that the components are decoupled. *pancake*, the suite of interfaces, is stable and totally abstract. Most of the components depend on it. *refine* can be integrated via an API, but file-based interaction is used in this context to isolate refine from the sweeping FUN3D and FUN3D INTG changes required for component-based development. Integration through the current implementation of interfaces is planned to enable new capabilities like curved mesh file reading, writing, and partitioning. The *refine* library remains accessible through a previous set of interfaces [5].

The interfaces are discussed by developers through biweekly meetings, and then implemented upon agreement. The ideal situation is that the update of interfaces adheres to the open-closed principle, so that the update of the components can be fully decoupled. The workflow is to first update the component that provides the implementation and then the components that consume it. However, in real life, the modification cannot be totally avoided. Coupled update of the components is needed occasionally as interfaces under development mature.

The golden rule of development and operations is to automate as many procedures as possible. Continuous integration and deployment is the key. Developers commit code in small chunks frequently and automated testing runs to detect bugs and integration problems. The software is deployed continuously for internal users to test. Any process that requires manual operation is likely be lost or grow stale over time, and extra effort is required to resume these operations. Since there are many automated processes, providing feedback to the right people is critical to promptly address issues. In each component, as there are a small group of developers, failures from the CI pipelines are reported to the developer who commits the change or the group. In the FUN3D component, where there are some tests that usually require a domain expert to solve the issue, the failure notice is sent to the code committer and the identified experts. The FUN3D INTG is monitored by a small group of focused developers. The failures will first be examined and then determined if any developers or groups need to be reached to resolve the issue.

Each component has its own testing and may have a different style on how the tests are set up. However, the lessons learned in maintaining these tests set some practices that are usually followed.

- Tests that are easy to fail run in the early stage, so that it gives a fast turnaround for the fix. They should also be run frequently. For example, a test in a merge request pipeline runs more often than in the testing pipeline after merge.
- Testing environment mirrors production to guarantee a successful deployment.
- A build is made once and used through the pipeline. This is important to codes that require long build time. For example, in the FUN3D component, there is one develop pipeline build for the generic Intel processors and that build is used for the performance, acceptance, and turbulence modeling tests. Builds are out-of-source, which keeps the generated files separate from the sources, so they can be easily cleaned and multiple builds can be made in parallel.
- Tests are broken into pieces that can easily be rerun. Running automated tests on clusters is subjected to more failure modes, such as occasional file system errors, node malfunction, and network issues. To make the process more robust, tests need to be able to easily rerun. Some tests are set to automatically retry at failure.
- Test outputs clearly show the results and provide information on the configuration and environment to help debug when failed. This is very helpful when running on clusters where many factors may contribute to the failure.
- The outputs of the tests should be easily cleaned.

An observation has been made that when a small number of developers are working on a component, the communication is more efficient. Each group may have its own work style. Figure 9 shows the size of several components in terms of the Source Lines Of Code (SLOC) and Table 3 shows the number of code committers to each component repository in the past 12 months, which to some extent reflects the interactions among developers. Many developers contribute to multiple components and no developers are full-time. They contribute to the code based on the need of the projects. While the communication in the small groups may differ, the FUN3D developers hold a weekly meeting for the whole team. This helps communicate across groups, offers a quick status view, and helps remove impediments.

Notifications and discussions are also through a Microsoft Teams channel and a developers email list.

Table 3 Number of Committers to Each Component Repository in the Past 12 Months.

Component	Number of Contributors
FUN3D	19
libcore	9
SFE	6
refine	3
SLAT	3
NFE	1
Yoga	8

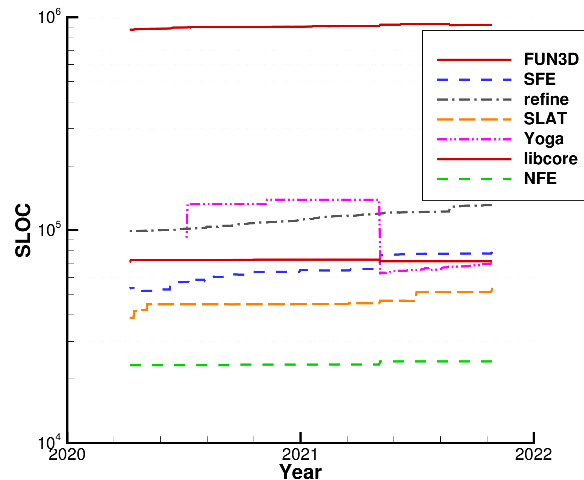


Fig. 9 Component SLOC.

The component-based development also has disadvantages. As the number of components increases, it takes extra effort to keep track of content related to the component other than the source code, such as test cases, documentation, and paperwork. Developers need time to get familiar with the new approach. When extending the interfaces, agreement needs to be made among developers, which requires intensive discussion and collaboration from different groups. However, these impediments can be overcome with some effort and are outweighed by the advantages.

IV. Conclusion

FUN3D development is taking the component-based approach to meet the challenge of the increasing software complexity. This approach improves code reusability and supports fast development. It allows the team to advance the state of the art and deliver production software at the same time. In the paper, the workflow and CI is presented. The integration of the components is mainly automated. Initially, when the component API is under development and changing, the integration of components experienced some difficulty. The process goes smoother with well-defined software interfaces. Currently, the FUN3D component still serves as the driver in the system. It is being refactored and componentized. The dynamically loaded modules have also been implemented recently to aid in the runtime selection of components. These will further ease the system integration and testing.

Acknowledgments

The authors would like to thank the FUN3D development team for the support. This work is carried out at at NASA Langley Research Center Computational AeroSciences Branch. Xinyu Zhang is supported by NASA LaRC TEAMS3 contract.

References

[1] Kleb, W. L., Nielsen, E. J., Gnoffo, P. A., Park, M. A., and Wood, W. A., "Collaborative Software Development in Support of Fast Adaptive AeroSpace Tools (FAAST)," AIAA Paper 2003-3978, 2003.

[2] Fast Adaptive Aerospace Tools (FAAST) Development Team, "Opportunities for Breakthroughs in Large-Scale Computational Simulation and Design," NASA TM-211747, Langley Research Center, Jun. 2002. https://doi.org/2060/20020060131.

[3] Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D., "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences," *NASA Technical Report*, , No. NASA/CR-2014-218178, 2014.

[4] Jones, W. T., Wood, S. L., Jacobson, K. E., and Anderson, W. K., "Interoperable Application Programming Interfaces for Computer Aided Engineering Applications," AIAA Paper 2021-1364, 2021.

[5] O'Connell, M. D., Druyor, C. T., Thompson, K. B., Jacobson, K. E., Anderson, W. K., Nielsen, E. J., Carlson, J. R., Park, M. A., Jones, W. T., Biedron, R. T., Kleb, B., and Zhang, X., "Application of the Dependency Inversion Principle to Multidisciplinary Software Development," AIAA Paper 2018-3856, 2018.

[6] Anderson, W. K., Newman, J. C., and Karman, S. L., "Stabilized Finite Elements in FUN3D," AIAA Paper 2017–77, 2017.

[7] Duryor, C. J., *Advances in Parallel Overset Domain Assembly*, Dissertation, The University of Tennessee at Chattanooga, Chattanooga, Tennessee, 2016.

[8] Wood, S., Jacobson, K., Jones, W. T., and Anderson, W. K., "Sparse Linear Algebra Toolkit for Computational Aerodynamics," AIAA Paper 2020–0317, 2020.

[9] Lau, K., and Cola, S., *An Introduction to Component-Based Software Development*, edited by K. Lau, Series on Component-Based Software Development, World Scientific, 2017.

[10] Vale, T., Crnkovic, I., de Almeida, E. S., da Mota Silveira Neto, P. A., Cavalcanti, Y. C., and de Lemos Meira, S. R., "Twenty-Eight Years of Component-Based Software Engineering," *Journal of Systems and Software*, Vol. 111, 2016, pp. 128–148. URL https://www.sciencedirect.com/science/article/pii/S0164121215002095.

[11] Szyperski, C., Gruntz, D., and S, M., *Component Software: Beyond Object-Oriented Programming*, edited by C. Szyperski, Component Software Series, Addison-Wesley, 2002.

[12] Kleb, B., and Wood, B., "CFD: A Castle in the Sand?" AIAA Paper 2004-2627, 2004.

[13] Nastac, G., Walden, A., Nielsen, E. J., and Frendi, K., "Implicit Thermochemical Nonequilibrium Flow Simulations on Unstructured Grids using GPUs," AIAA Paper 2021–0159, 2021.

[14] MacKenzie, D., Tromey, T., Duret-Lutz, A., Wildenhues, R., and Lattarini, S., *GNU Automake*, Free Software Foundation, Inc, 2020. URL https://www.gnu.org/software/automake/manual/automake.pdf, [Online; accessed 01-May-2021].

[15] *CMake Reference Documentation*, 2021. URL https://cmake.org/help/v3.20/, [Online; accessed 01-May-2021].

[16] *LaTeX2e for Authors*, 2020. URL https://www.latex-project.org/help/documentation/usrguide.pdf, [Online; accessed 01-May-2021].

[17] Martin, R. C., *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Prentice Hall Press, New Jersey, 2017.

[18] Meakin, R. L., Atwood, C. A., and Hariharan, N., "Development, Deployment, and Support of a Set of Multi-Disciplinary, Physics-Based Simulation Software Products," AIAA Paper 2011-1104, 2011.

[19] Bergeron, K., Kendall, R., Hariharan, N., Meakin, R., and Post, D., "The HPCMP CREATE™ Management Model – Part II DevOps Principles and Practices in HPCMP CREATE™," AIAA Paper 2021-233, 2021.

[20] *Git Reference Manual*, 2021. URL https://git-scm.com/docs, [Online; accessed 01-May-2021].

[21] Potvin, R., and Levenberg, J., "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM*, Vol. 59, 2016, pp. 78–87. URL http://dl.acm.org/citation.cfm?id=2854146.