T-infinity: The Dependency Inversion Principle for Rapid and Sustainable Multidisciplinary Software Development

Matthew D. O'Connell,* Cameron T. Druyor,* Kyle B. Thompson,†
Kevin E. Jacobson,‡ W. Kyle Anderson,§ Eric J. Nielsen,¶
Jan-Reneé Carlson,‖ Michael A. Park,¶ William T. Jones,**
Robert T. Biedron,†† and Bil Kleb‡‡
NASA Langley Research Center,
Hampton, VA 23681, USA

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

The CFD Vision 2030 Study recommends that, "NASA should develop and maintain an integrated simulation and software development infrastructure to enable rapid CFD technology maturation. ... [S]oftware standards and interfaces must be emphasized and supported whenever possible, and open source models for noncritical technology components should be adopted." The current paper presents an approach to an open source development architecture, named T-infinity, for accelerated research in CFD leveraging the Dependency Inversion Principle to realize plugins that communicate through collections of functions without exposing internal data structures. Steady state flow visualization, mesh adaptation, fluid-structure interaction, and overset domain capabilities are demonstrated through compositions of plugins via standardized abstract interfaces without the need for source code dependencies between disciplines. Plugins interact through abstract interfaces thereby avoiding N^2 direct code-to-code data structure coupling where N is the number of codes. This plugin architecture enhances sustainable development by controlling the interaction between components to limit software complexity growth. The use of T-infinity abstract interfaces enables multidisciplinary application developers to leverage legacy applications alongside newly-developed capabilities. While a prototype is demonstrated herein, a description of interface details is deferred until the interfaces are more thoroughly tested and can be closed to modification.

Nomenclature

I. Introduction

Significant progress has been made in Computational Fluid Dynamics (CFD), computational structures, and other simulation disciplines. Simulations that combine several disciplines can become fragile if low-level implementation details are directly coupled between disciplines. For example, if discipline "A" has direct, source-code dependence on

*Research Scientist, Computational Aerosciences Branch, AIAA Member.
†Research Scientist, Aerothermodynamics Branch, AIAA Member.
‡Research Aerospace Engineer, Aeroelasticity Branch, AIAA Member.
§Senior Research Scientist, Computational Aerosciences Branch, Associate Fellow AIAA.
¶Research Scientist, Computational Aerosciences Branch, Associate Fellow AIAA.
‖Research Scientist, Computational Aerosciences Branch, Senior Member AIAA.
**Computer Engineer, Computational Aerosciences Branch, Associate Fellow AIAA.
††Senior Research Scientist, Computational Aerosciences Branch.
‡‡NASA Senior Researcher for Computational Aerothermodynamics, Associate Fellow AIAA.
§§Engineer, Analytical Mechanics Associates, Inc., AIAA Member.

details within discipline “B” then a change in “B” can have unintended consequences within “A”. As N codes are coupled in this fashion, the development complexity scales as N^2 due to the interaction between these exposed implementation details. Within a discipline, coupling low-level implementation details retards their reuse; the choice is often made to reimplement mature, well-tested functionality and thus increase software development and maintenance costs.

The issues impeding multidisciplinary analysis and design were also identified by the Slotnick et al.’s CFD Vision 2030 Study [1]. In particular, Programmatic Recommendation 2 of the Study states,

> NASA should develop and maintain an integrated simulation and software development infrastructure to enable rapid CFD technology maturation. ... In order to be sustainable, dedicated resources must be allocated toward the formation of a streamlined and improved software development process that can be leveraged across various projects, lowering software development costs, and releasing researchers and developers to focus on scientific or algorithmic implementation aspects. At the same time, software standards and interfaces must be emphasized and supported whenever possible, and open source models for noncritical technology components should be adopted.

In order to enable increasingly multidisciplinary simulations, for both analysis and design optimization purposes, advances in individual component CFD solver robustness and automation will be required. The development of improved coupling at high fidelity for a variety of interacting disciplines will also be needed. This will require advances in foundational technologies, as well as increased investment in software development, since problem and software complexity continue to increase exponentially. In practice, domain experts are distracted from focusing on their discipline due to the complexity of creating and maintaining an entire software environment necessary to evaluate their methods. The proposed environment leverages the substantial investment in existing tools, allows extension for developing new technology, and helps combat the increasing demands of bug fixes and application support that come with increased usage.

Keyes et al. [2] provide a survey and status of software for multiphysics in the context of challenges with current approaches and opportunities for improvement that shares themes with the CFD 2030 Vision Study. They provide a thorough review of available frameworks and toolkits. For example, the CREATE-AV [3] project is a multidisciplinary simulation system for air vehicles that connects software components with a software integration framework (SIF) [4]. However, CREATE-AV’s restrictive release policy and SIF prevent these software components from being reused outside of CREATE-AV. Other examples of solver frameworks are OpenFOAM [5], Unicorn [6], LAVA [7], and SU2 [8]. Components within these frameworks are not engineered to be leveraged externally. The Trilinos [9–11] and PETSc [12] projects are primarily toolkits, consisting of cohesive classes that can be mixed, matched, and reused to solve different problems. They provide inspiration for the current approach, but expose more fine-grain implementation details than the approach described herein. In this work, a cohesive set of standardized abstract interfaces (named T-infinity) are used to explore a software infrastructure with the goals of Programmatic Recommendation 2 in mind. T-infinity has an open-source Apache 2.0 license to promote widespread use. This allows academia, industry, government, and commercial vendors to freely distribute plugins (software components that adhere to an abstract interface) or impose distribution restrictions to protect their critical technology. The Dependency Inversion Principle [13] is used to combat the tendency to create arbitrary direct interactions that incur an N^2 cost of change, where N is the number of components. By using the Dependency Inversion Principle (see Section III), plugins are agnostic to the implementation details of other software plugins but still allow the normal flow of control from high-level abstraction to low-level detail. Modifications to implementations are isolated to individual plugins, which allows the details of solving a problem to be wholly the responsibility of that plugin. There are already many CFD applications developed both within NASA and externally, each with their own user base and workflow.

T-infinity does not seek to replace these applications. Instead, the goal of T-infinity is to enable faster enhancement of these applications, and to allow the technologies underpinning these applications to be more widely tested and adopted. Illustrative example applications are presented in Section IV to demonstrate the benefits of standardized abstract interfaces. T-infinity is still under development, and exploration of additional interfaces is anticipated. The examples in Section IV demonstrate the use of existing interfaces for steady state simulations using unstructured computational fluid dynamics, structural dynamics, unstructured mesh adaptation, overset assembly, pre and post processing, and mesh deformation. Detailed interface descriptions are deferred until they can be closed to modification as described by Martin [14].

II. Architecture Requirements

A software environment must satisfy a number of design requirements to support the CFD 2030 Vision. Federated development, meaning the collaboration of decentralized and autonomous development teams, is a key outcome of adhering to these requirements. These requirements include publishing abstract interfaces with an open source license, allowing, but not requiring, the use of externally developed tools and emerging hardware, promoting sustainability through abstract interfaces, and judiciously determining the correct level of detail for these interfaces.

The development environment must rely on open source interfaces. Contributions from academia, government, and industry must be able to be assembled and leveraged. Some institutions will desire their contributions to be widely released under open source licenses such as GPL.* Alternatively, some institutions will not wish to distribute their software plugins either for reasons of commercial competitiveness, or national security. The interfaces must be released with a permissive open source license that is compatible with all use cases.

The software environment should allow flexibility in the use of third-party tools and emerging hardware. Computing architecture is expected to change rapidly over the next decade. Simulation tools will have to rapidly adapt to this new hardware with updated algorithms, libraries, and compilers. The development environment must not hinder rapid adoption of this new wave of technology. Federated development is essential to achieving stability and relevance in the face of disruptive changes.

Coupling software with standardized abstract interfaces avoids N^2 development effort, where N refers to the number of software packages being coupled. Without abstract interfaces, Foote and Yoder [15] assert that the software devolves into a "Big Ball of Mud." Standardized abstract interfaces allow software components to be interchanged. This interchangeability is valuable for both redundancy in capability and reproducibility of scientific results.

The level of detail in the interfaces must be carefully determined. The CFD Vision 2030 Study provides the caution that, "frameworks and standardization can lead to significant constraints and may not be the best solution in all cases." The Earth System Modeling Framework [16] has a caution, "focus on interoperability between relatively large coarse-grained components and leave internal numerics (another key component in overall performance) to be implemented in the manner deemed most efficient by applications developers." Trilinos [11] is organized to prevent, "unduly compromising package team autonomy." The impact that the level of detail has on an interface can be illustrated by a case study in mesh databases and interfaces used for mesh generation and adaptation. For example, GridEx [17, 18] provides a very high-level interface where a single function call returns a surface face mesh or a volume mesh. Meanwhile, the Unstructured Grid Consortium [19, 20] defines a much lower level interface where individual vertices and elements can be added and removed from a mesh database. The former choice of a high-level interface places the onus of data structure selection and performance on the underlying component, while the latter choice of exposing low-level operations in the interface places the onus on the framework for data structure selection and performance. Interfaces should provide an abstraction to hide certain details but not hide too much information to the point of lying or leaking the abstraction.†

An environment that fulfills all these design requirements leads to a software architecture made of independent, but cohesive, plugins, where each plugin can be designed, tested, developed, and deployed independently of any other plugin. T-infinity enumerates a subset of capabilities that fall within the scope of the CFD 2030 vision, and isolates them from one another by defining abstract interfaces that explicitly dictate the behavior of each capability. An application has the option of using one or more of the T-infinity abstract interfaces. This allows a legacy application to be incrementally modified to use the services provided by other implementations that adhere to the T-infinity abstract interfaces. This flexibility differentiates the current approach from an infrastructure that requires complete adoption to function. The environment should limit the number of requirements a software plugin must meet while allowing the maximum flexibility of any individual plugin.

Consider the center of Figure 1, which depicts an application that requires the use of a number of capabilities on the periphery in order to solve a complex engineering problem. Solid black arrows point toward T-infinity abstract interfaces that define the behavior of each required capability. Examples of existing software packages that implement capabilities required by the abstract interfaces are shown in orange, where this implementation relation is depicted with an open dashed arrow. These software packages were developed by various teams, from multiple institutions, in multiple locations, and in multiple languages. T-infinity abstract interfaces minimize the necessary interactions between each plugin, which makes collaboration possible while allowing different teams to work independently.

*GNU General Public License `https://www.gnu.org/licenses/gpl-3.0.en.html`, last accessed 9 May 2018.

†Joel Spolsky, "The Law of Leaky Abstractions" (2002) `https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/`, last accessed 9 May 2018.

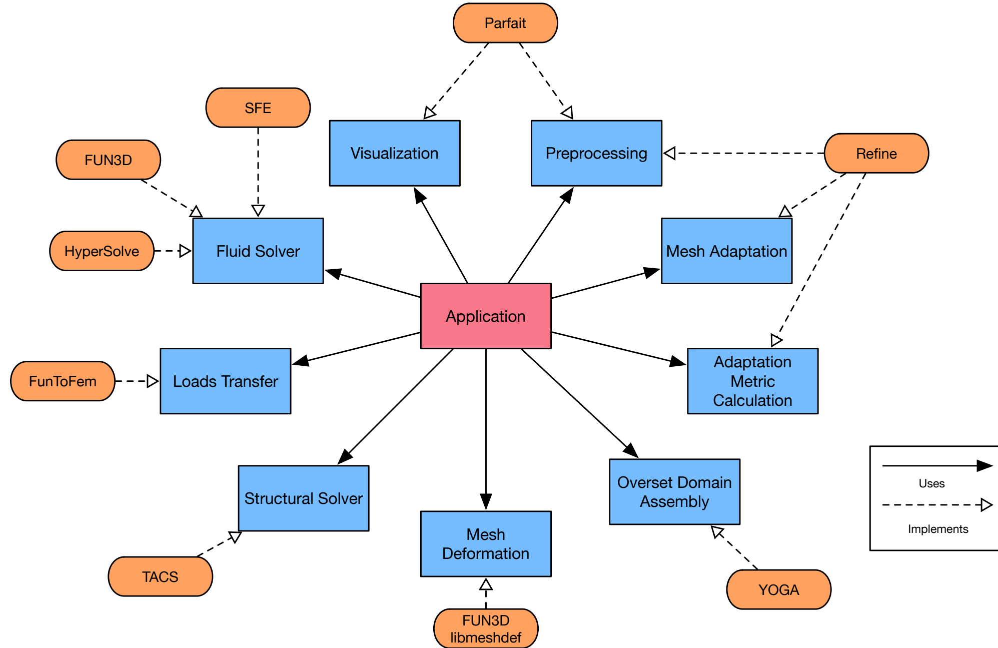


Fig. 1 Some of the T-infinity abstract interfaces for concepts, and some existing software packages that provide implementations of those interfaces.

For example, the Stabilized Finite Elements (SFE) [21] package implements the Fluid Solver abstract interface; the refine package [‡] implements the Preprocessing, Mesh Adaptation, and Adaption Metric Calculation abstract interfaces; and the FUNtoFEM [22, 23] package implements the Loads Transfer interface. None of these packages are directly coupled. Yet because they fulfill their respective T-infinity abstract interfaces, they can be used in concert. Several cases that demonstrate this flexibility are presented in Section IV.

III. Dependency Inversion Principle

Dependency Inversion is one component of the SOLID design principles for Object Oriented Programming (OOP). A more thorough discussion of SOLID can be found in Martin [14],[§] which summarizes the Dependency Inversion Principle as,

> High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details, details should depend on abstractions.

Figure 2 illustrates a direct dependency. In this example, ParMETIS [25], a graph partitioner, is directly called by the preprocessing code. The arrows connecting the preprocessor to ParMETIS show the direction of dependency between the two software modules. Figure 3 generalizes this software strategy where a high-level concept (preprocessing) has a source code dependency on the low-level detail (calling ParMETIS).

Calling ParMETIS is not a direct requirement for mesh preprocessing. When thinking abstractly, mesh preprocessing needs something that can partition and ParMETIS provides the partition function. The high-level concept (preprocessing) only relies on an abstract concept (partitioning). The Dependency Inversion Principle embodies this abstract thinking and inverts the direction of the source code dependency while still allowing the normal flow of control from high-level code to low-level code. If ParMETIS is chosen as the partitioner, the software developer would then write a ParMETIS

[‡]https://github.com/NASA/refine

[§]There is not a single point of genesis for the SOLID principles. Some of the design principles, such as the Liskov Substitution Principle, come from the work done by computer scientists in the late 1980s [24]. Robert C. Martin (known in the software community as “Uncle Bob”) is credited with first promoting the SOLID principles on his Object Mentor website and on the Usenet newsgroup `comp.object`. Initially, they were simply known as the “five first principles” during Usenet discussions and on Uncle Bob’s website. Later, Michael Feathers rearranged the order of these principles to form the mnemonic acronym SOLID.

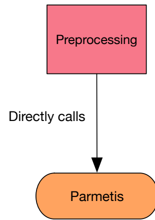


Fig. 2 Typical software strategy: Preprocessing a mesh requires ParMETIS.

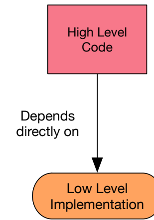


Fig. 3 Typical software strategy: high-level concepts depend directly on low-level details.

adapter that implements the Partitioner interface rather than directly calling ParMETIS from *within* the high-level preprocessing code. As shown in Figures 4 and 5, there is no longer a path of source-code dependency from the preprocessor to ParMETIS but the flow of control remains unchanged. Both the preprocessing application and the ParMETIS implementation depends on the abstract interface, and so the second dependency arrow direction has been inverted.

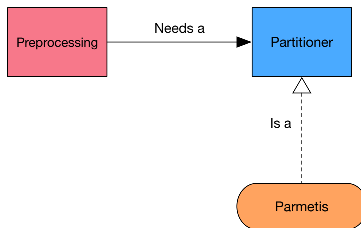


Fig. 4 Dependency inversion: Preprocessing a mesh requires a partitioner and ParMETIS is a partitioner.

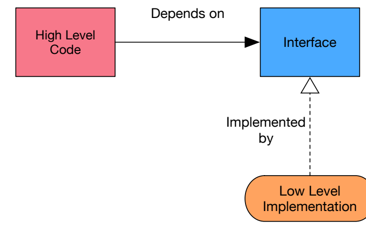


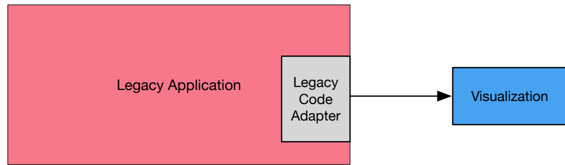
Fig. 5 Dependency inversion: A high-level concept depends on the abstract interface and a low-level detail implements the interface.

With the Dependency Inversion Principle, the high-level code is completely agnostic to the details of partitioning. Changes may come from a new version of ParMETIS, or by the desire to try out a new partitioner that uses different technology. These modifications will have no impact on the preprocessor because of the decoupling that is enforced by the inverted dependency.

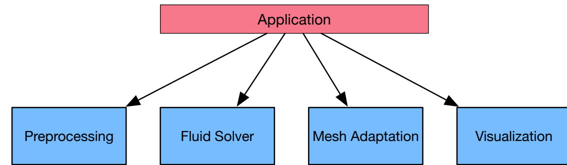
From the calling code's perspective, all of the details of solving the partitioning problem are wholly the responsibility of the partitioning plugin. The preprocessor can be written in any language that can interact with the adapter and has the freedom to depend on any software it chooses.

Software plugins implementing the T-infinity abstract interfaces could be distributed as separate libraries allowing their reuse. Application developers have many options on how they may choose to leverage these libraries. Existing legacy applications may choose to only leverage a single T-infinity abstract interface to gain one specific feature. An application developer would write an adapter from the application's existing data structures to functions defined by the T-infinity interface. For example, Figure 6a shows a legacy application using the T-infinity abstract interface for visualization. An adapter would have to be written for the legacy application's current mesh data structures that would fit the method calls in the T-infinity MeshInterface. Then the application could call the T-infinity visualization interface. The legacy application would need to write an adapter one time but would then be able to use any library that implemented the T-infinity interface without any further code modifications.

Alternatively, new applications could dynamically load T-infinity-compatible plugins without writing adapters. The T-infinity abstract interfaces were designed to be independent, yet cohesive. Figure 6b shows how a simulation involving multiple T-infinity plugins could be controlled from a flexible, dynamic driver. The dynamic driver would not directly interact the underlying data passed between libraries that implement the T-infinity-compatible plugins; and therefore, no adapters would need to be written to adapt the data. However, the T-infinity abstract interface calls would need to be



(a) Legacy codes could choose to adapt their current data to be input to individual plugins.



(b) The output of one plugin is designed to be compatible with the input of another. Simple coordinating applications can be written without writing data adapters.

Fig. 6 T-infinity plugins can be used flexibly. The plugins are designed to be decoupled, yet cohesive.

wrapped into the application's language if the language could not natively call T-infinity interfaces. This is the strategy used for the examples in Section IV.

IV. Illustrative Applications

The following demonstration applications were selected to motivate the use of T-infinity abstract interfaces by demonstrating the flexibility afforded by the modularization that the abstract interfaces enforce. Each plugin is more useful and powerful when it can be used in conjunction with other plugins than it would be on its own. Each selected case includes a diagram of the T-infinity-compatible plugins used by the application, and pseudo code to describe the flow of control of the application.

Performing these cases in a monolithic fashion is possible. A monolithic application is hard wired to, and therefore dependent upon, each component that it uses. Therefore, each added dependency increases the complexity and rigidity of the monolith. For example, replacing a component within a monolith with a new component of the same type is a potentially labor intensive and expensive process. If an application instead leverages plugins via a T-infinity abstract interface, replacing a plugin is simply a configuration change and does not require *any* changes at the source code level.

A. Visualization

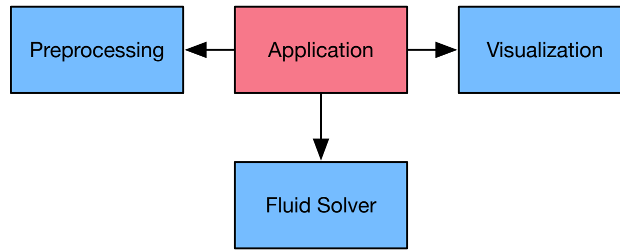
The purpose of the application in this simple example is to solve a fluid problem on a mesh and create a visualization of the solution. This minimal application only needs three T-infinity-compatible plugins, as shown in Fig. 7a. Pseudo code for the application is shown in Fig. 7b.

A mesh object is created on every MPI rank by Preprocessor, and that mesh is given to a Fluid Solver. The Fluid Solver is then instructed to solve the fluid problem. When Fluid Solver completes, the Visualization is instructed to generate a visualization object. The application asks the Fluid Solver for the names of the scalars it can produce, and then instructs the visualization object to add a field for each scalar. Finally, the visualization object is told to visualize the fields. Visualize, in this case, is intentionally vague because an implementation may choose to write a file, publish to a live webpage, draw on an Etch A Sketch®, and so on.

The application could be used with any Preprocessor implementation, any Fluid Solver implementation, or any Visualization implementation that satisfies T-infinity abstract interfaces. Figure 8 shows density contours and the mesh on the symmetry plane and a NASA Common Research Model [26] wing. Also shown are velocity vectors on a midspan cut plane. FUN3D [27] was selected for the fluid solver and used to generate the flow solution, and a binary VTK [28] writer was selected as the visualization plugin and used to generate the files. FUN3D does not need to have an internal method of exporting to VTK on its own because, through the use of T-infinity abstract interfaces, the application can process FUN3D flow solutions with any component that fits the Visualization interface. Not only does this give the user of the application a new post processing option, it does so without encumbering the FUN3D development team with the development and maintenance burden of supporting an additional package.

B. Mesh Adaptation

The ONERA M6 wing mesh adaptation case is described by Park et al. [29]. The T-infinity abstract interfaces used to access the plugins and corresponding application pseudo code needed to perform mesh adaptation can be seen in Fig. 9. The process begins by preprocessing the input mesh. A flow solution is computed and the Mach number field is extracted. The metric is calculated from the Mach field to specify the resolution and element orientation of



(a) Interfaces employed.

```
1  comm = getCommunicator()
2  preprocessor = getPreProcessor('path/to/PreProcessor/plugin')
3  mesh = preprocessor.importMesh('MeshFilename', comm)
4  fluid_solver = getFluidSolver(mesh, comm, 'path/to/FlowSolver/plugin')
5  viz_plugin = getVizualization('path/to/viz/plugin')
6
7  fluid_solver.solve()
8
9  viz = viz_plugin.createViz('OutputFilename', mesh)
10 for f in fluid_solver.listFields():
11     viz.addField(fluid_solver.field(f))
12 end
13 viz.visualize()
```

(b) Pseudo code.

Fig. 7 Flow simulation and visualization example.

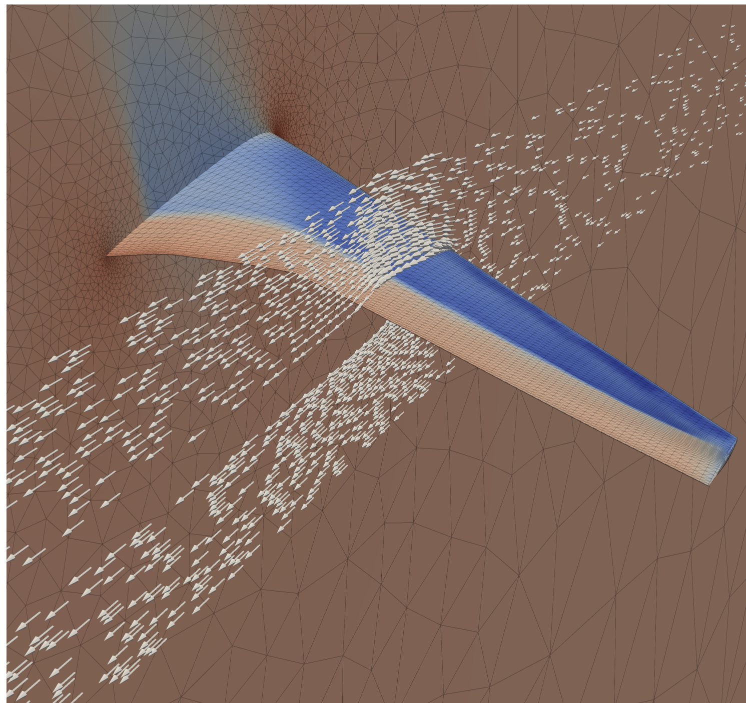
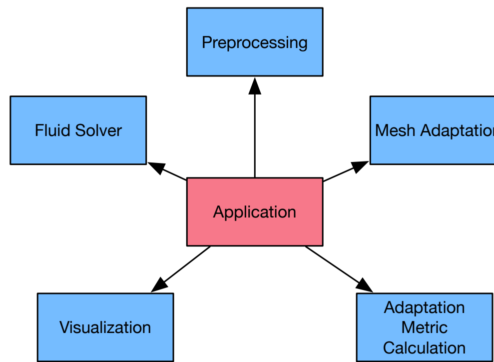


Fig. 8 Density contours and mesh on the symmetry plane and a NASA Common Research Model [26] wing. Also shown are velocity vectors on a midspan cut plane.

the new mesh. The Mesh Adaptation interface is used to transform the existing mesh and metric to the adapted mesh. The solution variables expected by the flow solver are interpolated to the new mesh. The process is repeated without transferring the meshes, solutions, or metrics through files to perform in-core mesh adaptation.



(a) Interfaces employed.

```
1  comm = getCommunicator()
2  preprocessor = getPreProcessor('path/to/PreProcessor/plugin')
3  mesh = preprocessor.importMesh('MeshFilename', comm)
4  metric_calculator = getMetricCalculator('path/to/MetricCalculator/plugin')
5  mesh_adaptation_plugin = getMeshAdaptation('path/to/Adaptation/plugin')
6
7  for step in number_of_adaptation_cycles
8    fluid_solver = getFluidSolver(mesh, comm, 'path/to/FlowSolver/plugin')
9    fluid_solver.setSolution(solution) unless step == 0
10   fluid_solver.solve()
11   mach = fluid_solver.field('mach')
12   metric = metric_calculator.calculate(mesh, comm, mach)
13   new_mesh = mesh_adaptation_plugin.adapt(mesh,comm,metric)
14
15   solution_variable_names = fluid_solver.expectsSolutionAs()
16   solution = fluid_solver.extractFields(solution_variable_names)
17   interpolator = getInterpolator(mesh, new_mesh, comm, refine_dir, refine_name)
18   interpolated_solution = interpolator.interpolate(solution)
19   fluid_solver.unloadPlugin()
20   mesh = new_mesh
21   solution = interpolated_solution
22 end
```

(b) Pseudo code.

Fig. 9 Flow simulation mesh adaptation example.

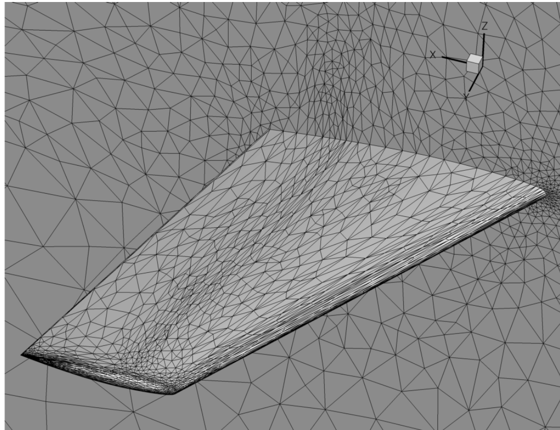
Anderson et al.'s SFE [21] implements a Fluid Solver interface via a Streamlined Upwind Petrov-Galerkin (SUPG) discretion of the Navier-Stokes equations with the SA-neg turbulence model [30]. The shock capturing method described by Anderson et al. is used for this transonic case at Mach 0.84, 3.06° angle of attack, 14.6 million Reynolds number based on root chord, and a temperature of 540 °R. While SFE uses global data (Fortran module variables), it has been modified to reinitialize after finalization to enable in-core mesh adaptation.

A description of the refine open source mesh adaptation mechanics package is provided by Park et al. [29]. It is available via `https://github.com/NASA/refine` under the Apache 2.0 License. It is designed to output a unit mesh [31] in a provided metric field. The refine plugin accesses the geometry through the EGADSlite [32] application program interface. The Preprocessing, Mesh Adaptation, and Adaptation Metric Calculation interfaces are implemented by refine.

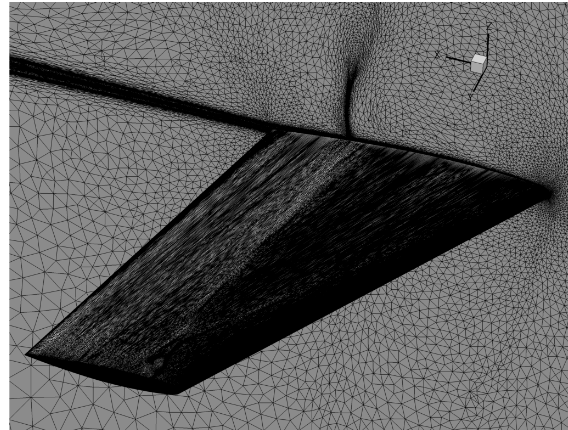
The multiscale metric is formulated to control the L_p-norm of the interpolation error of a solution scalar field. To form the metric, a Hessian of the scalar field is reconstructed by recursive application of a gradient reconstruction scheme. The gradient is computed in each element and a volume-weighted average is collected at each vertex [33]. The metric at each vertex is scaled to control the L_p-norm [33]. The mesh is adapted by refine to conform to the metric with an additional metric constraint imposed to resolve surface curvature.

Park et al. [29] used sequential execution of the refine mesh adaptation mechanics library. In this example, both SFE and refine used 80 cores for the first 13 meshes and 400 cores for the last 6 mesh adaptation steps. Complexity (which is proportional to mesh node count) was held fixed for 3 meshes and then doubled for the next set of 3 meshes. The

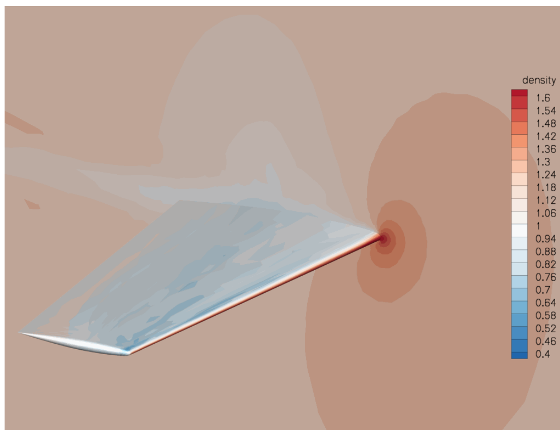
starting mesh for SFE and refine is shown in Fig. 10(a). The mesh after 19 mesh adaptation applications is shown in Fig. 10(b). The resolution of the surface, boundary layer, shock, and wake is dramatically increased. Density for the initial and final SFE SA-neg solution are shown in Fig. 10. The solution in Fig. 10(c) is on the mesh in Fig. 10(a). The solution in Fig. 10(d) is on the mesh in Fig. 10(b). The initial Euler based mesh lacks the appropriate boundary normal resolution necessary to support an SA-neg solution. After a series of mesh adaptation steps the boundary layer and lambda shock system have been resolved by the multiscale metric.



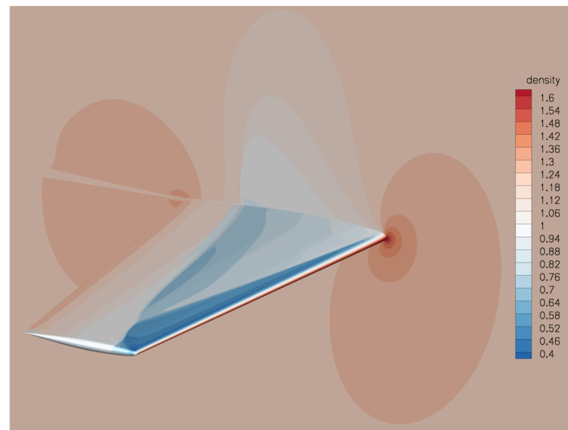
(a) 1st (inviscid) mesh, 57K vertexes.



(b) 19th mesh with SA-neg SFE [21], 3.4M vertexes.



(c) Density, 1st mesh with SA-neg SFE [21], 57K vertexes.



(d) Density, 19th mesh with SA-neg SFE [21], 3.4M vertexes.

Fig. 10 Mesh adaptation for an OM6 wing flowfield, which uses EGADSlite [32] to project the adapted surface grid on the CAD surface.

This example shows how plugins can be assembled to perform in-core mesh adaptation. Other flow solvers that implement the T-infinity abstract interface and allow reinitialization after finalization can also make use of mesh adaptation. The rapid assembly of plugins enables comparison of anisotropic mesh adaptation methods [34] and allows mesh adaptation research by groups that have not developed all the components shown in Fig. 9. The flexibility afforded by the interfaces allows new entrants to the solution adaptation research field and allows mesh adaptation to be performed simultaneously with other disciplines.

C. Fluid-Structure Interaction

A fluid-structure interaction (FSI) analysis capability is demonstrated with an iterative algorithm given in Fig. 11b. The coupled solver begins by creating MPI communicators for the solvers involved in the problem. The transfer communicator is the union of the fluid and structural communicators so that the transfer scheme can operate on data received from both solvers. Next, preprocessor plugins are used to read the fluid and structural meshes, and the plugins

required for the coupling (a Fluid Solver, a Structural Solver, a Transfer Scheme, and a Mesh Deformer) are loaded. The first step inside the solver loop is iterating the Fluid Solver. The surface pressures are then extracted from the Fluid Solver and integrated to get forces at the nodes. Those surface forces are passed to the load transfer, which returns the structural loads. The loads are sent to the Structural Solver. Next, the Structural Solver determines the displacement field of the structure under these loads. The displacement transfer takes the structural displacements and calculates the deflection of the fluid surface. The Mesh Deformer then updates the fluid volume mesh to conform to the deflected fluid surface. Next, the Fluid Solver receives the volume mesh to complete the coupling cycle. This process is repeated until the coupled problem converges as judged by the solver residuals as well as convergence of engineering quantities, e.g., lift, drag, and stress. As shown in Fig. 11, there is potentially a dependence of the load transfer on the structural displacements because load transfers are usually derived from the displacement transfer with the principle of virtual work in order to maintain a consistent and conservative load transfer [35]. The dependence of the load transfer on the displacement transfer through virtual work is also why the load and displacement transfers are treated as a single plugin rather than separate entities. This simple coupling procedure can become unstable in some cases; therefore, Aitken acceleration is used to improve the stability of the coupled solver [36].

The case selected to illustrate the capability of T-infinity for multidisciplinary problems is the undeformed Common Research Model (uCRM) [37]. The uCRM represents an equivalent jig shape reverse engineered from the aeroelastic deflections of the NASA Common Research Model. The structural model is comprised of shell elements that represent the leading and trailing edge spars, the ribs, and upper and lower skins of the wingbox. The Structural Solver plugin is TACS [38]. TACS is a parallel finite-element code that solves thin-walled structural problems with the shell elements based on the mixed-interpolation of tensorial components (MITC) formulation [39]. The Transfer Scheme plugin is FUNtoFEM [22, 23]. FUNtoFEM implements various transfer schemes in C++ such as radial basis functions (RBF) [40] and matching-based load and displacement transfer (MELD) [41]. The volume mesh is deformed in response to the surface mesh deformation with a linear-elasticity technique described by Biedron and Thomas [42]. Two Fluid Solvers, SFE and HyperSolve (an in-house finite-volume code), are applied to the uCRM problem for inviscid analysis of the flow.

The example simulates a transonic cruise condition (Mach 0.85 at 35,000 ft). Since the focus of these examples is the software rather than the correctness of the engineering result, coarse meshes are used; the fluid mesh has 24,187 nodes, and the structural mesh has 10,584 elements. Figure 12 shows the deflection of the wings calculated with SFE and HyperSolve. The structure is intentionally undersized to produce large deflections. The light gray surfaces are the outer mold line of the wing (the CFD surface), and the dark gray surfaces are the structural model of the wing box. The red surfaces show the upward deflection due to lift.

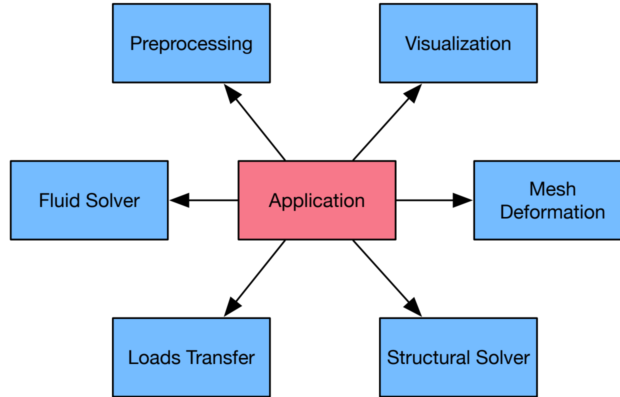
The FSI demonstration case presents some of the important benefits of the application of the Dependency Inversion Principle. One is the additional capability that the individual solvers can leverage by using the T-infinity abstract interfaces. Neither SFE nor HyperSolve have built-in FSI capabilities; however, use of the T-infinity abstract interfaces enables the creation of a steady-state FSI application without the need to develop an FSI-specific interfaces embedded in either fluid solver.

Another benefit highlighted by the FSI example is the elimination of the N^2 coupling problem. Since all the solver plugins in the problem depend on the single set of abstract interfaces defined by T-infinity, TACS, FUNtoFEM, and the volume mesh deformer are unaffected by replacing SFE with HyperSolve. To switch between fluid solvers, the only change necessary is specifying which Fluid Solver plugin should be loaded and which solver-specific settings should be passed to it. This modularity is also helpful when verifying the implementation of a new solver into the environment as well as studying the sensitivity of a result to differences in solvers such as discretization or implementation. Although the two solvers show similar deflected states of the wing in Fig. 12, they are not compared in more detail because it is not within the scope of this paper.

D. Overset Domains

Overset capability in T-infinity is demonstrated on a notional heavy-lift launch vehicle. The vehicle consists of three bodies. The union of the three overlapping meshes define the fluid simulation, with one mesh per body. In an overset simulation, the global MPI communicator is split and each domain is solved on its own communicator.

Figure 13a shows the abstract interfaces used for this case, and Fig. 13b shows pseudo code for the flow of control. The Overset Domain Assembler plugin has two responsibilities. First, the Overset Domain Assembler must generate a list of node IDs on each rank that should be "frozen" (i.e., the Fluid Solver should not solve those nodes). The Overset Domain Assembler is given a mesh, partitioned on the split communicator, and a communicator that spans all of the fluid domains. Second, the Overset Domain Assembler must create a list of receptor node IDs (a subset of the frozen

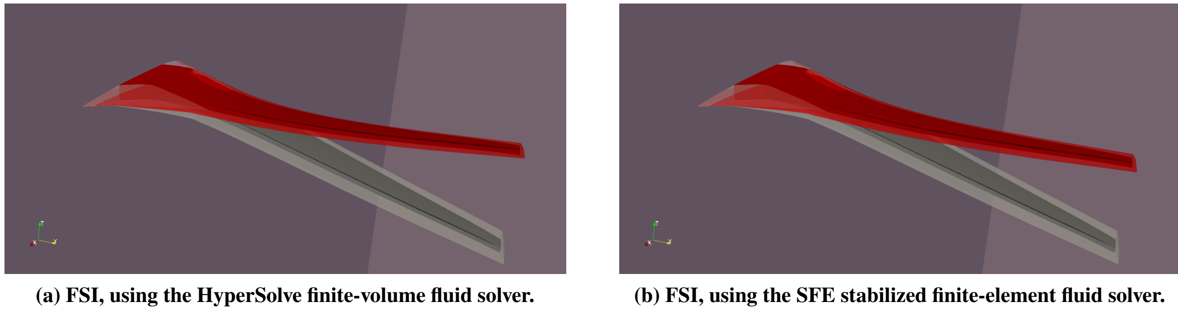


(a) Interfaces employed.

```
transfer_comm, fluid_comm, structure_comm = getCommunicators()

preprocessor = getPreProcessor('path/to/PreProcessor/plugin')
fluid_mesh = preprocessor.importMesh('MeshFilename', fluid_comm)
fluid_solver = getFluidSolver(fluid_mesh, fluid_comm, 'path/to/FlowSolver/plugin')
mesh_deformer = getMeshDeformer(fluid_mesh,fluid_comm,'path/to/MeshDeformer/plugin')

surface_mesh = extractSurfaceMesh(fluid_mesh, surface_boundary_tags)

structure_preprocessor = getPreProcessor('path/to/PreProcessor/plugin')
structure_mesh = structure_preprocessor.importMesh('StructuralMeshFilename', structure_comm)
structure_solver = getStructuralSolver(structure_mesh, structure_comm, 'path/to/StructuralSolver/plugin')

transfer_scheme = getTransferScheme(transfer_comm,fluid_comm,structure_comm,
                                    'path/to/TransferScheme/plugin')
transfer_scheme.setFluidMesh(surface_mesh)
transfer_scheme.setStructuralMesh(structure_mesh)
transfer_scheme.initialize()

for step in number_of_steps
  fluid_solver.solve()
  pressure = fluid_solver.extract('pressure')

  pressure_forces = calculatePressureSurfaceForce(surface_mesh,pressure)

  loads_on_structure = transfer_scheme.transferLoads(pressure_forces)
  structure_solver.setField('loads',loads_on_structure)
  structure_solver.solve()
  structure_displacements = structure_solver.extract('displacements')

  fluid_mesh_surface_displacements = transfer_scheme.transferDisplacements(structure_displacements)

  frozen_nodes = getBoundaryNodes(fluid_mesh)

  fluid_mesh = mesh_deformer.deform(fluid_mesh_surface_deflection,frozen_nodes)

  fluid_solver.updateNodeLocations(fluid_mesh)
end
```

(b) Pseudo code.

Fig. 11 Fluid-structure interaction example.

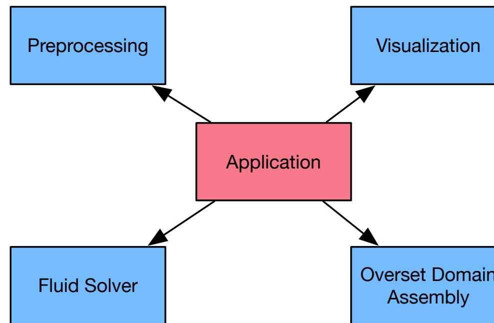


(a) FSI, using the HyperSolve finite-volume fluid solver.

(b) FSI, using the SFE stabilized finite-element fluid solver.

Fig. 12 Steady-state fluid-structure interaction analysis of the uCRM wing. The gray surfaces are the jig shapes, and the red surfaces are the deflected shapes.

nodes, determined by the assembler), and the solution at each receptor. The Overset Domain Assembler determines the solution for each receptor by using a callback from each Fluid Solver that allows the Overset Domain Assembler to ask the Fluid Solver for the solution value at an arbitrary point location within a selected cell.



(a) Interfaces employed.

```
1  global_comm = getCommunicator()
2  weights = [1,1,1]
3  domain_id = assignDomainIds(global_comm, weights)
4  domain_comm = splitCommunicator(global_comm, domain_id)
5  preprocessor = getPreProcessor('path/to/PreProcessor/plugin')
6  mesh_filenames = ['mesh1', 'mesh2', 'mesh3']
7  mesh = preprocessor.importMesh(mesh_filenames[domain_id], domain_comm)
8  fluid_solver = getFluidSolver(mesh, comm, 'path/to/FlowSolver/plugin')
9  domain_assembler = getDomainAssembler('path/to/DomainAssembler/plugin')
10
11 frozen_node_ids = domain_assembler.performDomainAssembly(mesh, solver, domain_id, global_comm)
12
13 for step in number_of_overset_steps
14   receptor_solutions = domain_assembler.updateReceptorSolutions(fluid_solver)
15   setSolutions(fluid_solver, receptor_solutions)
16   performFluidSolve(fluid_solver)
17 end
```

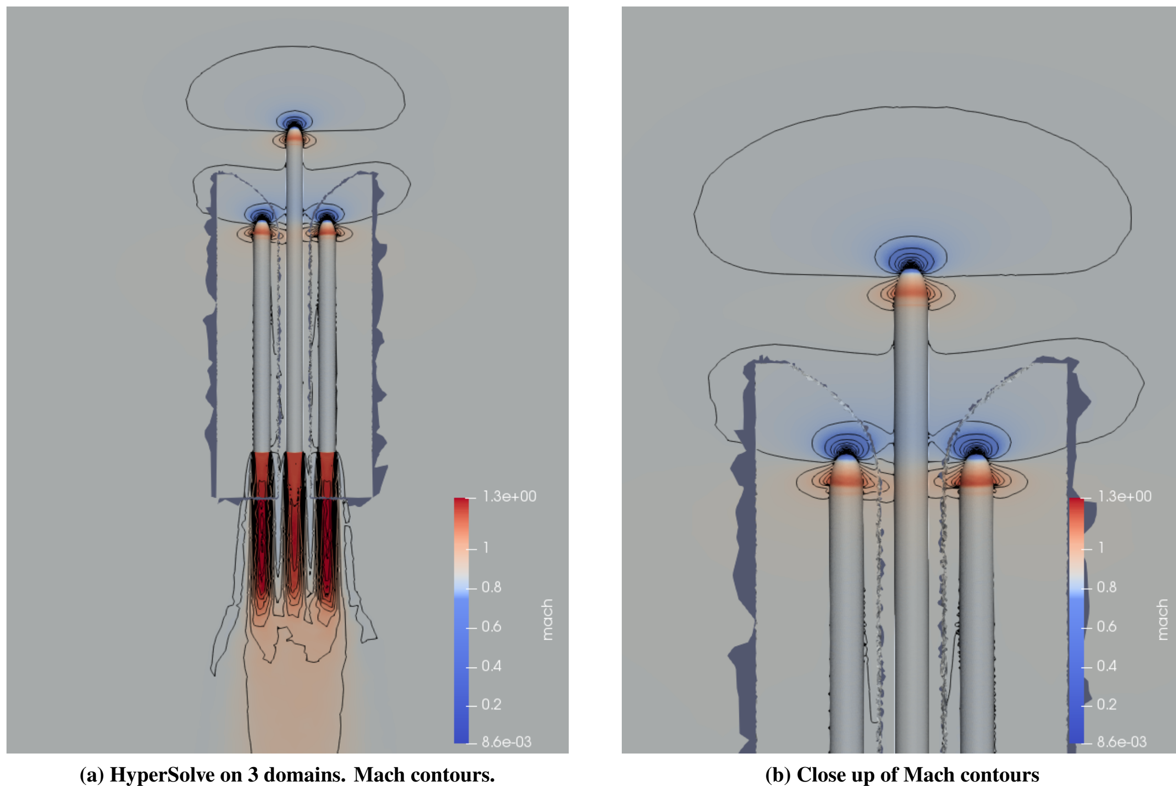
(b) Pseudo code.

Fig. 13 Fluid simulation on overset domains example.

Each rank is assigned to a domain based on the domain weights, where the communicator is split and ranks are assigned to the appropriate communicator to balance these weights. For simplicity, domains are evenly weighted for this example, but they can be generated to suite any given case. For each domain, a Preprocessor is loaded to import the mesh on the domain communicator. A Fluid Solver is loaded for each domain, and given the appropriate mesh and communicator. An Overset Domain Assembler is loaded, and then told to perform the assembly, given a mesh, fluid solver, domain ID, and the global communicator. The Overset Domain Assembler must communicate between multiple domains, so it requires a communicator that contains all of the domain communicators. After the Overset Domain Assembler determines appropriate locations for overset boundaries, it simply returns a list of node IDs on each rank of

nodes that the Fluid Solver shall not solve. Only one assembly is performed for this case because the meshes are not moving or changing between solve steps. Coupling iterations are performed to solve the full domain. Inside of each coupling iteration, the Overset Domain Assembler is instructed to fetch updated solutions for each receptor node. The Overset Domain Assembler is given a callback from each Fluid Solver that allows it to ask the solver to generate the solution at a point location within a particular cell. The Overset Domain Assembler uses this callback to get solution data for each receptor, then communicates receptor data to the appropriate ranks. The output is a list of receptor node IDs and their solution values. Each Fluid Solver's internal solution is updated with the receptor value. The fluid problem is solved on each domain, then the process is repeated.

Figure 14 shows a cut plane of an overset solution on three domains, where cells in the interpolation region have been removed from the visualization to show the location of the overset boundaries. Mach contours are plotted on each domain to demonstrate that the solutions are consistent across overset boundaries.



(a) HyperSolve on 3 domains. Mach contours.

(b) Close up of Mach contours

Fig. 14 Steady overset simulation of notional heavy-lift vehicle.

V. Concluding Remarks

T-infinity has advantages by providing an approach that:

- Presents a useful, but not invasive, level of coupling enforced by extensible interfaces
- Communicates between software modules without forcing a particular data structure
- Minimizes requirements for an optional "framework-level" capability
- Provides an interface set that is "knowable" (small enough to understand completely)
- Avoids N^2 coupling and provides a large set of features while adhering to a small set of discipline interfaces
- Promotes testing to perform good science and allows the evaluation of multiple implementations
- Provides an open set of standard interfaces via Apache 2.0 licensing

T-infinity is intended to empower software developers to sustainably impact simulation capabilities. Today's tools alone are insufficient to address the challenges outlined in the CFD 2030 Vision Study, and new modeling and simulation techniques must be developed to augment existing tools. Improvements must be guided and prioritized by evidence-based

feedback through a rigorous application of the scientific method. Capabilities that have been "almost ready" for years may be quickly brought to bear in production applications. For example, mesh adaptation may become ubiquitous [43] by using a CFD solver interface to access the mesh adaptation example. Collaboration with industry, academia, and commercial vendors can be enabled by releasing the interfaces with a permissive open source license, and unique combinations of plugins will be possible to develop creative applications that were not envisioned by the authors of the plugins.

Examples of these combinations of capabilities include the following. Mesh adaptation can be combined with overset assembly [44] or fluid-structure interaction. Appropriate thermal and chemical models can be automatically loaded as a solution evolves instead of requiring a user to specify them at solution initialization [45]. Combinations are possible that are not anticipated at this moment; and they can grow from end user needs and be rapidly assembled.

Once time-accurate analysis is addressed, the examples in Section IV can be combined to form an overset FSI application with adaptive meshes for rotorcraft analysis. Tools developed for one discipline can be combined with others, e.g., the refine tool developed for CFD mesh adaptation can be enhance the structural solver TACS. Another potential application to explore could be solving a linear elasticity problem for CFD volume mesh deformation with a structural solver plugin.

Combinations of components assembled into a monolithic structure need to be tested as a whole because interactions between integrated components are difficult to discover. Decoupling through the Dependency Inversion Principle turns the number of code paths from N^2 to N. For example, efforts to integrate FUN3D with HELIOS, OpenMDAO, FUNtoFEM, refine, and others are done individually at repeated cost. Integration through T-infinity abstract interfaces allows this integration to be done once for each plugin and unlocks integration with any other T-infinity plugin. The duplication of effort is eliminated by providing implementations that satisfy these abstract interfaces. This allows a specific implementation to be readily interchanged to verify correctness, performance comparisons, or the development of new implementations without the risk of introducing a regression in a highly-interconnected code base. Testing can be performed on an individual plugin or an assembly of plugins. Testing infrastructure that uses these interfaces can be reused between plugins.

An environment to support the CFD 2030 Vision must add value to existing software tools. Coupling existing tools is insufficient. Existing tools are the product of significant investment over decades including validation and verification (V&V) [46]. This V&V support has the opportunity to significantly accelerate the introduction of new capabilities by reducing the human capital required per newly-verified capability. New development can include Functional Equivalence Testing [47] where there is overlap with existing components. T-infinity will enable rapid V&V to so that the rigor in the CFD can be improved [48, 49]. For example, the complete verification process for the Turbulence Modeling Resource (TMR) [50, 51] could be provided by exposing an interface to a CFD solver. Simulations of validation quality experiments [52] can be automated to evaluate new methods and models. This could help enable V&V as a service, which would lower the bar to using these existing benchmarks and simplify automated continuous verification for numerical simulation [53].

The modularity enabled by use of the Dependency Inversion Principle addresses concerns raised by the Post and Kendall [54] lessons learned from the Accelerated Strategic Computing Initiative (ASCI). They document that successful codes often have life cycles that span decades and experience multiple paradigm shifts in hardware, memory structure, operating systems, and computer science. The abstract interfaces described herein support the transition of life cycle phases by allowing these codes to be decoupled into plugins that can be maintained independently [55]. These decoupled plugins result in a more sustainable development environment by reducing the interactions that must be maintained.

Acknowledgments

The authors would like to thank Dr. Pieter Buning and Dr. William Wood of NASA Langley Research Center for providing valuable feedback during the preparation of this manuscript.

This work was supported by the Transformational Tools and Technologies (TTT) project of the NASA Transformative Aeronautics Concepts Program (TACP); the Revolutionary Vertical Lift Technology (RVLT), the Hypersonic Technology (HT), and the Advanced Air Transport Technology (AATT) projects of the NASA Advanced Air Vehicles Program (AAVP); the Entry Systems Modeling (ESM) project of the NASA Game Changing Development (GCD) Program; and the Air Force Research Laboratory, Wright-Patterson Air Force Base.

References

[1] 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 CR-2014-218178, Langley Research Center, Mar. 2014. doi:2060/20140003093.

[2] Keyes, D. E., McInnes, L. C., Woodward, C., Gropp, W., Myra, E., Pernice, M., Bell, J., Brown, J., Clo, A., Connors, J., Constantinescu, E., Estep, D., Evans, K., Farhat, C., Hakim, A., Hammond, G., Hansen, G., Hill, J., Isaac, T., Jiao, X., Jordan, K., Kaushik, D., Kaxiras, E., Koniges, A., Lee, K., Lott, A., Lu, Q., Magerlein, J., Maxwell, R., McCourt, M., Mehl, M., Pawlowski, R., Randles, A. P., Reynolds, D., Riviére, B., Rüde, U., Scheibe, T., Shadid, J., Sheehan, B., Shephard, M., Siegel, A., Smith, B., Tang, X., Wilson, C., and Wohlmuth, B., "Multiphysics Simulations: Challenges and Opportunities," *The International Journal of High Performance Computing Applications*, Vol. 27, No. 1, 2013, pp. 4–83. doi:10.1177/1094342012468181.

[3] 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.

[4] Wissink, A. M., Sitaraman, J., Sankaran, V., Mavriplis, D. J., and Pulliam, T. H., "A Multi-Code Python-Based Infrastructure for Overset CFD with Adaptive Cartesian Grids," AIAA Paper 2008–927, 2008.

[5] Jasak, H., Jemcov, A., and Ťuković, Z., "OpenFOAM: A C++ Library for Complex Physics Simulations," *In Proceedings of the International Workshop on Coupled Methods in Numerical Dynamics*, 2007.

[6] Hoffman, J., Jansson, J., Vilela de Abreu, R., Degirmenci, N. C., Jansson, N., Müller, K., Nazarov, M., and Spühler, J. H., "Unicorn: Parallel Adaptive Finite Element Simulation of Turbulent Flow and Fluid–Structure Interaction for Deforming Domains and Complex Geometry," *Computers and Fluids*, Vol. 80, 2013, pp. 310–319. doi:10.1016/j.compfluid.2012.02.003.

[7] Kiris, C. C., Housman, J. A., Barad, M. F., Sozer, E., Brehm, C., and Moini-Yekta, S., "Computational Framework for Launch, Ascent, and Vehicle Aerodynamics (LAVA)," *Aerospace Science and Technology*, Vol. 55, 2016, pp. 189–219. doi:10.1016/j.ast.2016.05.008.

[8] Economon, T. D., Palacios, F., Copeland, S. R., Lukaczyk, T. W., and Alonso, J. J., "SU2: An Open-Source Suite for Multiphysics Simulation and Design," *AIAA Journal*, Vol. 45, No. 3, 2016, pp. 828–846. doi:10.2514/1.J053813.

[9] Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S., "An Overview of the Trilinos Project," *ACM Transactions on Mathematical Software (TOMS)*, Vol. 31, No. 3, 2005, pp. 397–423. doi:10.1145/1089014.1089021.

[10] Heroux, M., Bartlett, R., Howle, V., Hoekstra, R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and Williams, A., "An Overview of Trilinos," Tech. Rep. SAND2003-2927, Sandia National Laboratories, Aug. 2003.

[11] Heroux, M. A., and Willenbring, J. M., "A New Overview of The Trilinos Project," *Scientific Programming*, Vol. 20, No. 2, 2012, pp. 83–88. doi:10.3233/SPR-2012-0355.

[12] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H., "PETSc Users Manual," Tech. Rep. ANL-95/11 - Revision 3.8, Argonne National Laboratory, 2017.

[13] Martin, R. C., "The Dependency Inversion Principle," *C++ Report*, Vol. 8, No. 6, 1996, pp. 61–66.

[14] Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices*, Pearson, 2003.

[15] Foote, B., and Yoder, J. W., "Big Ball of Mud," *Fourth Conference on Patterns Languages of Programs (PLoP '97)*, 1997.

[16] Hill, C., DeLuca, C., Balaji, Suarez, M., and Silva, A. D., "The Architecture of the Earth System Modeling Framework," *Computing in Science Engineering*, Vol. 6, No. 1, 2004, pp. 18–28. doi:10.1109/MCISE.2004.1255817.

[17] Jones, W. T., "An Open Framework for Unstructured Grid Generation," AIAA Paper 2002–3192, 2002.

[18] Jones, W. T., "GridEx – An Integrated Grid Generation Package for CFD," AIAA Paper 2003–4129, 2003.

[19] Steinbrenner, J. P., Michal, T. R., and Abelanet, J. P., "An Industry Specification for Mesh Generation Software," AIAA Paper 2005–5239, 2005.

[20] Gopalsamy, S., Michal, T. R., and Shih, A. M., “Grid Generation in the Framework of the Unstructured Grid Consortium API using the Geometry and Grid Toolkit GGTK,” AIAA Paper 2006–528, 2006.

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

[22] Kiviaho, J. F., Jacobson, K., Smith, M. J., and Kennedy, G., “A Robust and Flexible Coupling Framework for Aeroelastic Analysis and Optimization,” AIAA Paper 2017–4144, 2017.

[23] Jacobson, K., Kiviaho, J. F., Smith, M. J., and Kennedy, G., “An Aeroelastic Coupling Framework for Time-accurate Analysis and Optimization,” AIAA Paper 2018–100, 2018.

[24] Liskov, B., “Keynote Address - Data Abstraction and Hierarchy,” *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, ACM, New York, NY, USA, 1987, pp. 17–34. doi:10.1145/62138.62141.

[25] Schloegel, K., Karypis, G., and Kumar, V., “A Unified Algorithm for Load-balancing Adaptive Scientific Simulations,” 2000.

[26] Vassberg, J. C., DeHaan, M. A., Rivers, S. M., and Wahls, R. A., “Development of a Common Research Model for Applied CFD Validation Studies,” AIAA Paper 2008–6919, 2008.

[27] Biedron, R. T., Carlson, J.-R., Derlaga, J. M., Gnoffo, P. A., Hammond, D. P., Jones, W. T., Kleb, B., Lee-Rausch, E. M., Nielsen, E. J., Park, M. A., Rumsey, C. L., Thomas, J. L., and Wood, W. A., “FUN3D Manual: 13.3,” NASA TM-2018-219808, Langley Research Center, Feb. 2018.

[28] Schroeder, W., Martin, K., and Lorensen, B., *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 4th ed., Pearson Education, Inc., 2006.

[29] Park, M. A., Barral, N., Ibanez, D., Kamenetskiy, D. S., Krakos, J., Michal, T., and Loseille, A., “Unstructured Grid Adaptation and Solver Technology for Turbulent Flows,” AIAA Paper 2018–1103, 2018.

[30] Allmaras, S. R., Johnson, F. T., and Spalart, P. R., “Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model,” *Seventh International Conference on Computational Fluid Dynamics (ICCFD7)*, 2012.

[31] Loseille, A., and Alauzet, F., “Continuous Mesh Framework Part I: Well-Posed Continuous Interpolation Error,” *SIAM Journal on Numerical Analysis*, Vol. 49, No. 1, 2011, pp. 38–60. doi:10.1137/090754078.

[32] Haimes, R., and Dannenhoffer, J. F., III, “EGADSlite: A Lightweight Geometry Kernel for HPC,” AIAA Paper 2018–1401, 2018.

[33] Alauzet, F., and Loseille, A., “High-Order Sonic Boom Modeling Based on Adaptive Methods,” *Journal of Computational Physics*, Vol. 229, No. 3, 2010, pp. 561–593. doi:10.1016/j.jcp.2009.09.020.

[34] Park, M. A., Loseille, A., Krakos, J. A., and Michal, T., “Comparing Anisotropic Output-Based Grid Adaptation Methods by Decomposition,” AIAA Paper 2015–2292, 2015.

[35] Smith, M. J., Hodges, D. H., and S. Cesnik, C. E., “Evaluation of Computational Algorithms Suitable for Fluid-Structure Interactions,” *Journal of Aircraft*, Vol. 37, No. 2, 2000, pp. 282–294. doi:10.2514/2.2592.

[36] Irons, B. M., and Tuck, R. C., “A Version of the Aitken Accelerator for Computer Iteration,” *International Journal for Numerical Methods in Engineering*, Vol. 1, No. 3, 1969, pp. 275–277. doi:10.1002/nme.1620010306.

[37] Kenway, G., Kennedy, G., and Martins, J. R. R. A., “Aerostructural Optimization of the Common Research Model Configuration,” AIAA Paper 2014–3274, 2014.

[38] Kennedy, G. J., and Martins, J. R. R. A., “A Parallel Finite-Element Framework for Large-Scale Gradient-based Design Optimization of High-Performance Structures,” *Finite Elements in Analysis and Design*, Vol. 87, 2014, pp. 56–73. doi:10.1016/j.finel.2014.04.011.

[39] Bathe, K.-J., and Dvorkin, E. N., “A Formulation of General Shell Elements——the use of Mixed Interpolation of Tensorial Components,” *International Journal for Numerical Methods in Engineering*, Vol. 22, No. 3, 1986, pp. 697–722. doi:10.1002/nme.1620220312.

[40] Harder, R. L., and Desmarais, R. N., “Interpolation using Surface Splines,” *Journal of Aircraft*, Vol. 9, No. 2, 1972, pp. 189–191. doi:10.2514/3.44330.

[41] Kiviaho, J. F., and Kennedy, G. J., “An Efficient and Robust Load and Displacement Transfer Scheme based on Weighted Least-Squares,” AIAA Journal, American Institute of Aeronautics and Astronautics, Reston, VA (submitted for publication).

[42] Biedron, R. T., and Thomas, J. L., “Recent Enhancements to the FUN3D Flow Solver for Moving-Mesh Applications,” AIAA Paper 2009–1360, 2009.

[43] Park, M. A., Krakos, J. A., Michal, T., Loseille, A., and Alonso, J. J., “Unstructured Grid Adaptation: Status, Potential Impacts, and Recommended Investments Toward CFD Vision 2030,” AIAA Paper 2016–3323, 2016.

[44] Shenoy, R., Smith, M. J., and Park, M. A., “Unstructured Overset Mesh Adaptation with Turbulence Modeling for Unsteady Aerodynamic Interactions,” *AIAA Journal of Aircraft*, Vol. 51, No. 1, 2014, pp. 161–174. doi:10.2514/1.C032195.

[45] 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. doi:2060/20020060131.

[46] Oberkampf, W. L., and Roy, C. J., *Verification and Validation in Scientific Computing*, Cambridge University Press, 2010.

[47] Gnoffo, P. A., Wood, W. A., Kleb, B., Alter, S., Glass, C., Padilla, J., Hammond, D., and White, J., “Functional Equivalence Acceptance Testing of FUN3D for Entry, Descent, and Landing Applications,” AIAA Paper 2013–2558, 2013.

[48] Kleb, W. L., and Wood, W. A., “CFD: A Castle in the Sand?” AIAA Paper 2004–2627, 2004.

[49] Kleb, W. L., and Wood, W. A., “Computational Simulations and the Scientific Method,” AIAA Paper 2005–4873, 2005.

[50] Rumsey, C. L., Smith, B. R., and Huang, G. P., “Description of a Website Resource for Turbulence Modeling Verification and Validation,” AIAA Paper 2010–4742, 2010.

[51] Rumsey, C. L., “Recent Developments on the Turbulence Modeling Resource Website,” AIAA Paper 2015–2927, 2015.

[52] Oberkamp, W. L., and Trucano, T. G., “Verification and Validation Benchmarks,” *Nuclear Engineering and Design*, Vol. 238, No. 3, 2008, pp. 716–743. doi:10.1016/j.nucengdes.2007.02.032.

[53] Farrell, P. E., Piggott, M. D., Gorman, G. J., Ham, D. A., Wilson, C. R., and Bond, T. M., “Automated Continuous Verification for Numerical Simulation,” *Geoscientific Model Development*, Vol. 4, No. 2, 2011, pp. 435–449. doi:10.5194/gmd-4-435-2011.

[54] Post, D. E., and Kendall, R. P., “Software Project Management and Quality Engineering Practices for Complex, Coupled Multiphysics, Massively Parallel Computational Simulations: Lessons Learned From ASCI,” *International Journal of High Performance Computing Applications*, Vol. 18, No. 4, 2004, pp. 399–416. doi:10.1177/1094342004048534.

[55] Willenbring, J. M., Heroux, M. A., and Heaphy, R. T., “The Trilinos Software Lifecycle Model,” *Third International Workshop on Software Engineering for High Performance Computing Applications. SE-HPC ’07*, 2007. doi:10.1109/SE-HPC.2007.5.