DOE-ROCSTAR-09596

ILLINOIS ROCSTAR

PHASE II FINAL REPORT

Topic 2. Increasing Adoption of HPC Modeling and Simulation in the Advanced Manufacturing and Engineering Industries
Subtopic b. HPC Support Tools and Services
Identification Number DE-SC0009596: Infrastructure for Multiphysics Software Integration in High Performance Computing-Aided Science and Engineering

Michael T. Campbell, SBC Principal Investigator
Development Team:
Masoud Safdari
Jessica E. Kress
Michael J. Anderson
Samantha Horvath
Mark D. Brandyberry
Woohyun Kim
Neil Sarwall
Brian Weisberg

OpenMultiphysics

Illinois Rocstar LLC
1800 South Oak Street, Suite 108
Champaign, IL 61820

Reporting Period: April 14, 2014 through October 15, 2016

This work was funded in whole or in part by DOE Identification Number DE-SC0009596. The U.S. Government has for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license to use, modify, reproduce, release, perform, display, or disclose the work by or on behalf of the U.S. Government.

U.S. Department of Energy
Washington DC

PHASE II FINAL REPORT

Identification Number DE-SC0009596: Infrastructure for Multiphysics Software Integration in High Performance Computing-Aided Science and Engineering

Michael T. Campbell*
Illinois Rocstar LLC
1800 S. Oak, Suite 108
Champaign, IL 61820
`www.illinoisrocstar.com`
*tech@illinoisrocstar.com

Ceren Susut
DOE Program Manager
Office of Science – Germantown
U.S. Department of Energy
Germantown Building
1000 Independence Avenue, S.W.
Washington D.C., 20585-1290

Acknowledgments

Illinois Rocstar LLC would like to thank the Department of Energy and the SBIR Program for supporting and funding this work. We would also like to thank Aerojet-Rocketdyne, AMAX, Boeing, Caterpillar, Kitware, NCSA, Orbital-ATK, Sandia National Laboratory, and the Computational Science and Engineering program at the University of Illinois for their support, participation, and invaluable information.

Contents

List of Figures

List of Tables

1 Executive Summary

The project described in this report constructed and exercised an innovative multiphysics coupling toolkit called the **I**llinois Rocstar **M**ulti**P**hysics **A**pplication **C**oupling **T**oolkit (*IMPACT*). *IMPACT* is an open source, flexible, natively parallel infrastructure for coupling multiple uniphysics simulation codes into multiphysics computational systems. *IMPACT* works with codes written in several high-performance-computing (HPC) programming languages, and is designed from the beginning for HPC multiphysics code development. It is designed to be minimally invasive to the individual physics codes being integrated, and has few requirements on those physics codes for integration. The goal of *IMPACT* is to provide the support needed to enable coupling existing tools together in unique and innovative ways to produce powerful new multiphysics technologies without extensive modification and rewrite of the physics packages being integrated.

There are three major outcomes from this project: 1) construction, testing, application, and open-source release of the *IMPACT* infrastructure, 2) production of example open-source multiphysics tools using *IMPACT*, and 3) identification and engagement of interested organizations in the tools and applications resulting from the project. This last outcome represents the incipient development of a user community and application echosystem being built using *IMPACT*. Multiphysics coupling standardization can only come from organizations working together to define needs and processes that span the space of necessary multiphysics outcomes, which Illinois Rocstar plans to continue driving toward.

The *IMPACT* system, including source code, documentation, and test problems are all now available through the public gitHUB.org system to anyone interested in multiphysics code coupling. Many of the basic documents explaining use and architecture of *IMPACT* are also attached as appendices to this document. Online HTML documentation is available through the gitHUB site. There are over 100 unit tests provided that run through the Illinois Rocstar Application Development (IRAD) lightweight testing infrastructure that is also supplied along with *IMPACT*. The package as a whole provides an excellent base for developing high-quality multiphysics applications using modern software development practices.

To facilitate understanding how to utilize *IMPACT* effectively, two multiphysics systems have been developed and are available open-source through gitHUB. The simpler of the two systems, named *ElmerFoamFSI* in the repository, is a multiphysics, fluid-structure-interaction (FSI) coupling of the solid mechanics package *Elmer* with a fluid dynamics module from *OpenFOAM*. This coupling illustrates how to combine software packages that are unrelated by either author or architecture and combine them into a robust, parallel multiphysics system.

A more complex multiphysics tool is the Illinois Rocstar *Rocstar Multiphysics* code that was rebuilt during the project around *IMPACT*. *Rocstar Multiphysics* was already an HPC multiphysics tool, but now that it has been rearchitected around *IMPACT*, it can be readily expanded to capture new and different physics in the future. In fact, during this project, the *Elmer* and *OpenFOAM* tools were also coupled into *Rocstar Multiphysics* and demonstrated. The full *Rocstar Multiphysics* codebase is also available on gitHUB, and licensed for any organization to use as they wish.

Finally, the new *IMPACT* product is already being used in several multiphysics code coupling projects for the Air Force, NASA and the Missile Defense Agency, and initial work on expansion of the *IMPACT*-enabled *Rocstar Multiphysics* has begun in support of a commercial company. These initiatives promise to expand the interest and reach of *IMPACT* and *Rocstar Multiphysics*, ultimately leading to the envisioned standardization and consortium of users that was one of the goals of this project.

2 Introduction

2.1 Significance

IllinoisRocstar Multiphysics Application Coupling Toolkit (*IMPACT*)

- Lower barriers to entry into multiphysics modeling and simulation
- Decrease multiphysics application development time
- Facilitate sustainable, reusable multiphysics development
- Integrate in-house, open, and commercial tools together
- Decrease product time-to-market by increasing use of multiphysics modeling and simulation

Many of today's important and challenging problems in science and engineering involve multiple complex, interacting physical systems, often incorporating different material states with combustion or other sources of energy release (e.g. see Figure 1). Examples of such systems include fluid-structure interaction (FSI), conjugate heat transfer (CHT), thermo-mechanical coupling (TMC), and shock-to-detonation of energetic materials (SDT). Multiphysics refers to the coupled, advanced modeling techniques used to simulate these interacting systems. Large-scale modeling and simulation of such multiphysics problems using high performance computing (HPC) has become a crucial component of research and development in the private sector, academia, and national laboratories.

Several factors conspire to drive the cost of development and ownership of multiphysics capabilities inordinately high: for example, a private sector entity has limited choices when deciding how to address a multiphysics simulation requirement. One must either invest in expertise for a "build-your-own" solution or purchase external expertise and capabilities. There are currently no industry-accepted standards or protocols for tools and interfaces, commercial or otherwise, that support general software and simulation integration and coupling for multiple parallel simulation applications. As a result, both choices incur costly software development and refactoring activities that often result in a single limited-use capability or project-specific solution.

Commercial off-the-shelf (COTS) solutions are also costly and often difficult to apply diversely in the user's environment. Vendor lock-in drives the costs of development and ownership of multiphysics capabilities to prohibitive levels, particularly for small-to-medium-sized businesses. The difficulty of establishing viable and diverse solutions with COTS-based software is compounded by prohibitively expensive high performance computing (HPC) or multiprocessor licensing models. These difficulties are not confined to the private sector but are shared by industry, academia, and government entities with large-scale multiphysics simulation needs.

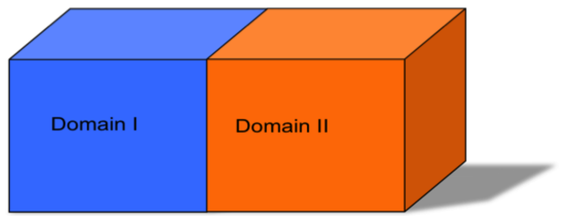
These technological and economic barriers to the rapid production and testing of high fidelity, high performance multiphysics simulation software are a significant bottleneck for research efforts in areas that currently or will soon rely heavily on multiphysics simulation tools [Council on Competitiveness (2011)]. An open, industry-accepted set of standard interfaces and protocols for parallel software integration is a key technology for addressing many of these challenges and its establishment would increase the adoption of HPC-based M&S within the industry sector.

Over the course of this project, Illinois Rocstar has developed the *IllnoisRocstar Multiphysics Application Coupling Toolkit* (*IMPACT*), an open source multiphysics software integration environment that facilitates the integration of multiple parallel software components for the purpose of multiphysics simulation capabilities. *IMPACT* is designed for use by the broader community interested in developing multiphysics capabilities with existing simulation applications.

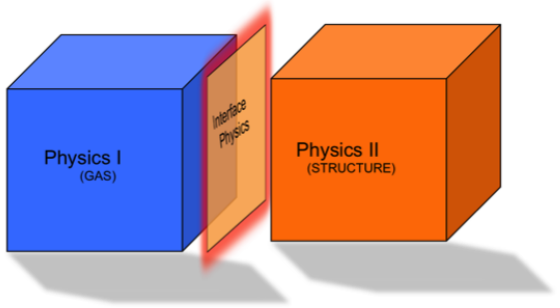
Standardization and specification of software integration interfaces across application, language, and platform boundaries in an HPC environment is necessary for advancing the state-of-the art predictive multiphysics simulation capabilities as the underlying algorithms and HPC platforms evolve. General data-driven interfaces and their adoption by the community comprise a key step in establishing these standards and specifications.

2.2 Public Benefits

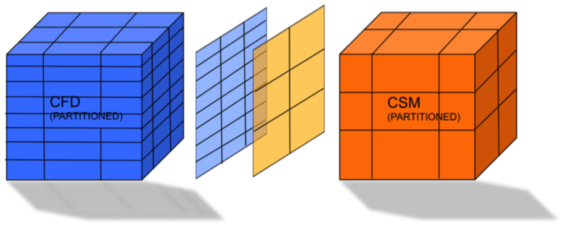
Modern designs must seek to provide advanced interfaces for scientists, engineers, and analysts; these interfaces must hide many of the details of the platforms and software mechanics required to interface multiple components. Such infrastructure will significantly reduce the barrier to entry for private sector entities endeavoring to develop multiphysics capabilities that leverage the nation's HPC resources. To remain competitive in the modern world marketplace, simulation applications must be designed for deployment in integrated environments and coupled model while simulating ever more complex interacting systems.



(a) Two 3D domains abut and interact through a 2D interface surface.



(b) In a broad class of multiphysics problems, the physics of the interacting system can be modeled as separable systems.



(c) The domains are each treated by domain-specific models, and may be disparately discretized.

Figure 1: In partitioned multiphysics, each domain is simulated by a dedicated, domain-specific simulation application and interaction occurs at the intersections of the domains.

It is noted in [Keyes et al. (2013)] that there exists a common need for greater encapsulation of software while allowing for flexible access to data. Another benefit, as discussed in [Slotnick et al. (2013)], is greater sustainability of software development in the future. If a set of standards is established, it will become easier to incorporate new developments into legacy code, which in turn allows the field to progress more steadily instead of continually recreating past work.

Although the use of multiphysics is pervasive in academia and at national laboratories, much of the effort in ground-breaking research is continually duplicated due to the lack of an infrastructure for developing highly inter-operable, reusable software components. The infrastructure developed in this project enables existing high-quality software to be integrated, orchestrated, and operated as components in composite software systems, fostering collaboration throughout the M&S community.

Integration-ready software encapsulating couple-ready models are the future of HPC-based M&S and a key to maintaining national competitiveness.

An open standardized infrastructure would significantly enhance the ability of the private sector entity to leverage HPC-based M&S in its critical mission, and at a fraction of the current cost. The benefits here include increased realism, decreased time-to-solution, increased manufacturing capability, reduced manufacturing cost, and a decrease in the product design cycle.

The efficacy with which composite software systems can be composed and orchestrated on modern, parallel systems to address complex problems in science and engineering can be greatly improved through use of this technology. The economic implications and range of commercial applications are promising. Integration-ready software encapsulating couple-ready models are the future of HPC-based M&S and a key to maintaining national competitiveness.

2.3 A Note about Language

In their final report, the particpants of the Institute for Computing in Science multiphysics workshop [Keyes et al. (2013)] stress the importance of precise language when discussing multiphysics and its challenges. This notion resonates strongly with the Illinois Rocstar development team and is important for understanding the subject matter of this work. Keyes makes the crucial distinctions between "strong" and "weak" coupling of multiphysics models. He also distinguishes between "loose" and "tight" algorithms used to actuate the coupling. For the purpose of this work, we introduce some further distinctions. When referring to real, physical systems, our team uses the word "interact." When multiple aspects of a physical system (or systems) interact, multiphysics methods may be needed to model them. We reserve the word "couple" for referring to mathematical models and numerical algorithms used in modeling interacting systems. We reserve "integration" for referring to the software systems that implement these models. Software integration is distinct from numerical coupling and refers to the mechanics of interfacing and operating multiple software components in concert toward a common goal. Software integration is the main subject matter of this project.

2.4 Technical Objectives

The high level technical objective of this project is to design and implement an integration infrastructure which will support the development of multiphysics capabilities. Our approach has been one in which we separate the multiphysics capabilities from the underlying software integration constructs, then implement the multiphysics-specific capabilities on top of the general software integration infrastructure. In rough order of execution the technical goals of the project are:

1. **Design and implement a software integration infrastructure**
   The software integration infrastructure will provide the primitive constructs and capabilities required for the integration of multiple parallel applications into a composite software system to facilitate multiphysics simulation.

2. **Design and implement multiphysics simulation capabilities and services**
   Using the interface, constructs, and infrastructure as designed by this project, implement multiphysics-specific capabilities and services with the ultimate end-goal of implementation of an example multiphysics simulation capability with services and solvers required by the example system simulations.

2.5 Community Goals

In addition to the specific technical goals of this project above, Illinois Rocstar also seeks to spur a community discussion and development of an open standard for methods and protocols for general parallel application coupling. This goal is as important as the specific technical goals and far more difficult. To realize this standardization goal, a representative community must first be established.

The Illnois Rocstar development team has learned through execution of the project that building a community for establishing standards and to promote the open source product by "pulling" users is difficult. Originally, the project included the following community-oriented goals:

1. **Distribute the software and infrastructure with open/free licensing**
   To ensure that the infrastructure can be used in any environment without licensing issues, it must be made available under a free *do-anything*, non-copyleft license. Users of this infrastructure must not be required to expose internal developments, reciprocate, or purchase anything of any kind. Users will be free to redistribute the infrastructure in their own derivative products.

2. **Identify a collection of multiphysics example and V&V problems**
   These problems are used to demonstrate the feasibility and functionality of a working simulation application developed during this project. In addition, it is envisioned that they may also be used in capability-driven efforts to design and develop enhanced capabilities.

3. **Build a community of stakeholders to guide the design of the infrastructure**
   A community of scientists, engineers, analysts, and developers from industry, academia, and government is required to ensure that the developments of this project are useful and relevant for all involved.

As we have learned over the course of this project, some of the specific design and implementation goals presented here conflict with the broader goal of standards initiation. We desire to spur the development of a standard interface and protocols for general parallel application integration, yet this project presents a specific design and a multiphysics infrastructure implementation that does not build to an industry-accepted standard or set of protocols. The development of standards and protocols requires a community effort and has proven to be beyond the scope of this project; instead we present both a brief characterization of a general infrastructure and our specific infrastructure implementation. We envision this effort as a call for standardization, as well as a lead-on to the standardization effort, wherein our infrastructure implementation can be refactored for standard compliance.

2.6 Approach

During the initial phase of the project, the thrust of the community involvement track involved design and production of a community involvement website (`http://openmultiphysics.org`), and word-of-mouth discussion during personal and technical meetings. Contacts with a number of large companies and universities were made, but while initial interest was there, maintaining interest during product development, when there was not a testable, runnable product available became difficult. We discovered, on this and other current open-source projects that there needs to be at least a basic product in place before external interest can be generated.

Given the issues with generating community involvement discussed above, the focus of the project returned to example programs, documentation, and expanded use. As will be discussed in this report, two open source examples of use of the *IMPACT* infrastructure have been constructed and released during this project. One, the *ElmerFoamFSI* fluid-structure-interaction package illustrated coupling two non-Illinois Rocstar tools that had no common heritage, and had not been designed to be coupled. The second rebuilt the internal Illinois Rocstar *Rocstar Simulation Suite* package around *IMPACT* to produce the new *Rocstar Multiphysics* package. In addition, both Elmer and the OpenFoam module were easily coupled into *Rocstar Multiphysics* to enhance it's capabilities. Further discussion of the community involvement aspects of the project are discussed in Section 5.

2.7 Products

The project products are open, freely available multiphysics simulation applications capable of simulating the multiphysics V&V cases. In addition, the infrastructure is applied to a series of representative problems to pave the road for the user community adoption. In the subsequent sections of this report the products developed are described in detail.

2.8 Environment

This is Illinois Rocstar's first open source project. Changes to our development environment were required to effectively accomplish the project goals as we had envisioned them. Early in the project, we spoke with developers of the computational science and engineering software packages *Trilinos* and *LIME*, who highly recommended that we adopt and adapt the *TriBITS* lifecycle model for this project. It wasn't until later in the project that we fully appreciated the extent of *TriBITS* relevance.

The process by which a piece of CSE software begins life as a set of research requirements and then matures into a trusted high-quality capability is both commonplace and extremely challenging. [Bartlett et al. (2012)]

To best achieve our project goals and to establish a sustainable development infrastructure for stakeholder-driven design, we have adopted and adapted a Lean/Agile approach [Poppendieck and Poppendieck (2003)] to the development and project management.

2.8.1 Scrum

We have adapted a Scrum-like approach to our environment, implementing the following practices:

- Daily stand-up
- Two-week development iterations
- Project backlogs
- Team planning
- Demo/Review/Retro ceremonies
- Distributed "project owner" (one for each project)
- Fixed scrum master

2.8.2 Automated Build and Testing

We have developed and implemented an automated build-and-test framework that provides developers with several constructs for adding and executing unit and integration tests. The testing framework supports continuous, hourly, and nightly tests, submitting all results to a project dashboard. The framework leverages Kitware's *CMake* and *CDash*, and adds our own original innovative code that supports building and testing on multiple remote platforms with asynchronous mechanisms for running parallel tests through batch systems.

2.8.3 Customer Team

Our customer team is an internal team, the purpose of which is to act as the customer. The customer team tests, attempts to break, and otherwise helps developers harden the iteration products. Since we have many projects and our team works across these projects, the customer team roles change from task to task. To ensure each feature is tested appropriately, the customers must consist of those staff members who were not involved in the development of the tested feature.

2.8.4 Test-driven Development

In this project we have developed a test-driven development (TDD) process. We have set up a robust framework to support TDD and are gaining experience with it as we execute our projects. We have been pushing ourselves to discuss how each existing (and developing) feature should be tested.

2.8.5 *TriBITS* Lifecycle

TriBITS lifecycle model is adopted for this project. This task is somewhat challenging, since it involves refactoring a legacy code of an unspecified maturity level. These aspects are notoriously difficult to effectively grandfather into the system. In this project we began by addressing the key components of sustainability for the project product: *open source*, *core domain distillation documentation/design*; *exceptionally well-tested*, *clean code*; and *minimization of dependencies*. See [Bartlett et al. (2012)] for more information on the *TriBITS* lifecycle model.

2.9 Infrastructure Development

Our planned approach to simply refactor the legacy *Rocstar* application was not well aligned with the technical development practices of the current project. Grandfathering this existing code into any formal Lean/Agile engineering code development and lifecycle model is challenging by itself. Parallel to our effort to retrofit the legacy code with good software engineering practices, the infrastructure development effort proceeded with the following approach and activities:

- **Survey existing/modern frameworks :** During the period of performance, we studied other frameworks in more detail. This examination proved essential in guiding our code development as well as our commercialization effort. In particular, we strove to understand the most relevant and modern coupling packages, such as *LIME*, *OpenFSI*, and *PreCICE*.
- **Planning and design :** The ultimate goal of this project is to support a stakeholder-driven design with closed development. That is, we want the design to be driven by our stakeholder community, and the development and implementation to be performed completely by Illinois Rocstar. As per our Lean/Agile approach, we conducted weekly technical planning meetings wherein each development iteration was planned and *Customer Teams* acted as surrogates for the stakeholder community.
- **Refactoring :** The initial phase of the project was mostly refactoring activities. Several aspects of the legacy *Rocstar* infrastructure were not amenable to the targeted distribution strategy for the final infrastructure product. Besides refactoring for abstraction model correspondence, we factored out incompatibly-licensed code (e.g., GNU Public License (GPL) and other proprietary or copyleft-licensed code), Illinois Rocstar intellectual property, and any otherwise experimental code that is not yet ready for release to the general public.

3 *IMPACT*

3.1 Overview

The *Illnoisrocstar Multiphysics Application Coupling Toolkit* (*IMPACT*) is a general infrastructure for software integration in support of multiphysics simulation capabilities. The overarching goals of this software are to: reduce the cost of development and ownership of multiphysics capabilities; to lower the technical and economic barriers of entry for HPC-based Modeling and Simulation (M&S) in advanced manufacturing and engineering industries; and to initiate a standardization effort for parallel software integration in the HPC environment.

This software leverages experience gained over 15 years of development, ownership, and operation of a massively parallel multiphysics simulation application, *Rocstar*, initially developed under the DOE ASCI program at the University of Illinois Center for Simulation of Advanced Rockets (CSAR). Informed by our community of stakeholders, which includes other companies with multiphysics needs, Illinois Rocstar has extracted *Rocstar*'s underlying coupling framework and refactored it for general use in developing multiphysics capabilities.

IMPACT is the result of extracting and generalizing the *Rocstar* multiphysics software integration infrastructure for use by the broader community interested in developing multiphysics capabilities with existing simulation applications. We envisioned that this general open multiphysics infrastructure would serve as a reference implementation to help guide the development of a new open standard for highly inter-operable multiphysics software design and execution. Our motivating problem domain is illustrated in Figure 1, where multiple domains interact across a moving, reacting interface. The resulting work, however, is not limited to this particular situation and, moreover, transcends multiphysics altogether.

3.2 Design Philosophy

Part of our goal for this project was to spur an inter-organization discussion of standards and protocols for general parallel software integration. To encourage this, we build an abstraction that separates out capabilities and data constructs that are domain-independent (i.e., independent of multiphysics and M&S). Our efforts to produce such an abstraction are presented in this section. The motivations for this abstraction hierarchy are:

- software reuse,
- maximizing interoperability among disparate software components,
- reducing complexity in developing integrated software systems,
- facilitating more rapid prototyping,
- providing encapsulation,
- heterogeneity,
- and to launch a platform for standards development.

A significant portion of the effort required in building integrated composite software systems lies in infrastructure development. Common architectures include those in which the various constituent components are centrally orchestrated or end-to-end workflows with no centralized orchestration. The integration of multiple executables, one of which may have "ownership" of the control flow, is often required. These architectures sometimes involve physical networks between the various components of the integrated system. This situation usually calls for physical network-based communication and often involves complicated, event-driven control flow management.

In the current work, we have attempted to design a software integration infrastructure that generalizes the common features of these architectures, and encapsulates them into a separate toolkit for use by the development community in software integration endeavors. Further, as we describe in later sections, we have attempted to identify a minimal set of domain-independent multiphysics capabilities and factor them onto the infrastructure as "services," such that the services, together with the software integration toolkit, form a general infrastructure for multiphysics capability development. A high-level depiction of the general infrastructure layers is shown in Figure 2. The bottom layer represents a software integration toolkit which provides the basic constructs allowing applications to *publish* native data structures and functions. In this context, by publish, we mean that the application can describe, and provide access to native data structures and functions by outside software. Each layer is discussed in more detail in the following sections.

3.3 Abstraction

The overarching technical objective of the project was to design and implement a general infrastructure for software integration in support of the development of multiphysics capabilities. Our approach was to form an abstract model wherein all multiphysics-specific capabilities were factored out into layers with increasing domain specificity. The abstraction is shown in Figure 2. This model forms a framework around which industry-wide standards may be discussed and developed.

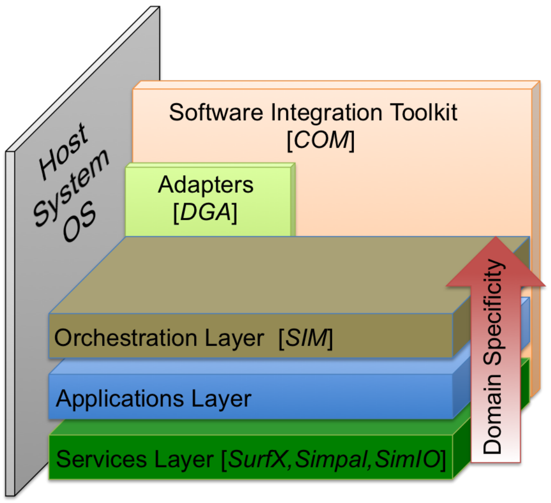

Host System OS
Software Integration Toolkit [COM]
Adapters [DGA]
Orchestration Layer [SIM]
Applications Layer
Services Layer [SurfX,Simpal,SimIO]
Domain Specificity

Figure 2: Abstraction model for software integration infrastructure - Application layer components can use capabilities from the Services layer, interface with other application layer components, and participate as part of a composite software system coordinated by the Orchestration layer. All inter-component exchanges are mediated by the Software Integration Toolkit. Problem and domain specificity increases from bottom to top layers. Software packages implementing components of each layer are shown in square brackets.

The most general layer with no multiphysics-specific constructs is called the Software Integration Toolkit (SIT) Layer. The SIT provides the basic constructs necessary for the integration of multiple software components. Specifically, the SIT provides for i) *application encapsulation*, ii) *native data publication*, iii) *native method publication*[*], and iv) *inter-component communication* (ICC). The SIT layer encapsulates the constructs most suitable for industry-wide standardization.

The layer at the next level of domain-specificity is the Service Layer (green in Figure 2). This layer provides capabilities through the SIT that can be used to implement and perform domain-specific capabilities and tasks. For example, a service to accomplish data mapping between the disparately discretized surfaces in Figure 1 would be a common and useful service in the partitioned multiphysics domain. Services may be implemented as *service applications* or as

[*]In this context, publication refers to the act of making something available to outside, non-native components.

service adapters. Service adapters are those that provide an enhanced capability by mutating or morphing the SIT, while service applications simply use the SIT to provide a capability.

The Applications Layer, shown in blue in Figure 2, encapsulates those applications that provide the "primary" domain-specific capabilities that must be integrated in the composite software system. In a multiphysics system, the single physics domain applications would live in the applications layer.

This integration is done through the SIT by adapting the application source code or wrapping the application executable or interface using an API provided by the SIT implementation. This API provides the constructs and mechanisms necessary for the application to share its data and functions with the integrated software system. Applications that interact with the coupled system only through the infrastructure are considered fully integrated. All others are considered as only partial integrations. Once integrated, whether fully or partially, the user application becomes a *component* of the integrated system. Fully integrated components are also referred to as *modules*.

Application adaptations for integration should be shallow. The SIT provides an API that can understand, perhaps with minor changes, the application-native data structures and interface with application-native methods.

Finally, the Orchestration Layer of the model integration infrastructure (shown in olive green in Figure 2) encapsulates the driver for the integrated composite software system. The orchestration layer is the most domain-specific piece of the integrated capability; often, but not always, this piece is implemented as an orchestrating driver, or *orchestrator*. Other possibilities for orchestration layer constructs include "passive" or event-driven middleware components.

3.4 Software Integration Layer

At the core of the infrastructure lies an abstraction layer for general software integration in the HPC environment. This layer, the Software Integration Toolkit (SIT), defines the primitive software integration constructs that facilitate the sharing of application-native data and methods across the component-component boundary. As will be explained in the remainder of this section, it is designed to support several different composite software system architectures and types of user applications. We identify the following constructs and capabilities as those under purview of the SIT:

Application Encapsulation

The SIT must provide an abstraction that encapsulates or represents an application and its interface to the composite system. All of the applications' interactions with the composite software system are conducted through this abstraction. This abstraction must support serial and parallel applications, which may use a variety of parallelization strategies (e.g., OpenMP[*], MPI[†], or proprietary[‡]). Support should be provided for integrating both library applications as well as applications that must run as stand-alone executables (i.e., users should be able to integrate both stand-alone applications and libraries).

Native Method Publication

The SIT must provide abstractions and constructs for publishing application-native methods (i.e., functions) and associated metadata. Since applications integrated into the composite system will typically be coded in a variety of programming languages, allowing coexistence of multiple programming

[*] `http://openmp.org/wp/`

[†] `https://www.open-mpi.org/` and `https://www.mpich.org/`

[‡] e.g. `https://software.intel.com/en-us/intel-mpi-library`

languages is necessary. This piece of the SIT must also include the mechanisms required to invoke application-native methods across the programming language and component boundaries.

Native Data Publication
To support the vast disparity in programming languages and data structures among composite system components, a unified, language-independent, abstract view of application-native data is necessary to standardize intermodule exchanges. This abstraction must be flexible and self-describing. General abstractions, constructs, and mechanisms for publishing application-native data and associated metadata must be provided by the SIT. This includes mechanisms for specifying access policies on data (e.g., read-only, read/write, etc.) and layout of the data in memory (e.g., see Figure 9).

Inter-Component Communication
An interface must be provided to hide the mechanics and complexity of the communication between components of the integrated software system. This Inter-Component Communication (ICC) interface allows for the greatest variety in the architecture for the applications and the composite software system. For example, to support architectures in which some component must run as a stand-alone executable, some form of interprocess communication (IPC) must be used to actuate communication between the stand-alone component and the rest of the system.

3.5 Services

In our proposed software integration architecture, services are specific capabilities provided through the constructs of the SIT that support domain-specific integrated software systems. Examples include computational and numerical services such as parallel I/O, mesh smoothing, and surface propagation. In our tear-down of the *Rocstar* simulation application, we have identified two types of services: *adapters* and *applications*.

Service Adapters
Service adapters are non-application services that mutate parts of the SIT toward a particular domain or application. These types of services provide additional or specialized abstractions to be used in the higher layers of the architecture (i.e., in Service Applications, User Applications, or Orchestration). Mesh-aware data structures and data structures with built-in parallelism are two examples of service adapters.

Service Applications
Service applications interact with the integrated software system in very much the same way that user applications do (i.e., through the SIT). In general there is little distinction between user applications and service applications, but greater distinction comes with domain-specific applications. The general distinction is that service applications typically do not have their own data and, instead, operate on data owned by other components. For example, multiphysics simulation services might include such capabilities as mesh smoothing, surface propagation, data mapping, and remeshing.

3.6 User Applications

User applications are those that provide the "primary" domain-specific capabilities that must be integrated into the composite software system. In a multiphysics case, two example user applications could be an application implementing a computational fluid dynamic (CFD) model with another implementing a computational solid mechanics (CSM) or transient thermal model. In this case, the target simulation for the integrated software system could include fluid solid interaction (FSI) or congjugate heat transfer (CHT).

User applications are integrated through the SIT by adapting the application source code or wrapping the application executable or interface using an API provided by the SIT implementation. This API provides the constructs and mechanisms necessary for the application to share its data and functions with the integrated software system. Applications that interact with the coupled system only through the infrastructure are considered fully integrated; all others are considered as only partial integrations. Once integrated, whether fully or partially, the user application becomes a *component* of the integrated system.

As mentioned earlier, application adaptations for integration should be shallow. The SIT should provide an API that can understand, in some cases with modifications, the application-native data structures and interface with application-native methods. The integration adaptations are typically in the application's native programming language and live outside the main source code for the application. This avoids the exposure of any potentially proprietary information to other components of the integrated software system.

3.7 Orchestration

The orchestration layer of the model integration infrastructure is the piece that implements the integrated composite software system. Often, but not always, this is implemented as an orchestrating driver, or *orchestrator*. Other possibilities for orchestration layer constructs include "passive" or event-driven middleware components. Regardless of implementation, orchestration layer constructs have access to one or more applications through the SIT and use intrinsic functionalities, or those provided by services, to actuate the interactions between system components.

In multiphysics systems, orchestrators typically manage the control flow and implement the coupled timestepping schemes. For example, *Rocstar*'s orchestrator manages control flow, implements the coupled timestepping, and actuates the data transfers between physical domain-specific components, including implementation of the jump conditions.

3.8 Multiphysics Infrastructure

Project goals include the implementation of a multiphysics infrastructure based on abstraction, discussed in Section 3.3. The main crux and focus of this project has been to develop the SIT abstraction and implement its functionalities. Faithful representation of the general design abstraction has been somewhat challenging since the implementation and abstraction have co-evolved with our understanding of the requirements of our various collaborators and potential customers during the course of the project. We have implemented a working infrastructure; however, it does not exactly follow the general abstraction model.

Illinois Rocstar's design for the infrastructure is largely based on the architecture of its multiphysics simulation application, *Rocstar* (e.g., see Figure 11). In *Rocstar*, multiple application "modules" share functions and data through an interface component with a centrally orchestrated control flow. As will be described, our adaptations to adhere more closely with the abstract design presented in Section 3.3 have resulted in increased utility and support of more diverse architectures.

We use an object-oriented approach to infrastructure implementation. Our infrastructure is implemented in C++ and supports software components written in C, C++, and Fortran 90. Flexible and clean interoperability between these languages is emphasized.

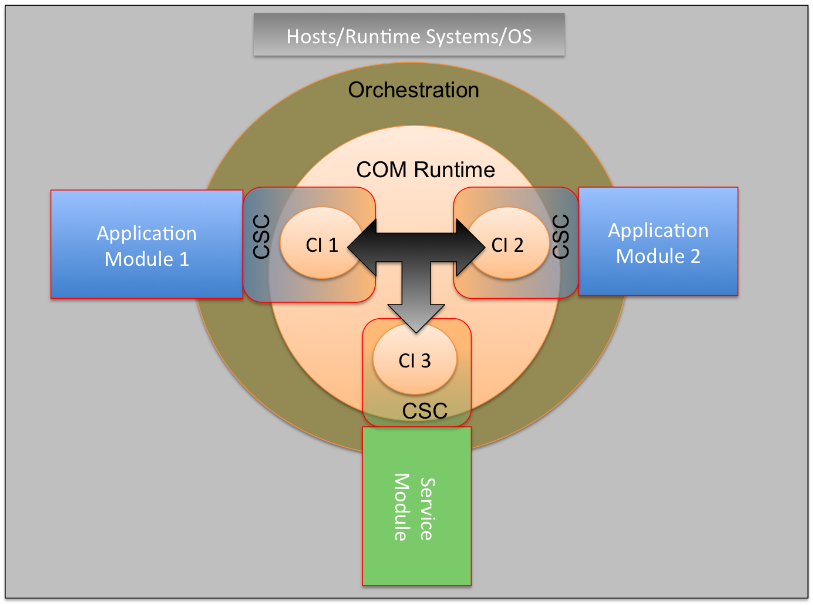


Figure 3: *COM* architecture with multiple software *modules*, wherein no ICC layer is needed to communicate. Each module is loaded at runtime and shares the same process with the orchestrator.

3.8.1 Multiphysics Services

We identified a minimal set of problem domain-independent multiphysics capabilities and factored them onto the infrastructure as "services," such that the services, together with the software integration toolkit, form a general infrastructure for multiphysics capability development. The service packages are indicated in Figure 2. Support for discrete geometries (i.e. meshes) were built-in using a service adapter, *Discrete Geometry Adapter* (*DGA*). Surface-surface data mapping is provided by the *SurfX* service application, and disk I/O services are provided by *SimIO*.

3.8.2 Multiphysics Orchestrator

The *Simulation Integration Manager* (*SIM*) is our orchestration package. It has the mechanisms, constructs, and API required for managing multiple applications, their data, and control flow between component objects. *SIM*, and its constructs are discussed in more detail in Section 3.9.1. An overview of the full, integrated system is illustrated in Figure 7.

3.9 Infrastructure Implementation

Implementation of a multiphysics infrastructure is based on the abstraction proceeded with an implementation of the SIT layer, a set of orchestration layer constructs, and several services. These software packages fit into the abstraction as shown in Figure 2 in square brackets, e.g. [*COM*] and [*DGA*]. Each implemented software package is briefly discussed in the subsequent sections.

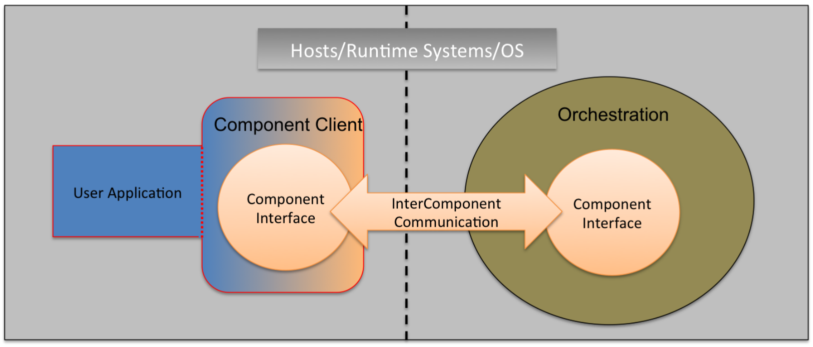


Figure 4: *COM*'s Component Interface encapsulates the user application and contains inter-component communication (ICC) endpoints, which hide the details of data and function call transport across any physical network that may separate the user application process(es) from that of the Orchestration component.

3.9.1 SIT

Our implementation of the SIT layer is embodied in the *Component Object Manager* (*COM*). The *COM* package provides several constructs for encapsulation of applications, their published interfaces, and data. The main high-level constructs of *COM* can be seen in Figure 4, with an example of an integrated application in Figure 7. One of the components of *COM* is the *Component-side Client* (CSC), an application-native construct that uses the *COM* API* to produce one or more *Component Interface* (CI) objects. CI objects encapsulate the application's published functions and data, and provide the interface through which external components can access the application; the ICC thus provides communication between CIs. Application-native data are encapsulated by *DataItem*s and *DataGroup*s, which are not shown in Figures.

Component Interface Object (CI) The Component Interace Object, shown in Figure 5, is the main abstraction for components of the integrated software system. Specific application-native data and methods are encapsulated in distributed, uniquely-named Component Interface Objects through which all inter-component interactions are mediated. A component constructs a CI at runtime and populates it with application-native data and functions. Components reference a CI, their contained data, and functions using their unique names, which are of character-string type. CI names must be unique across all modules. Components may access CI-encapsulated data and functions directly or by obtaining integer handles from the *COM* runtime system. Accessing functions and data through handles provides transparent language interoperability while maintaining the object-oriented paradigm. For architectures with components separated by physical networks, ICC endpoints can be used to synchronize control flow and data across the network.

Component-Side Client (CSC) The Component-side client is the software component responsible for creating a CI and "registering" an application's native data and functions. This client is the software construct that must be written to integrate an external software application with the composite system. The CSC is not an infrastructure piece, per se, but uses the API provided by the infrastructure to implement an application-specific construct that allows the application to publish its native data and functions. CSC's are almost always written in the application's native language and are built as an intrinsic piece of the application for which they are developed. In some cases (e.g., when the source code for the application is not available) the CSC can be a wrapper construct for a stand-alone executable.

*Application programming interface.

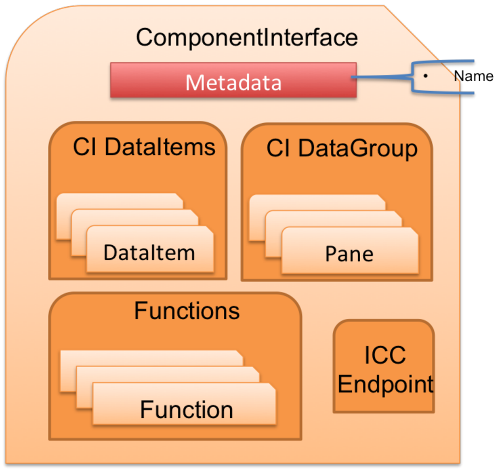


Figure 5: The *COM* Component Interface Window provides access to application-native data and functions through the ICC interface. Currently, CI supports only one DataGroup, which may have multiple instances.

Component Module Object (CM) In the *COM* infrastructure implementation, an application can be completely encapsulated into a Component Module Object; CM-integrated applications hereafter are called "modules." Modules are designed to be built into a shared object that is dynamically loaded at system runtime. *COM* provides an API and runtime system for loading and managing modules at runtime. Modules are useful in systems that support plug-and-play of different applications which offer the same interface. Plug-and-play is a commonly desired feature for multiphysics systems where users want to try a variety of domain-specific simulation applications.

3.9.2 Publishing Native Data

In the *COM* infrastructure, application-native data are encapsulated into named *DataItems*. Multiple DataItems may be gathered together into composite named data types called *DataGroups*. All DataItems and DataGroups belong to one or more CI objects with a defined access policy and scope. DataItems and DataGroups are further described below.

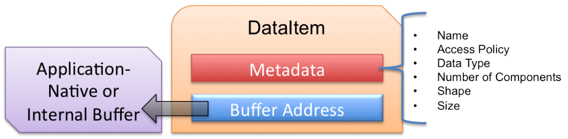
DataItem A DataItem is the lowest level of encapsulation provided by the infrastructure for a component's native data. As illustrated in Figure 6a, each DataItem has a name, a pointer to a memory address (i.e., a buffer), and a collection of metadata, which describes the relevant sizes, shape (i.e., how the data are arranged in the buffer), datatype, and access policy. Buffers associated with DataItems can have contiguous or staggered layouts with or without constant strides. Strides indicate offsets between neighboring entities of the DataItem. See Figure 9 for examples of data layouts currently supported by *COM*. All DataItems are associated with either a CI or a DataGroup.

Aggregate DataItems For example, consider a DataItem named "Coordinates," which stores the Cartesian coordinates of a mesh. The number of components for the Coordinates DataItem is three, one for each of X, Y, and Z. If the mesh has N nodes, the DataItem has total size of $3N$. If the application has these coordinates in a floating point array data structure with two scalar fields per point, say pressure and temperature, for example, arranged as $\{X_1,Y_1,Z_1,P_1,T_1,X_2,Y_2,Z_2,P_2,T_2\ \ldots.\}$, then the layout is *pointwise strided* (e.g., see Figure 9).

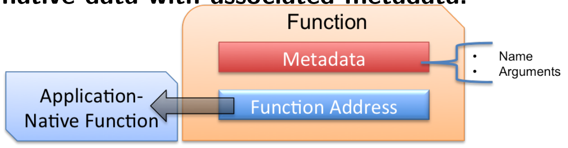
The Coordinates DataItem is created with a pointer to X_1, where it expects to find three floating point values, one for each component. The DataItem has a stride of five to skip P_n and T_n. The "Pressure" DataItem would get number of components 1, with a stride of five to skip $T_n, X_{(n+1)}, Y_{(n+1)}$, and $Z_{(n+1)}$.

DataGroup A DataGroup, depicted in Figure 6c, is a uniquely identified collection of DataItems, essentially representing a simple composite data type. Drawing on the example above, a "Surface" DataGroup could consist of the "Coordinate," "Pressure," and "Temperature" DataItems. Components can then reference given data by referring to "Surface.Coordinate" or "Surface.Pressure." DataGroups are quite useful in situations where there will be many of a given construct that the DataGroup was created to represent. For example, there would be an instance of the Surface DataGroup for each surface of a given *type* in a simulation. An example of a surface type could be a surface with a given boundary condition, so that each boundary condition has an associated surface, and thus a Surface DataGroup.

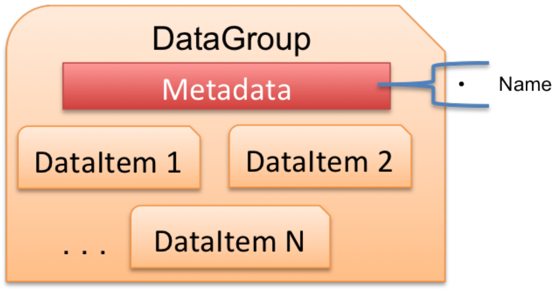
COM defines a number of aggregate DataItems. Aggregate DataItems enable high level, inter-module interfaces. For example, one can pass the "all" DataItem of a CI to a parallel I/O routine to write all of the contents of a CI into an output file with a single call. As another example, it is sometimes more convenient for users to have *COM* allocate memory for the DataItems and have the component codes retrieve memory addresses for those buffers from the CI. *COM* provides an API for memory allocation, which takes a CI–DataItem name pair as input. A user can pass in "all" for the DataItem name, which will create *COM*-allocated memory for all the unregistered DataItems (i.e,. those whose buffer pointers are NULL).



(a) A DataItem provides access to application-native data with associated metadata.



(b) Void functions (i.e., with no return value) are published by providing a pointer to the function with a description of its arguments.



(c) DataGroups are named groups of DataItems.

Figure 6: Basic infrastructure primitives.

Component-wise Access *COM* also allows for multi-component DataItems to be accessed component-by-component by specifying "*i*-DataItemName" ($i \geq 1$) to refer to the *i*th component. Again drawing on the above example, to get the X-coordinate, one could refer to "1-Coordinate."

All DataItems and DataGroups are created as part of a given CI and associated with application-native data by registering the physical location of the source data buffer. *COM* also offers an API for internal allocating space for data buffers. A DataItem can be associated with either the CI or a DataGroup. Examples of CI-associated DataItems include a data structure that encapsulates the internal states of a module or some control parameters. An example of a DataGroup DataItem is an integer flag for the boundary condition type of a surface patch or the surface solution field.

The name of DataItems and DataGroups IDs must be unique within a CI. Components can refer to DataItems and DataGroups by name (as described above) and can access their stored data by obtaining a "handle." The handle of a DataItem can be either mutable or immutable, where an immutable handle allows only read operations to its referenced DataItem, similar to a "const reference" in C++. Each DataGroup has a user-defined positive integer ID, which must be unique within the CI across all processors but need not be consecutive.

In a parallel setting, a DataGroup belongs to a single process, while a process may own any number of DataGroups. A CI can have multiple DataItems and DataGroups, but one current restriction of the CI is that all of its DataGroups must have the same types of DataItems, although the sizes of DataItems may vary. This restriction means that a component must create multiple CIs, one for each type of DataGroup defined by the application. We hope to address this limitation in the future with the concept of *DataFrames*.

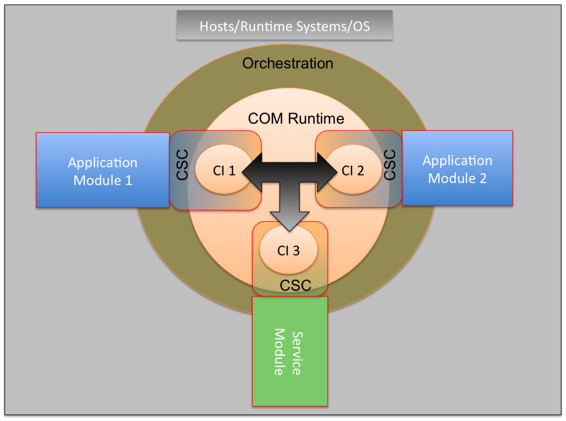


Figure 7: *COM* architecture with multiple software *modules*, wherein no ICC layer is needed to communicate. Each module is loaded at runtime and shares the same process with the orchestrator.

3.9.3 Publishing Native Methods

A CI may contain not only data members but also function members. As shown in Figure 6b, *COM* represents application-native functions with a function pointer, a set of metadata about the function, and its arguments. Components can register application-native functions into a CI to allow other components to invoke the function through the *COM* interface. Registration of functions enables a limited degree of runtime polymorphism. It also overcomes the technical difficulty of linking object files compiled from different languages, where the mangled function names can be platform and compiler dependent; as a reminder, function names must be unique within a given CI. This section describes the main features of *COM* function encapsulation.

Member Functions Outside the realm of the simplest functions, a typical function needs to operate with certain internal states. In object-oriented programs, such states are encapsulated in an "object," which is passed to a function as an argument instead of being scattered into global variables, as in traditional programs. In some modern programming languages, this object is passed implicitly by the compiler to allow for cleaner interfaces. In mixed-language programs, even if a function and its context object are written in the same programming language, it is difficult to invoke such functions across languages because C++ objects and Fortran 90 data structures are incompatible.

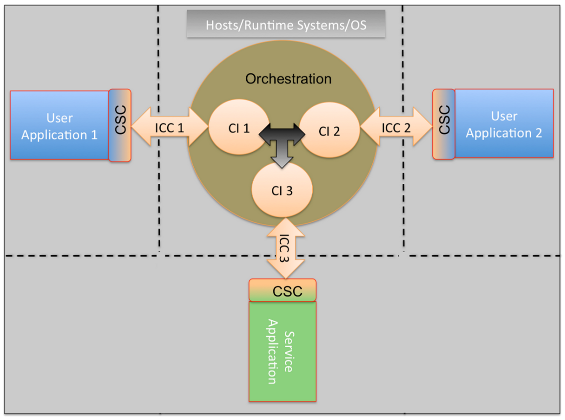


Figure 8: *COM* architecture with multiple software components, including two user applications and one service application. Each component may be operating in its own process and separated by a physical network.

To address this problem, *COM* provides for member functions of DataItems. Specifically, during registration a function can be specified as the member function of a particular DataItem in a given CI. *COM* keeps track of the specified DataItem and passes it implicitly to the function during invocation in a way similar to C++ member functions. Because the caller no longer needs to know the context object of the callee, this concept overcomes the incompatibility without sacrificing object-oriented behavior.

Optional Arguments *COM* supports the semantics of optional arguments similar to that of C++ to allow for cleaner codes. Specifically, during function registration, a component can specify the last few arguments as optional. *COM* passes null pointers for those optional arguments which are missing corresponding actual parameters during invocation.

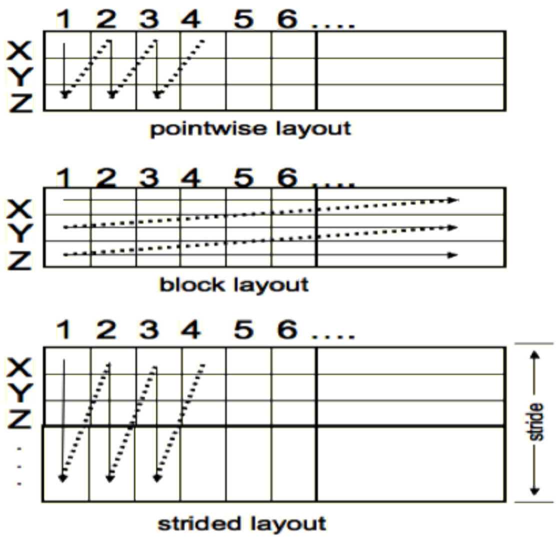


Figure 9: Common data layouts in memory.

3.9.4 Inter-Component Communication

COM's ICC implementation is not fully implemented; however, despite being incomplete, an architecture including ICC over TCP/IP is testing positively. *COM*'s architecture with direct memory access simply omits the ICC (e.g., see Figure 7). Strictly speaking, the abstract architecture model does not support this construction and an ICC should be present with a direct access substrate. This shortcoming of ICC can be planned for the near future, as well as implement an MPI substrate for the ICC.

3.10 Multiphysics Services

The services layer is where the implementation begins to focus on a more domain-specific application (i.e., multiphysics-simulation specific). Our goal is to provide general, simulation-specific, application-independent capabilities that reduce the cost of development of domain-specific simulation applications. The application-independent multiphysics capabilities and support services are implemented using the primitive constructs capabilities provided by the SIT implementation (i.e., the *COM* package).

Our multiphysics services are presented as service adapters and service applications as indicated in abstraction (Section 3.3). Section 3.10.1 will outline our service adapters, while the various service applications are discussed in Section 3.10.2.

3.10.1 Service Adapters

We have added the crucial simulation-specific capability of dealing with discrete geometries (i.e., meshes) and mesh-associated data fields as service adapters. These adapters inherit from the SIT implementation but offer enhanced, built-in simulation-specific capabilities. An explanation of our service adapters follows.

Support for Discrete Geometries We have added built-in, permanent mesh-specific DataItems to DataGroups to provide inherent support for discrete geometries. Our specialized DataGroups, with these permanent DataItems, are called *Panes* to differentiate them from generic DataGroups. Each pane comes with several built-in DataItems that may or may not be populated.

The new DataItems represent the nodal coordinates, element connectivities, and inter-connectivity among panes. These specialized DataItems provide support for block-structured and unstructured meshes with linear or quadratic 2D and 3D elements. Currently supported are hexahedron, tetrahedron, prism, and pyramid elements, along with their 2D analogues for surface meshes (i.e., squares and triangles). Mesh DataItems are given special, reserved names and data types.

The nodal coordinates are double-precision, floating-point numbers with three components per node. If the coordinates of a DataGroup are stored contiguously, the storage can be registered using the DataItem name “nc.” Otherwise the *x*-, *y*-, and *z*-components must be registered separately using the DataItem names “1-nc,” “2-nc,” and “3-nc,” respectively.

COM supports both surface and volume meshes, which can be either multi-block structured or unstructured with mixed elements. For multi-block meshes, each block corresponds to a DataGroup/Pane in a CI. Structured meshes have no connectivity tables and the shape of a Pane is registered using the DataItem name “st.” For unstructured meshes, each Pane has one or more connectivity tables, where each connectivity table contains consecutively numbered elements of the same type.

Each connectivity table must be stored in an array with contiguous or staggered layout, registered using reserved keywords (such as “t3” or “t-3” for contiguous or staggered 3-node triangles, respectively). To facilitate parallel simulations, *COM* also allows a user to specify the number of layers of ghost nodes and cells for structured meshes, as well as the numbers of ghost nodes and cells for unstructured meshes.

Support for Solution/Field Data Service adapters for mesh-associated solution fields have also been added. These adapters allow DataItems to be associated with the DataGroup’s mesh nodes or elements. A nodal or elemental DataItem of a DataGroup/Pane is conceptually a 2D dataset: one dimension corresponds to the nodes/elements and the other corresponds to the data within a node/element. Field variables are nodal or elemental DataItems that have no pre-designated names or data types. A user must first define such an DataItem in the CI and then register the addresses of the DataItem for each Pane.

For a specific Pane, if a field variable is stored in an array with contiguous or staggered layout, then the array is registered with a single call. If it is stored in multiple arrays, then the component must register these arrays separately, similar to registering staggered nodal coordinates.

3.10.2 Service Applications

The current implementation of the multiphysics infrastructure includes several service applications. These applications provide key capabilities for multiphysics simulations building on the lower layers of the infrastructure abstraction and depending on the service adapters discussed in Section 3.10.1. The service applications we are currently distributing support a large number of multiphysics simulations with abutting domains and few-to-no physical processes occurring at the interface between domains. It is important to note that simulations with interface physics are not precluded from our current distribution, but the infrastructure is not offering any *built-in* service for surface propagation, nor solution mapping for overlapping domains. The services applications currently implemented are explained below and arranged according to the capability they offer.

Mesh-associated communication operations
: Our *SurfMap* package provides services that perform geometry/process mapping (i.e., matching interacting geometries across process and component boundaries). *SurfMap* provides several routines for MPI communication across adjoining surface meshes, including MPI collective operations on shared boundaries. Also provided are facilities for calculating *ghost zones* for connecting surface meshes.

Surface-specific numerics
: The *SurfUtil* package provides generic numerical routines to calculate face normals, areas, curvature, and quality metrics on discrete surfaces.

Simple math for field data

Simpal is a service application that provides simple scalar, vector, and matrix mathematical operations commonly needed for field data. These services operate on the primitive DataItems provided by *COM*, providing a powerful API for performing the types of operations commonly associated with implementing numerical physics (e.g., implementation of jump-conditions, convergence checks, etc).

Surface-surface data mapping

Services for accurate and, if necessary, conservative transfer of data between disparate discretizations are provided by the *SurfX* package. *SurfX* implements an advanced, state-of-the-art data transfer method developed at the University of Illinois ASCI Center. Robust methods for accurate and conservative transfers are key in multiphysics simulation fidelity and reliability [Jiao and Heath (2005), Jiao et al. (2006)].

The *SurfX* package enables physics modules with abutting domains to exchange quantities across the interface between them. By construction, the data transfer scheme exactly conserves mass, momentum, and energy across the interface, even if their respective meshes do not match [Jiao and Heath (2005), Jaiman et al. (2005), Jiao et al. (2006)]. Conservation is achieved by using the common refinement of the two meshes, each subdivision of which lies entirely within a cell face in both surface meshes [Jiao and Heath (2005)].

Interpolation errors are minimized in the least squares sense, leading to a scheme that is several orders of magnitude more accurate than previous conservative methods. The high accuracy of the transfer method makes it ideal to monitor the conservation of the overall system; identify unexpected errors as quickly as possible, such as those result from either software bugs or hardware failures; and to audit the propagation of errors among different physics modules.

Disk I/O Services

Low level disk I/O operations are encapsulated by the *SimIO* package. *SimIO* offers a high level API allowing components to write out standard format, platform-independent files containing state and solution information. *SimIO* operates on CI, DataGroups (and Panes), and DataItems, so components must register data with *COM* if they wish to use *SimIO* to handle parallel I/O. *SimIO* currently supports *HDF4* and *CGNS* formats with conversion tools for *VTK* and *Plot3D* formats.

3.11 Multiphysics Orchestrator

The *Simulation Integration Manager* (*SIM*) is our orchestration package. It has the mechanisms and constructs required for managing multiple applications, their data, and control flow between component objects. *SIM* is oriented towards partitioned multiphysics (e.g., see Figure 1) where any jump conditions between simulation components and solution data are exchanged at the interfaces between their respective domains. *SIM* is designed around several primitive constructs that serve as the building blocks for developing coupled simulation drivers. A description of *SIM* constructs follows.

Agent

The *SIM* Agent is responsible for interfacing with a particular application and for constructing a set of CI to represent a particular application. Typically, a given simulation will require a particular interface (i.e., particular set of functions) to facilitate plug-and-play of different modules; the agent specifies a set of CI that present the required interface.

Action

The Action construct encapsulates a generic procedure that operates on CI and DataItems and presents it as a standard interface. As currently implemented, actions provide an "`init()`, `run()`, and `finalize()`" interface and hide further (arbitrary) complexity inside.

Scheduler

The Scheduler is responsible for operating a sequence of Actions. Actions can be sequenced, initialized, run, and finalized in arbitrary order. The scheduler offers several interfaces for scheduling actions and handling exceptions from action operation.

Coupling

A *SIM* coupling object encapsulates a given scheduler and is responsible for constructing the scheduler by adding actions (i.e., scheduling actions) to implement a particular simulation or class of simulations (e.g., aeroelasticity with CFD and CSM agents).

Driver

A driver represents the highest level of orchestrator function. It controls the simulation by driving the coupling object and implements any timestepping routines.

When an implicit solver is involved, predictor-corrector iterations are required to converge on the jump conditions between domains. The implementation of these jump conditions involves operations, such as manipulating data on some interface mesh or transferring data between different meshes. These additional operations are independent of the physical modules, are provided by the service modules, and invoked by a centralized orchestration module at this level.

4 Implementation and Application

4.1 *IMPACT*-enabling Process

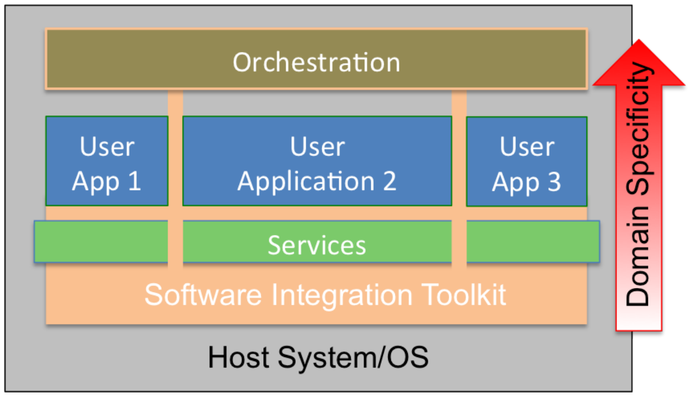


Figure 10: Through shallow modifications, existing user applications can use capabilities from the Services layer, interface with other user applications, and participate as part of a composite software system coordinated by the Orchestration layer. All inter-component exchanges are mediated by the Software Integration Toolkit. Problem and domain specificity increases from bottom to top.

The ultimate goal of the *IMPACT* project is to minimize programming effort for computational physicists interested in coupling single-physics codes. The first step for the user is to identify at least a pair of single-physics codes and follow a series of subsequent steps to couple them through *IMPACT*; we refer to this process as *IMPACT*-enabling.

IMPACT-enabling requires some knowledge about each of the codes involved in this composite software development effort. As discussed in Section 3.3, *IMPACT* provides a well-defined abstraction for this purpose and tries to minimize the effort level needed while streamlining the programming process for users.

Figure 10 illustrates the overall architecture of an *IMPACT*-enabled product. At the very top level, a user must develop the coupling algorithms using the constructs provided in the orchestration layer. Different coupling schemes can be used for different physics combinations. In Section J we provide a thorough discussion about the mathematical aspects of coupling and we refer users to this section for more information.

The orchestrator is usually implemented as a single standalone application which interacts directly with the end-user. It may also be a middle-ware or a service module implemented for a more general front-end application; these types of decisions are made at the user's discretion. Regardless of these details, the orchestrator interacts with application modules (the blue components in Figure 10). Therefore, these application modules should be developed as a separate component for each physics solver.

The physics modules use *IMPACT* facilities including data and member function abstraction and encapsulation constructs (the light brown area of Figure 10) and potentially any combination of services they need (the green components in Figure 10) to interact with the core of the physics solver. As Figure 10 indicates (and as discussed in Section 3.8), the low level services and constructs are generally applicable to any physics solver. The higher level components will be more domain-specific. In the forthcoming sections, we will provide a series of examples applications developed and build using *IMPACT*. These application programs provide reference examples of how to use *IMPACT* and they are provided in our publicly accessible repository.

4.2 *ElmerFoamFSI*

4.2.1 Overview

OpenFOAM[*] is an open-source, computational fluid dynamic (CFD) code widely adopted by researchers across most areas of engineering and science. The code has a long list of features and facilities to solve for anything from fluid dynamics to heat transfer and solid mechanics. *OpenFOAM* has an internal fluid-solid

[*]`http://www.openfoam.com/`

(or fluid-structures) module to solve for problems involving interaction between fluid and structures.

Elmer[*] is another well-known open source finite element analysis (FEA) software with a sizable user base. Similar to *OpenFOAM*, *Elmer* has physical models and capabilities to simulate a wide range of problems including structural mechanics, fluid dynamics and electromagnetics. For further information about these codes, we refer reader to the comprehensive documentation of these codes on their respective websites.

For the first example application of *IMPACT*, we choose these two well known codes to build a standalone multiphysics solver. As an example of application for *IMPACT* infrastructure, we coupled *OpenFOAM* with *Elmer* to generate the *ElmerFoamFSI* product, a standalone software capable of simulating problems involving fluid-structures interactions (FSI).

For this purpose, *ElmerFoamFSI* utilizes one of *OpenFOAM*'s incompressible flow solvers and *Elmer*'s structural mechanics solver. In *ElmerFoamFSI* the fluid domain is discretized using finite volume (FV) scheme and the structures domain is discretized into a finite element (FE) mesh. These numerical schemes are well known in their respective communities, and *IMPACT* provides the required abstraction level to support both FV and FE schemes.

The application modules for *Elmer* and *OpenFOAM* are implemented using a similar methodology, with deviations made only when necessary to address specific concerns unique to each code. There are, of course, differences between the two applications ranging from overall structure to programming language. For instance *OpenFOAM* is a C++ code and *Elmer* is a Fortran 90 code. However, the essential methodology followed in creating application module for each code is similar thanks to the *IMPACT*'s full support for these programming languages. Due to this similarity in approach, different stages of the implementation of these modules are discussed generally here to avoid redundancy with specific examples when appropriate.

4.2.2 Development Process

In this section the development process of *ElmerFoamFSI* is discussed. The process has five distinctive stages which should be followed for any *IMPACT*-enabling project. We have tried to use a general tone for these steps; however, specific details are provided as needed.

Stage 1: Elementary *IMPACT* module The first step in creating a module from an application is to develop an empty "test module" that can be loaded by the driver via *IMPACT*. This test module must be compiled as a dynamic library so it can be loaded by *IMPACT* at run time.

Once this simple task is complete, the module can be gradually enriched to include all member data and functions which expose relevant parts of the application core to *IMPACT*. This inclusion of code from the coupled application is generally done in stages to make the process more manageable. In the first stage, the new module should be able to call the highest level functions needed to startup and run each application involved. The function(s) can be registered from the module to *IMPACT*, and therefore be accessible from the outside world.

Stage 2: Driver utility In this stage, the native application can essentially be run and managed by an external driver. At this point it is prudent to test the development and verify that the same results can be achieved by solving problems with the untouched code and the new module/driver system. In order to test the implementation of the module during the process, a driver for the module can be developed. This

[*]`https://www.csc.fi/web/elmer`

driver can be a standalone product (optional) or can be gradually morphed to act in the place of the ultimate orchestrator. The driver code essentially will provide access to the application module through *IMPACT*.

Stage 3: Functional *IMPACT* module In the next stage, the user must determine three major components of the native code: i) initialization, ii) running, and iii) finalization. Once these pieces identified, they should be broken into three separate module functions callable by the driver. These functions are fairly straightforward with initialization encompassing all set up to be done before running and finalization encompassing all clean up to be done after.

The most crucial aspect of the run function is that it should be capable of stepping the application's solver in two different ways. It may need to take multiple steps in time to a specified end time or only take one step forward in time. In this way, the driver can control the flow of information in and out of the application at each driver time step.

Note that in some cases one may want the module application to take smaller timesteps than the timesteps requested by the overall driver or orchestrator. These decisions are taken based on the user discretion, the physics involved, and type of problems to be solved. Ensuring that the time-stepping can be controlled in this function may involve breaking apart and refactoring the native application.

In the case of *Elmer* it was fairly straightforward to locate the time stepping loop and incorporate all previous and subsequent functionality into initialize and finalize functions for the module. *OpenFOAM* proved to be more difficult based on the organization of the code, which illustrated some of the potential differences in coupling software.

Once the driver can successfully control the time stepping of the application, the remaining task is to register the application's data with the *IMPACT*. In the case of *Elmer* this meant accessing the mesh data associated with the FSI BC, the loads on the structure, and the displacements calculated by the solver. Each of these components must be accessed from *Elmer* and, therefore, they need to be registered with *IMPACT*. In the final coupling, the driver/orchestrator can thus access calculated loads from *OpenFOAM*, provide those loads to *Elmer*, then do the same with the displacements from *Elmer* to *OpenFOAM*.

Stage 4: Module testing After the implementation of a fully functional *IMPACT* module (or during development), it is highly necessary to test the code. The driver utility plays major role in testing the module. A suite of tests should be created to ensure the proper operation of all registered functions and data using the test driver mentioned above. These tests include regression tests, which ensure the module/driver obtains the same results as the original application, as well as unit tests that call functions in the module and verify that data is updated correctly. In the *ElmerFoamFSI* product, a series of these tests are provided in the *testing* folder of each module for the reference.

Stage 5: Orchestrator After completing the module implementation, the remaining development task is to design and implement an orchestrator that can work with both modules in the way their own test drivers do. In the *ElmerFoamFSI* example the orchestrator code is developed separately and solves an FSI problem described by a series of input files that describe the problem for the *Elmer* and *OpenFOAM* modules and orchestrator code. The orchestrator is the front-end application for our *ElmerFoamFSI* product. The main functionalities performed by *ElmerFoamFSI* orchestrator are as following:

- analyzing user's input,
- loading and initializing *Elmer* and *OpenFOAM*

- stepping each application in time and exchanging physical quantities (tractions, displacements etc.) between them
- output solution information (probe files, hdf and vtk outputs etc.)
- finalizing each application and ending the simulation

A series of example FSI problems, regression-, and unit-tests are provided by *ElmerFoamFSI* to verify the functionality of each of its components. A brief discussion of these examples are provided in the subsequent sections of this chapter. More details about the extensive testing of the *ElmerFoamFSI* is beyond the scope of this report, and interested readers are referred to our publicly accessible repository [*] and the online documentation [†].

4.3 *Rocstar*

4.3.1 Overview

The *Rocstar* simulation application was developed and successfully deployed over the lifetime of Illinois' DOE ASCI Center for Simulation of Advanced Rockets (CSAR) as a predictive simulation tool for solid rocket motor (SRM) internal ballistics and performance (Dick et al. (2006)). It is recognized in the SRM industry as one of the world's only fully 3-D, time-accurate SRM simulators. *Rocstar* is MPI parallel and has demonstrated scalability to tens of thousands of cores on many of the world's largest HPC platforms. Since the time of CSAR, *Rocstar* has been utilized in many academic, government, and industry settings to conduct predictive simulations of general fluid-structure systems which interact across moving and reacting interfaces.

As illustrated in the *Rocstar* software architecture Figure 11, *Rocstar* orchestrates multiple software and simulation components in concert to simulate multiphysics systems. *Rocstar*'s individual component codes are based on fundamental research and development in turbulence modeling, multiphase flow, constitutive modeling, combustion chemistry, computational mechanics, coupling methodology, etc. Although the original target for *Rocstar* was internal ballistics of an SRM, *Rocstar* is capable of simulating a wide variety of multi-component systems involving fluid dynamics, structural dynamics, combustion, and their interactions. *Rocstar* features multiple state-of-the-art solvers for various types of physical components.

4.3.2 *Rocstar* Modules

Rocstar includes several complementary finite-volume compressible flow solvers that are formulated on moving meshes (ALE) to handle geometrical changes. Modules *Rocflo* and *Rocflu* are general use CFD solvers, each with strengths for specific problems. *Rocflo* uses either a centered or an upwind scheme with Roe flux splitting on multi-block structured meshes. *Rocflu* operates on unstructured tetrahedral or mixed-element meshes and employs a novel high-order WENO-like approach, as well as the HLLC scheme to handle strong transient flows including shocks. Further, the fluid solvers are integrated with additional physics modules for simulating turbulent and multiphase fluid flows. Supporting integrated physics modules include *Rocturb*, which provides three classes of turbulence models, including LES, RANS, and hybrid models (either LES with a near-wall model or DES).

[*]`https://github.com/IllinoisRocstar/ElmerFoamFSI`
[†]`http://illinoisrocstar.github.io/ElmerFoamFSI/index.html`

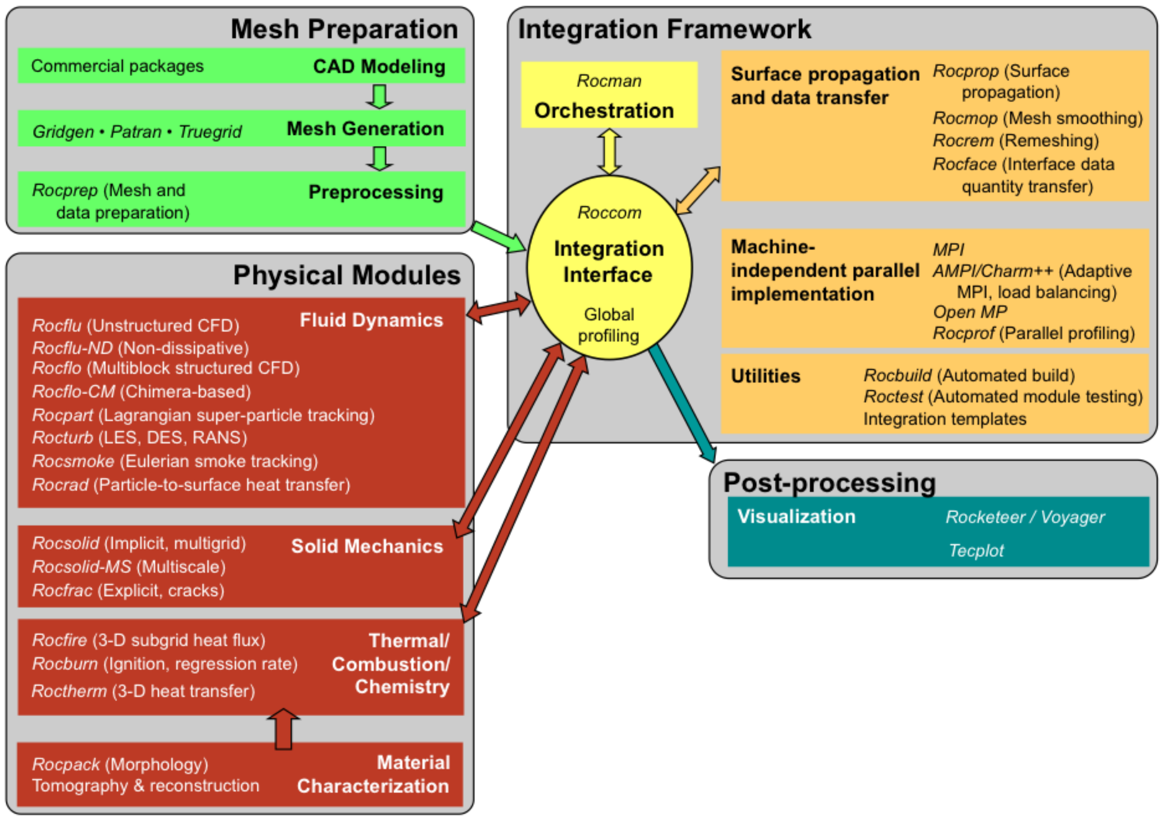

Mesh Preparation
Commercial packages
CAD Modeling
Gridgen • Patran • Truegrid
Mesh Generation
Rocprep (Mesh and data preparation)
Preprocessing
Physical Modules
Rocflu (Unstructured CFD)
Rocflu-ND (Non-dissipative)
Rocflo (Multiblock structured CFD)
Rocflo-CM (Chimera-based)
Rocpart (Lagrangian super-particle tracking)
Rocturb (LES, DES, RANS)
Rocsmoke (Eulerian smoke tracking)
Rocrad (Particle-to-surface heat transfer)
Fluid Dynamics
Rocsolid (Implicit, multigrid)
Rocsolid-MS (Multiscale)
Rocfrac (Explicit, cracks)
Solid Mechanics
Rocfire (3-D subgrid heat flux)
Rocburn (Ignition, regression rate)
Roctherm (3-D heat transfer)
Thermal/ Combustion/ Chemistry
Rocpack (Morphology)
Tomography & reconstruction
Material Characterization
Integration Framework
Rocman
Orchestration
Roccom
Integration Interface
Global profiling
Surface propagation and data transfer
Rocprop (Surface propagation)
Rocmop (Mesh smoothing)
Rocrem (Remeshing)
Rocface (Interface data quantity transfer)
Machine-independent parallel implementation
MPI
AMPI/Charm++ (Adaptive MPI, load balancing)
Open MP
Rocprof (Parallel profiling)
Utilities
Rocbuild (Automated build)
Roctest (Automated module testing)
Integration templates
Post-processing
Visualization
Rocketeer / Voyager
Tecplot

Figure 11: Overall architecture of the *Rocstar* multiphysics simulation application.

Aside from its compressible-flow solvers, *Rocstar* is also equipped with a finite-element structures solver called *Rocfrac*. *Rocfrac* uses an explicit finite-element scheme and supports different element shapes including tetrahedral and hexahedral elements and their 2D analogs. *Rocfrac* is also capable to account for moving meshes (ALE), material and geometric non-linearities.

On top of fluid and structures modules, *Rocstar* is also equipped with a module to account for solid-fuel burning during simulation called *RocburnAPN*. Finally, *Rocstar* also comes with service modules for mesh refinement, solution transfer and many other ones which their description is beyond the scope of this report. Interested readers are referred to *Rocstar*'s online source code repository[*] and its documentation[†].

4.3.3 *IMPACT*-enabled *Rocstar*

The *IMPACT* project originally started by piecing apart the simulation management and orchestration facilities of *Rocstar*. In this project, we have carefully separated the modules from *Rocstar* extended, then generalized them into *IMPACT*. Since *Rocstar* is a comprehensive multiphysics simulation program, and is also the mother of *IMPACT*, we have then decided to rebuild *Rocstar* code based on the new hardened *IMPACT* core. The new *Rocstar* code has been renamed *Rocstar Multiphysics*. All of the modules of *Rocstar* including *Rocflo*, *Rocflu*, *Rocfrac* were then restructured based on *IMPACT*. In the subsequent sections, some of the example simulation performed by *Rocstar Multiphysics* are discussed in detail. The original

[*] `https://github.com/IllinoisRocstar/Rocstar`
[†] `https://sourceforge.net/projects/rocstar/`

code for *Rocstar Multiphysics* can be accessed from our public repository*.

4.4 Stand-alone Module Testing

In this section, we briefly present our verification study for application modules developed for *Elmer* and *OpenFOAM*. The testing framework utilized for each module also includes extensive unit and regression testing. These tests can be performed upon application is successfully built by issuing the command *'make test'*. For further discussion about testing we refer to the respective section of the online documentation for *ElmerFoamFSI*†.

4.4.1 *OpenFOAM* Stand-alone Testing

After selecting *OpenFOAM* as the open CFD solver for *ElmerFoamFSI*, some example V& V problems for *OpenFOAM* were chosen. These selected problems consider a situation with 2D laminar flow around a cylinder as well as a cylinder with a rigid horizontal bar mounted at the rear. The purpose of these validation problems is to ensure that the open-source solvers used behave properly in thdered for *OpenFOAM* are single phase and single material with incompressible flow. Both steady-state and transient/unsteady problems were employed.eir standalone mode (uniphysics) before integrating the codes. This also provides a starting point for determining the best method of integration. The problems consi

Case 1: The first verification case study was a 2D study of steady-state flow over a cylinder with a parabolic inlet condition as described in Schafer and Turek (1996). Four different grid resolutions were used for this case, and the relative errors in $\Delta p, c_d$ and c_l for the different resolutions are presented in Table 1.

Table 1: Relative error in $\Delta p, c_d$, and c_l for Case-1 simulation with different grid resolutions.

Relative error	49 120 cell grid	12 280 cell grid	3 070 cell grid	730 cell grid
Δp	0.0119668	0.0505758	0.0372198	0.0838296
c_d	0.0169855	0.0392541	0.0186599	0.0285247
c_l	0.314267	31.3874	0.0959664	0.231629

The solutions produced by the different grid resolutions generally converge to within 10% of the reference values except for the lift coefficient c_l. It is known that c_l has increased sensitivity to simulation parameters over other quantities such as c_d and Δp. In particular, the lift-coefficient for the 12 280 cell grid shows a large discrepancy compared to the reference value. We believe that this discrepancy can be greatly reduced by reducing the time-step used. Figure 12 shows an image of the streamwise velocity for one run of this case.

*`https://github.com/IllinoisRocstar/Rocstar/tree/ElmerModule`

†`http://illinoisrocstar.github.io/ElmerFoamFSI/index.html#quick_start`

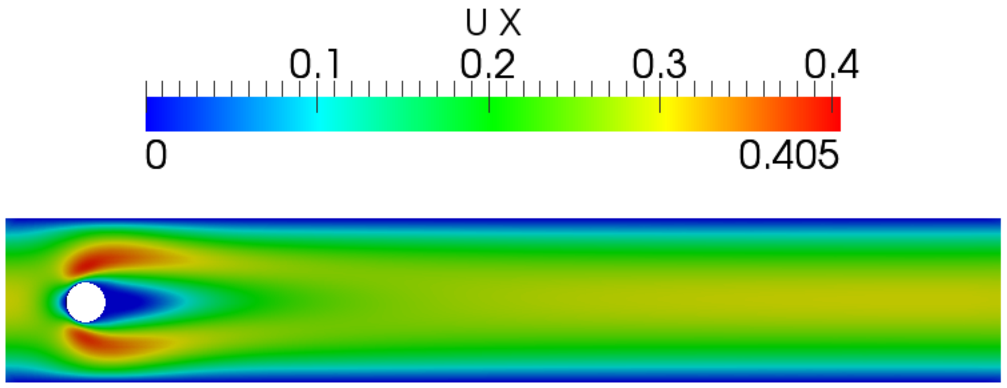


Figure 12: Streamwise velocity contour for Case-1 with parabolic inlet velocity and CFL = 0.1.

Case 2: This is identical to the first case except Case-2 is run with a higher Reynolds number than that in Case-1; i.e. 100 compared to 20 for Case-1. The higher Reynolds number triggers unsteadiness in the flow in the form of periodic vortex shedding and oscillatory flow downstream of the cylinder. After a stationary solution was attained, a quantitative comparison with the reference solution Schafer and Turek (1996) for Δp, c_{dmax}, c_{lmax}, and St was performed.

The pressure difference Δp was computed at time $t = t_0 + 1/2f$, where f is the shedding frequency and the inital time $t = t_0$ corresponds to the flow state where $c_l = c_{lmax}$. The Strouhal number was computed in the time-range of 15 - 30s. In this range the solution has reached a periodic oscillatory condition.

The result is presented in Table 2 for the three grid resolutions used. The relative errors are bounded and converge to the reference values with the finer grids. Figure 13 shows the streamwise velocity contour for one run from this case.

Table 2: Relative error in $\Delta p, c_d$, c_l, and St for Case-2 simulation with different grid resolutions.

Relative error	12 280 cell grid	3 070 cell grid	730 cell grid
Δp	0.025694	0.0298173	0.147044
c_d	0.00168824	0.0310062	0.0626892
c_l	0.006093	0.0578726	0.913368
St	0.0416667	0.0559896	0.186198

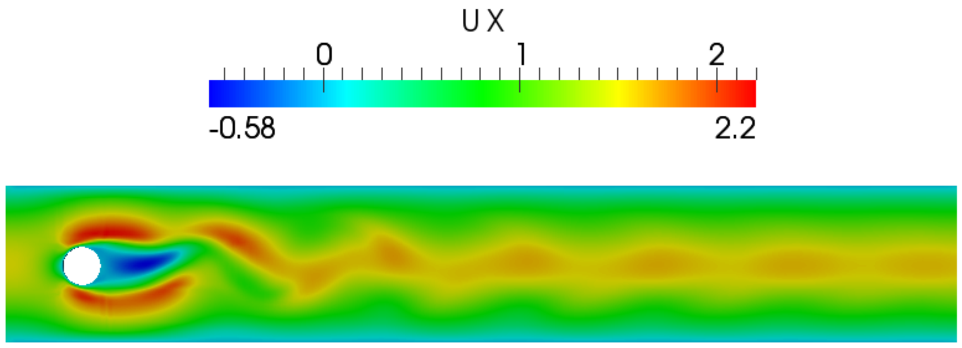


Figure 13: Streamwise velocity contour for Case-2 with parabolic inlet velocity

Case 3: This case used the same grids as the first two cases; however, the parabolic inlet velocity was prescribed as a profile oscillating in time. Four grid resolutions were used for this case and the relative errors in $\Delta p, c_d$ and c_l for the different resolutions are presented in Table 3.

Table 3: Relative error in $\Delta p, c_d$, and c_l for Case-3 simulation with different grid resolutions.

Relative error	49 120 cell grid	12 280 cell grid	3 070 cell grid	730 cell grid
Δp	0.206864	0.0355727	0.00949091	0.0972727
c_d	0.0655939	0.0485078	0.0100108	0.00324237
c_l	0.995322	0.279131	0.0901021	0.547201

As mentioned above, c_l is more sensitive to simulation parameters, accounting for the higher relative error. Figure 14 shows an image of the streamwise velocity at 4 s for one run of this case.

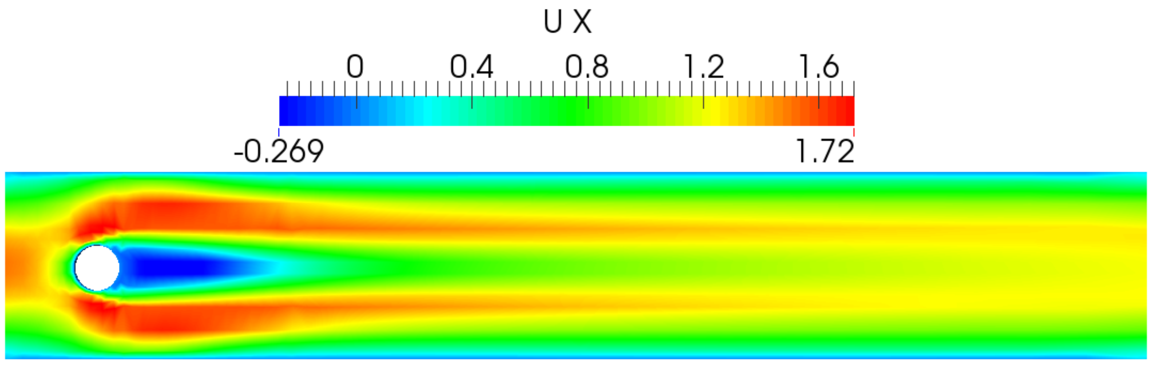


Figure 14: Streamwise velocity countour of Case-3 with parabolic inlet velocity at time 4 s.

Case 4: This case studies flow over a cylinder with a rigid beam attached. Like Case-1 this problem has a parabolic inflow condition and is steady-state. Four grid resolutions were used, similar to the previous cases. The relative errors in $\Delta p, c_d$ and c_l for the different resolutions are presented in Table 4, and Figure 15 shows the velocity magnitude for one run from this case.

Table 4: Relative error in $\Delta p, c_d$, and c_l for Case-4 simulation with different grid resolutions.

Relative error	49 120 cell grid	12 280 cell grid	3 070 cell grid	730 cell grid
Δp	0.0119668	0.0505758	0.0372198	0.0838296
c_d	0.0169855	0.0392541	0.0186599	0.0285247
c_l	0.314267	31.3874	0.0959664	0.231629

Performing these initial V&V cases with *OpenFOAM* laid the groundwork for being able to verify an *OpenFOAM* module created to work with *IMPACT*.

4.4.2 *Elmer* Stand-alone Testing

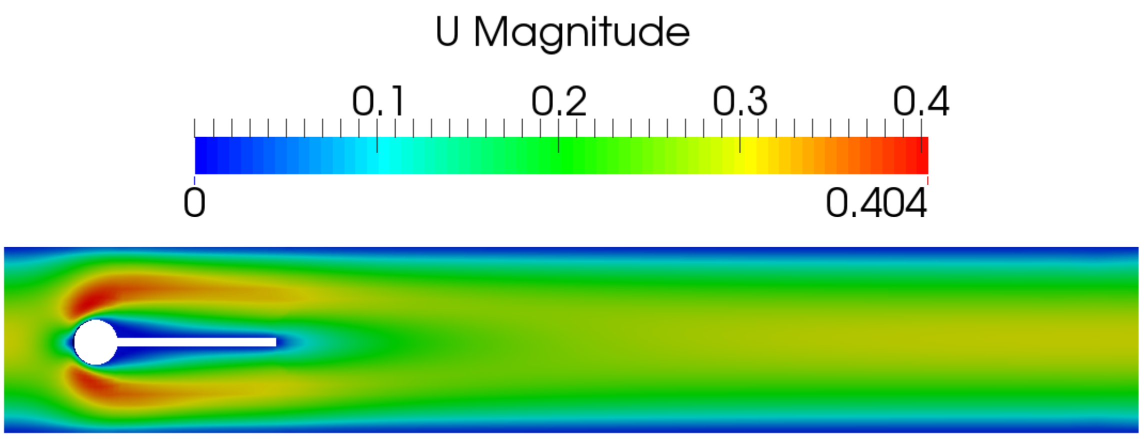


Figure 15: Velocity magnitude contour of Case-4 with parabolic inlet velocity.

In this V&V study, we verified whether the *Elmer* module could match a physically-derived analytical solution for the same scenario.

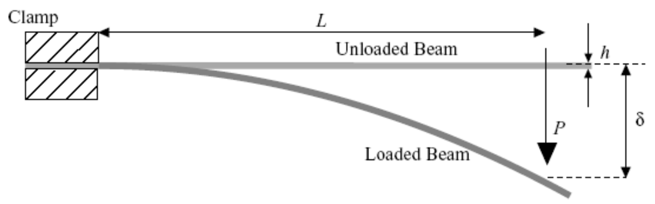


Figure 16: Typical cantilever beam showing length L, concentrated load P and transverse displacement δ. Graciously borrowed from `http://www.doitpoms.ac.uk/tlplib/thermal-expansion/printall.php`

Case 1: In this case, a static load is applied to the end of a cantilever beam to verify if the transverse displacement calculated by *Elmer* and the analytical solution are close, if not the same. The beam is a HronTurek-dimensioned beam. Figure 16 shows a typical cantilever beam of length L. For this HronTurek beam with $L = 0.351m$, we calculate the maximum deflection at the end of the beam. The curvature of the beam is given by

$$\kappa = \frac{M}{EI} = \frac{\partial^2 \delta_{\max}(x)}{\partial x^2} \tag{1}$$

where M is the concentrated load committing the force upon the beam, E is the Elastic Modulus and I is the moment of Inertia.

Solving for $\delta(x)$ where $x = L$ gives

$$\delta_{\max}(L) = \frac{PL^3}{3EI} \tag{2}$$

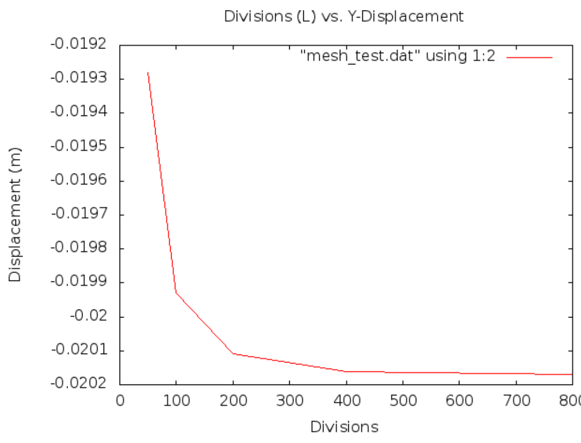
It is important to note that for the case of the HronTurek Beam with a cross sectional area of bh, the moment of Inertia (I) will be

$$I = \frac{bh^3}{12} \tag{3}$$

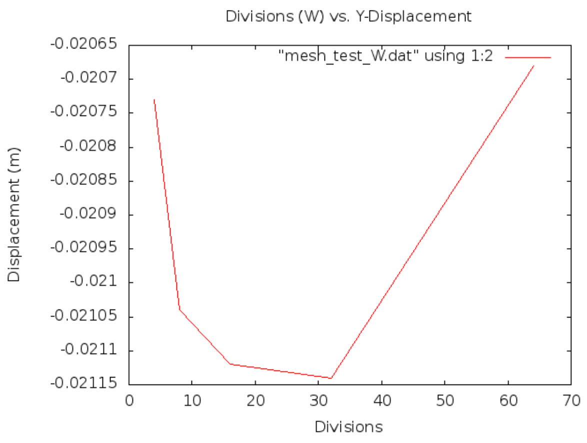
where h is the axis where the bending moment is applied. For this example $b = 0.0506m$ and $h = 0.02m$. *Elmer* utilizes the finite element method to discretize the beam to a mesh. Here the pressure is spread over the entire cross sectional area (of $\frac{N}{m^2}$). The tables and graphs below show how the horizontal displacement converges versus the density of mesh in each direction. Here, all variables are kept constant with $F(Pressure) = 5E5\,\frac{N}{M^2}$ and $E = 10.0E9$.

Div. (L)	Y-Disp.	Div. (W)	Y-Disp.	Div. (H)	Y-Disp.
50	-1.928E-2	4	-2.073E-2	10	-2.124E-2
100	-1.993E-2	8	-2.104E-2	12	-2.127E-2
200	-2.011E-2	16	-2.112E-2	18	-2.129E-2
400	-2.016E-2	32	-2.114E-2	21	-1.655E-2
800	-2.017E-2	64	-2.068E-2	24	-2.126E-3

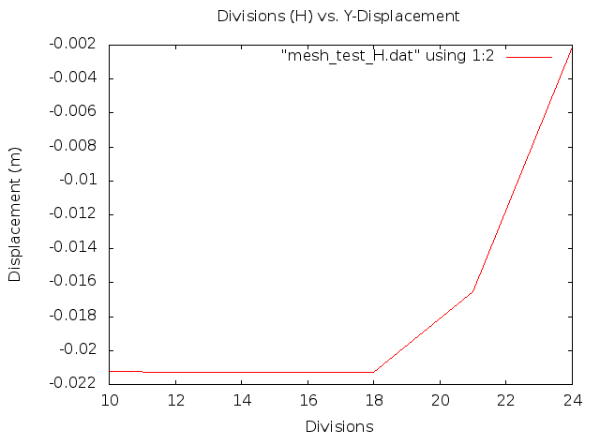
Table 5: Relationships showing converging of vertical displacement versus number of divisions in each direction.



(a) Meshes in the x-direction vs. the vertical displacement.



(b) Meshes in the z-direction vs. the vertical displacement.



(c) Meshes in the y-direction vs. the vertical displacement.

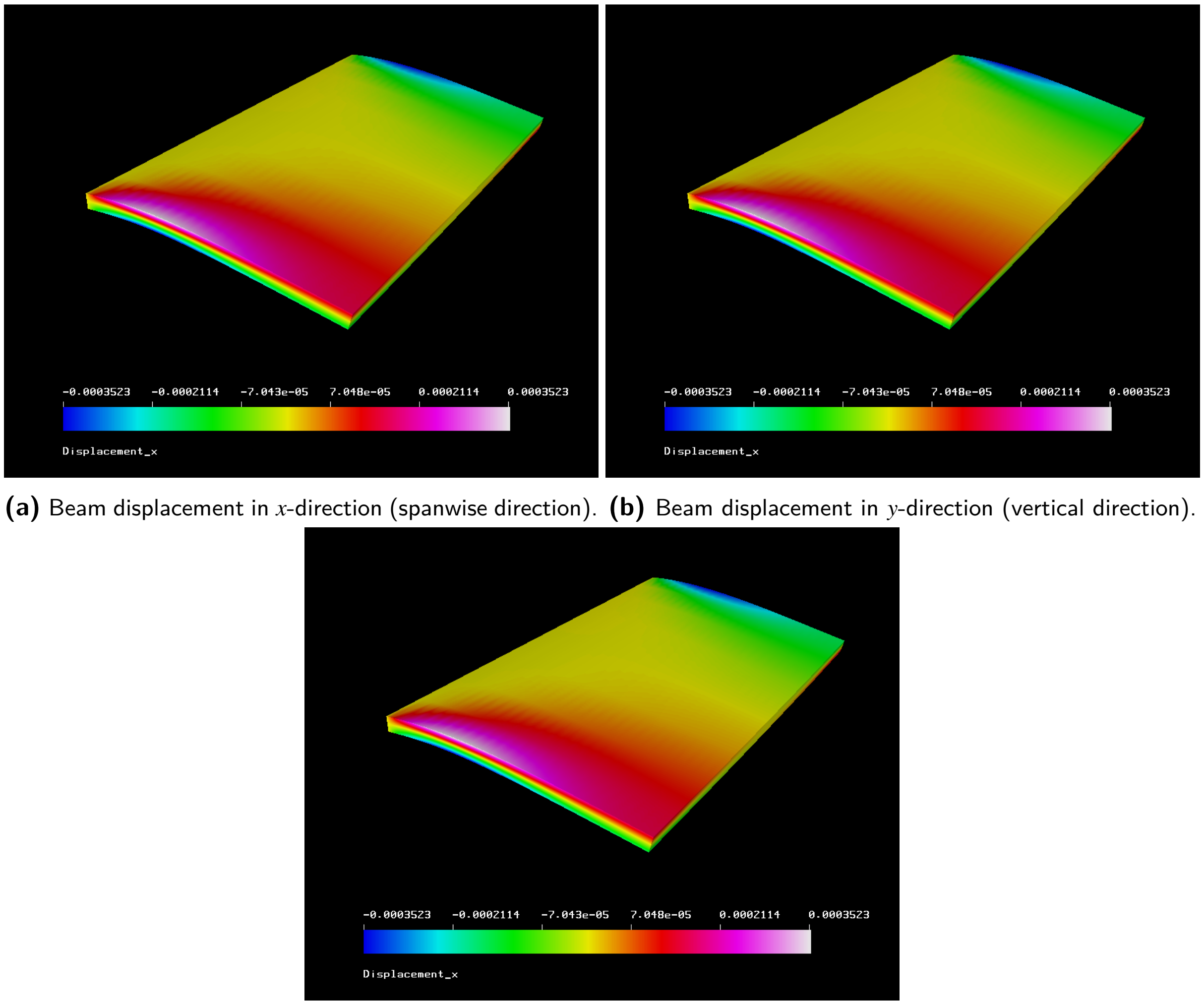
Figure 17: Graphical relationships of the mesh density in each direction and its effect on the vertical displacement.

The more nodes the mesh has (in each dimension) the better the approximation to the analytical solution will be; however, there is a trade off of computational cost to approximation accuracy. The analytical equations above are only valid once the reduction in error vs. the number of mesh nodes starts to converge. There are periods when the software will produce outliers which are ignored (as shown above in the second graph).

In this example, we found that *Elmer* closely matched the analytical solution once there were 400 nodes along the length, 16 nodes along the base, and 12 nodes along the height of the beam.

The finite element software *Elmer* has been selected to perform the structural mechanics piece of the example FSI problem. Like with *OpenFoam*, an initial V&V case was also performed with *Elmer* for a uniphysics problem. This test case was done to verify our use of *Elmer* and to develop a better understanding for the integration process.

Case 2: The test problem in this case study is a beam structure found in Turek and Hron (2006). The computed displacements due to the gravitational load are shown in Figure 18. The y-displacement, which is the primary comparison parameter in this case, is in reasonable agreement with the reference value reported in Turek and Hron (2006). The relative error is approximately 5%.



(a) Beam displacement in x-direction (spanwise direction). **(b)** Beam displacement in y-direction (vertical direction).

(c) Beam displacement in z-direction (chordwise direction)

Figure 18: Beam displacement for V&V problem using *Elmer*.

4.5 *ElmerFoamFSI* Testing

Three main test cases are used for verifying and validating *ElmerFoamFSI*. The three problems increase in complexity to verify the fact that tractions and displacements are properly exchanged between the fluids and solids solvers. In the first two cases, verification is performed by comparing against an analytical solution. For the third case, experimental data is referenced.

Additionally, each problem is solved first using only the *OpenFOAM* suite, which provides useful verification data for comparison against the coupled *ElmerFoamFSI* solution. Information about the specific solvers being used from *OpenFOAM* and *Elmer* is provided in the subsequent sections, with the discussion of each test case in Sections 4.5.1 – 4.5.4.

The *OpenFOAM* module implements an FSI solver based on built-in *icoFSIElasticNonLinUL* solver of *OpenFOAM*. *icoFSIElasticNonLinUL* uses an iterative algorithm with adaptive under-relaxation for strong coupling (partitioned approach in FSI, see Section J for discussion). Both solid (U_{solid}) and fluid fields (U,p) are solved for with separate solvers for the solid and fluid domains coupled together.

The fluid solver is an ALE finite volume incompressible flow solver with automatic mesh smoothing, and the solid solver uses finite volume discretization based on updated Lagrangian formulation. The adaptive under-relaxation is performed based on the Aitken algorithm. Note that *ElmerFoamFSI* uses only the fluid solver of *OpenFOAM* and, to make the example case as simple as possible, a non-iterative algorithm is implemented. For further information on the *OpenFOAM* solver, interested readers are referred to the *OpenFOAM* suite manual *OpenFOAM Extend Project* (2016).

When using *ElmerFoamFSI* the fluids solver is coupled to one of the solids solvers from *Elmer*. The tractions and displacements are transformed between the solvers using *IMPACT*. The solid solver selected from *Elmer* is a linear elastic solver with large deformation which is a reasonable assumption for the order of strains observed in these problems and solves "Nonlinear elasticity" equation. More information about the solid solver can be found in the *Elmer* overview and solver manuals (Elmer (2016)).

The details of the physics of FSI coupling and the preparation of input files for *Elmer*, *OpenFOAM*, and *ElmerFoamFSI* are provided in Section I and in the online documentation of the project. It is worthwhile to note that *ElmerFoamFSI* is implemented as a MPI parallel program. We judiciously made this choice as both *OpenFOAM* and *Elmer* projects use the same parallelization scheme and it was the best opportunity to test all the facilities of *IMPACT* in coupling parallel codes.

4.5.1 Verification: Static Problem

Problem Description: The problem selected to verify *ElmerFoamFSI* simulates the interaction between a static viscous fluid with a beam structure. The fluid domain is a square subjected to a uniform pressure. The boundary conditions are selected in a such way to minimize fluid velocity and motion on the side and top walls. The bottom wall consists of a cantilever beam (structures component) attached on one side but free to move in the y-direction elsewhere. Since there is no fluid velocity, the only contributing factor affecting the beam displacement is due to the pressure from the fluid.

Considering the fact that traction vector has both normal and tangential components, in the simple verification problem the normal component will be the major contributing factor. Therefore, this first problem verifies the pressure-driven component of the total tractions being applied to the beam and ensures it is passed accurately to the structure during coupled FSI simulation. Figure 19 shows the geometry of the problem.

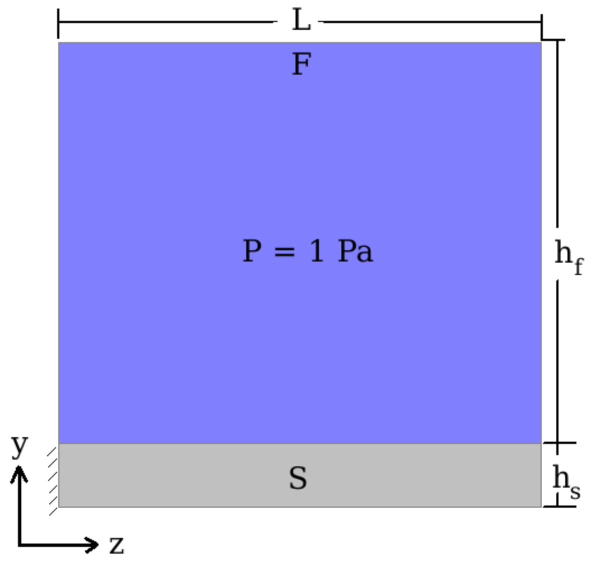


Figure 19: Domain and boundary conditions for static verification problem.

The dimensions of the domain are $L = 1.0\,\text{m}$, $h_s = 0.1\,\text{m}$, and $h_f = 1.0\,\text{m}$. The out-of-plane dimension is selected to be 0.1 m for both domains, arbitrarily. The velocity is initialized to 0.0 m/s throughout the fluid domain and the pressure given a uniform value of 1.0 Pa. Table 6 summarizes the properties of the solid and fluid used in the simulation. No changes are made to any of the initial conditions throughout the simulation other than allowing the problem to reach a steady state. The total simulation time is selected to be $t = 5.0\,\text{s}$.

Table 6: Properties of fluid and solid domains.

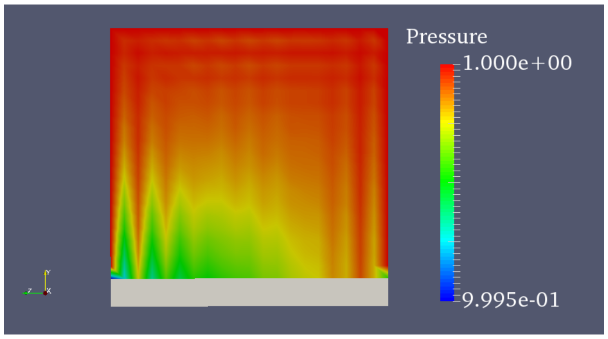
Property	Value	Units
ν_f	1.0×10^{-3}	N s/m^2
ρ_f	1.0	kg/m^3
E_s	1.4×10^6	Pa
ν_s	0.4	
ρ_s	10	kg/m^3

Exact Solution: Based on Euler-Bernoulli beam theory for small deformations, the exact value for the maximum deformation for the tip of the beam in the y-direction is

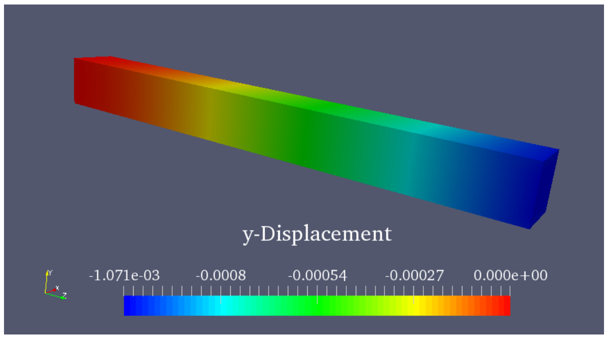
$$\delta_y = \frac{\omega L^4}{8EI}, I = \frac{bh^3}{12} \tag{4}$$

where $b = h = 0.1\,\text{m}$ for the depth and height of the beam, $\omega = 0.1\ N/m$ for the pressure of the fluid, $L = 1.0\,\text{m}$ for the length of the beam, $E = 1.4 \times 10^6\,\text{Pa}$ as Young's modulus for the beam, and I is the moment of Inertia for the cross section of the beam. Using these values, the maximum deflection in the y-direction is calculated as 1.071×10^{-3} m.

***OpenFOAM* Simulation:** As discussed above, for the verification this test problem is first solved using sole *OpenFOAM* solver icoFSIElasticNonLinUL. Figures 20a and 20b show contour plots for the pressure and y-displacement, respectively, for $t = 5.001$ s. This solver predicts a maximum absolute y-displacement of $\delta_y = 1.071 \times 10^{-3}$m for the cantilever.



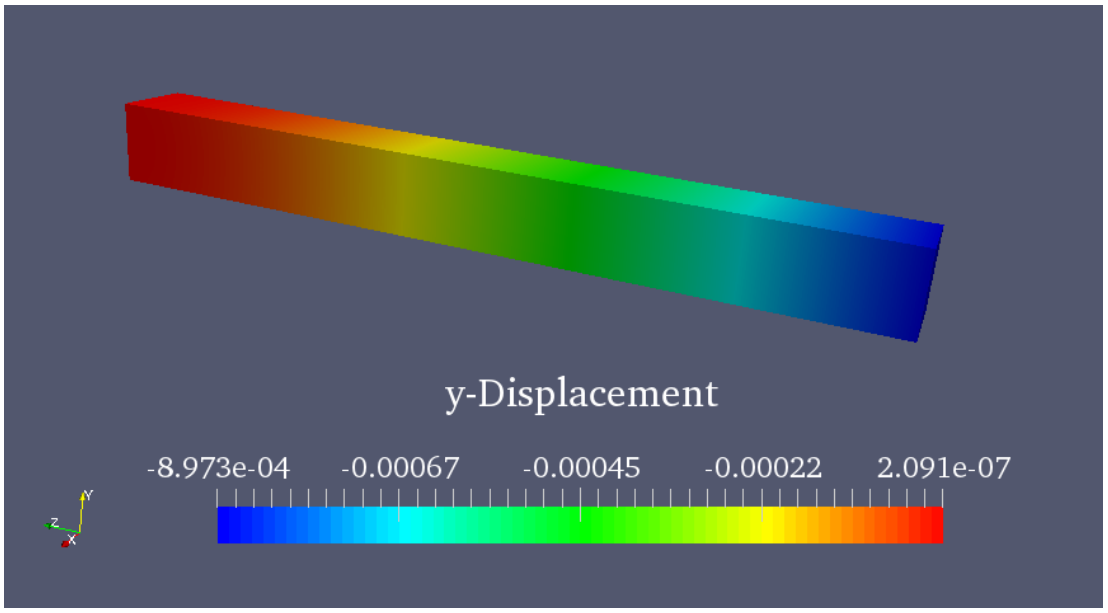
(a) Contour plot of pressure (Pa) in the fluid domain at $t = 5.001$ s. The small variations result from the incompressible fluid reacting to the movement of the beam, and the vertical "streaking" is a numerical artifact from the course discretization and the visualization software.



(b) Image of the y-direction displacement of the beam at $t = 5.0$ s. (Only the beam is shown.)

Figure 20: *OpenFOAM* simulation results for simple static problem.

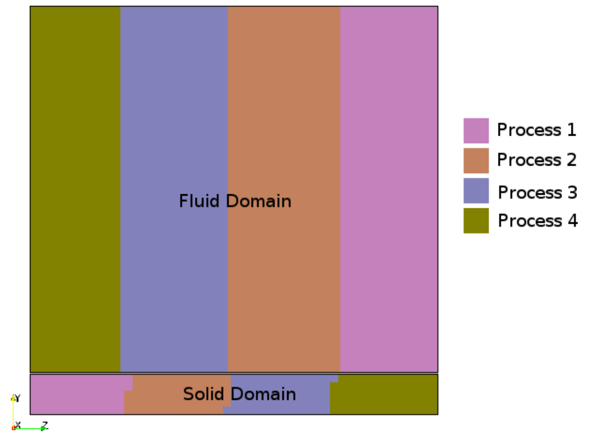
Serial *ElmerFoamFSI* Simulation: In the next step, the problem is solved using the *ElmerFoamFSI*. The solid mesh for *Elmer* is generated using the *ElmerGrid* utility. Figure 21a shows the contour plot for the maximum displacement obtained by *ElmerFoamFSI* for $t = 5.0$ s. The maximum predicted absolute y-displacement at this time is $\delta_y = 8.973 \times 10^{-4}$m.



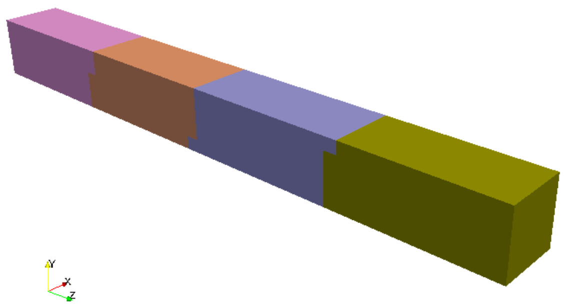
(a) Contour plot of the y-displacement field along the cantilever at time $t = 5.0$ s.

Figure 21: *ElmerFoamFSI* simulation results for simple static problem.

Parallel *ElmerFoamFSI* Simulation The static problem was also solved with parallel *ElmerFoamFSI* solver. Both solid and fluid domains were decomposed to a set of partitions for the parallel simulation. Figures 22 shows these partitions for $n_{cpu} = 4$ simulation. In the parallel simulation, *ElmerFoamFSI* registers information for each process separately and exchanges data between the processor as needed. The parallel simulation is performed using the same boundary conditions and the maximum deflection captured for the tip of the beam is $\delta_y = 8.870 \times 10^{-4}$m.



(a) Whole domain



(b) Beam domain

Figure 22: Parallel domain partitioning used in the parallel *ElmerFoamFSI* simulation of the static verification problem.

Results and Discussion: The exact solution for maximum absolute displacement is $\tilde{u} = 1.071 \times 10^{-3}$ m, the *OpenFOAM* solution is $u_{opf} = 1.071 \times 10^{-3}$ m, and the *ElmerFoamFSI* solution is $u_{elf} = 8.973 \times 10^{-3}$ m. With respect to the exact solution, *OpenFOAM* shows a relative error of less than $\varepsilon_{opf} = 0.04\%$ possible due to rounding error (but potentially less), and *ElmerFoamFSI* has a relative error of $\varepsilon_{elf} = 16.14\%$. Parallel *ElmerFoamFSI* also reproduces a similar values for the maximum tip deflection.

The error in the *ElmerFoamFSI* solution is acceptable and understandable within the context; the key difference between the two algorithms is the use of sub-iterations between timesteps in the pure *OpenFOAM* case. The lack of sub-iterations in the coupled *Elmer/OpenFOAM* case makes it much harder to achieve accurate results. In an effort to obtain the best result possible, the grid was refined and the timestep reduced to as small a step as sustainable for the algorithm. We have concluded that 16.14% is acceptable considering the lack of sub-iterations. Additionally, it was determined that since the objective of this exercise was to prove feasibility of coupling two disparate applications, that implementing a sub-iteration algorithm in the coupling schemed was a lower priority.

4.5.2 Verification: Dynamic

Problem Description: In the next step of verification, a more dynamic problem is used to test the accuracy of *ElmerFoamFSI* in transient problems. In this problem, a viscous fluid (F) is subjected to a velocity profile to generate a simple in-plane traction on the walls of the channel. One of the walls is a cantilever beam (S). Friction-based traction is applied to the top face of the cantilever to create axial deformation (δ_z).

The boundary conditions are selected such that only axial deflection is allowed for the beam, therefore, the effect of the vertical deflection created by potential normal pressure is negligible and only tangential components of the traction vector contribute to the deflection. The setting of the problem is necessary to test the accuracy of the tangential traction and displacement exchanges. Figure 23 shows the geometry of the problem.

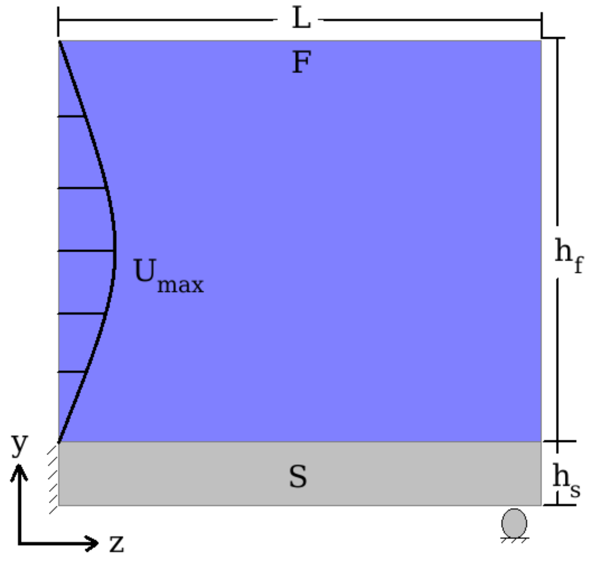


Figure 23: Domain and boundary conditions for dynamic simple verification problem.

The dimensions of the domain are $L = 1.0m$, $h_s = 0.01m$, and $h_f = 0.9m$. The velocity profile is symmetric with $U_{max} = 1.5\ m/s$ with axis of symmetry at $y = 0.46m$. The out-of-plane dimension is selected to be $0.1m$ for both domains, arbitrarily. Table 7 summarizes the properties of the solid and fluid used in the simulation. The velocity profile is ramped from zeros to its full developed form within the transition period of $T_{trans} = 0.1s$ and then kept constant for the rest of simulation. The total simulation time is selected to be $T = 5.0s$.

Table 7: Properties of fluid and solid domains.

Property	Value	Units
ν_f	1.0×10^{-3}	N s/m^2
ρ_f	1.0	kg/m^3
E_s	1.4×10^6	Pa
ν_s	0.4	
ρ_s	10.0×10^{-3}	kg/m^3

Exact Solution: The exact value for the maximum axial displacement for this problem can be simply calculated. For this, traction (τ) applied to the beam is given by

$$\tau = \mu \frac{\partial U}{\partial y}|_{y=0} = 6.\bar{6} \times 10^{-3}\ Pa. \quad (5)$$

We then integrate the total displacement along the beam to achieve

$$\delta_z = \int_0^{1.0} \varepsilon_{zz}(z)dz = 2.3807 \times 10^{-7}\ m. \tag{6}$$

***OpenFOAM* Simulation** The simulation is first performed solely by the FSI solver of the *OpenFOAM* suite. Figure 24 shows the contour plot for the pressure and velocity for $t = 5.0s$ and 100×100 discretization and 100×2 discretization for fluid and solid domains, respectively. This solver predicts a maximum axial displacement of $\delta_z = 2.194 \times 10^{-7} m$ for the cantilever.

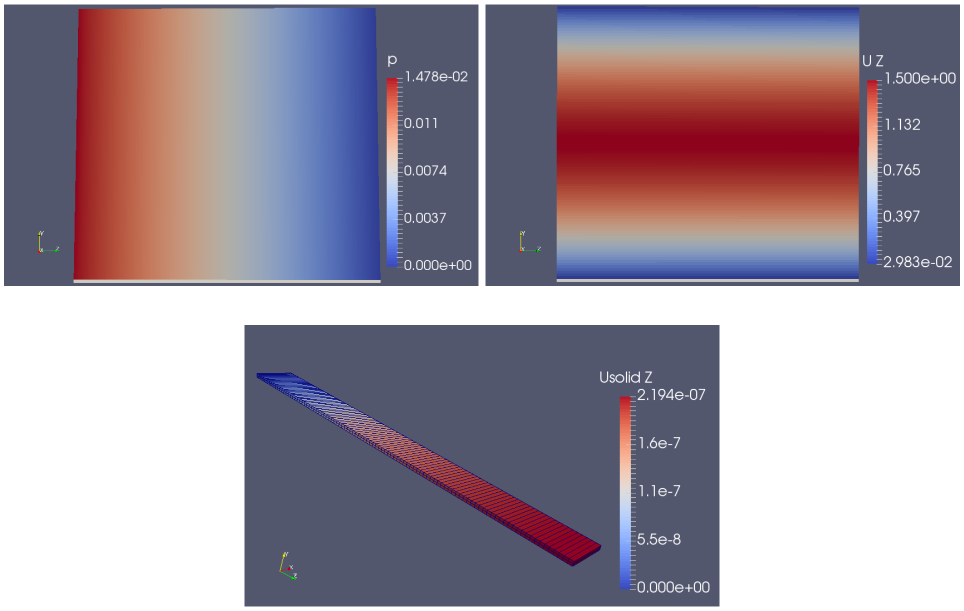


Figure 24: *OpenFOAM* simulation result for $t = 5.0s$ showing (a) pressure (Pa), (b) velocity (m/s) and (c) displacement contours (m).

***ElmerFoamFSI* Simulation** In the next step, the problem is solved with *ElmerFoamFSI*. The solid mesh for *Elmer* is generated by *ElmerGrid* and matches with the fluid grid used by *OpenFOAM*. Figure 25 shows the contour plot for the maximum displacement obtained by the *ElmerFoamFSI* for $t = 5.0s$ and the same discretization. The maximum displacement obtained in the axial direction was $\delta_z = 2.098 \times 10^{-7} m$.

Parallel *ElmerFoamFSI* Simulation The same simulation was carried on with the parallel *ElmerFoamFSI* using $n_{cpu} = 4$ processes. The total displacement captured in the axial direction was $\delta_z = 2.001 \times 10^{-7} m$.

Results and Discussion The exact formulation of the simple dynamic problem reports $\tilde{u} = 2.381 \times 10^{-7} m$ for the axial deformation, *OpenFOAM* solution is $u_{opf} = 2.194 \times 10^{-7} m$, and *ElmerFoamFSI* solution in serial and parallel is $u_{elf} = 2.098 \times 10^{-7} m$ and $u_{elf} = 2.001 \times 10^{-7} m$, respectively. In comparison with the exact solution, *OpenFOAM* shows a relative error of $\varepsilon_{opf} = 7.5\%$, and *ElmerFoamFSI* has a relative error of $\varepsilon_{elf} = 12.0\%$. The errors for both *OpenFOAM* and *ElmerFoamFSI* are acceptable as

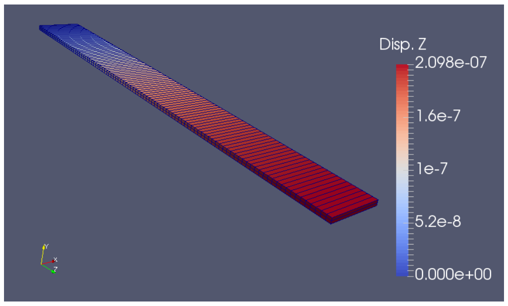


Figure 25: *OpenFOAM* solution for the displacement field along cantilever.

- the coupling scheme used for the problem is a non-iterative scheme (and therefore is not fully energy-conservative),
- numerical errors associated to the temporal and spatial discretizations used,
- iterative nature of the linear system solver and the fact that without a proper pre-conditioner, iterative system solvers used in parallel *OpenFOAM*, *Elmer*, and *ElmerFoamFSI* can be inaccurate,
- numerical truncation errors,
- oscillatory behavior of the solution.

If we accept the *OpenFOAM* solution as reference (to cancel out iteration errors), the relative error for the serial and parallel *ElmerFoamFSI* will be $\varepsilon_{elf} = 4.3\%$ and $\varepsilon_{elf} = 8.8\%$, respectively.

4.5.3 Verification: Hron-Turek Problem

Problem Description A series of fluid-solid interaction problems are described by Turek and Hron (2006). These fluid-solid interaction problems are designed based on actual experiments and are well accepted as the benchmark for validating a FSI solver. Here, we use one of these problems to validate the *ElmerFoamFSI* code.

Figure 26 illustrates the configuration of the problem: in this 2D problem, laminar incompressibe flow (F) is entering a channel from its left side with a fully-developed (after a small transition period) parabolic velocity profile. A cylinder with a cantilever beam attached (S) is submerged in the fluid close to the entry of the channel. Traction loads generated by fluid motion/friction develop on the faces of the cylinder and the cantilever, which causes the beam to deform and oscillate. In this problem, a self-induced oscillation is developed in the beam as it interacts with the fluid.

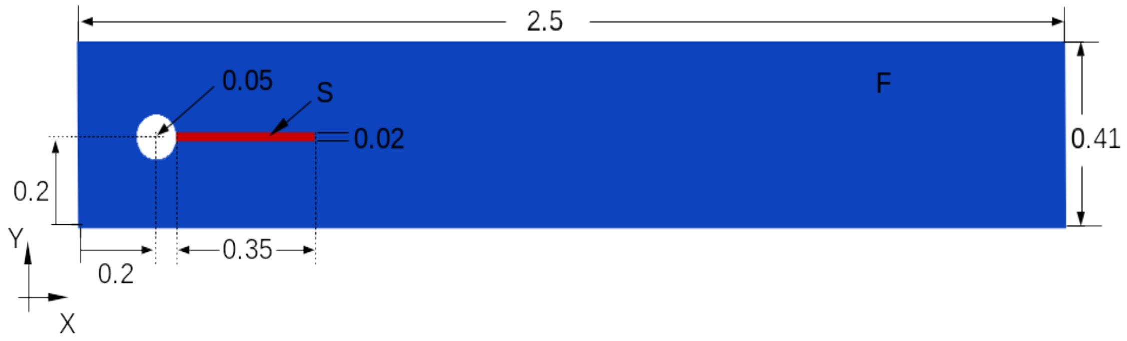


Figure 26: Domain and boundary conditions for Heron Turek verification problem.

We denote the deformation of the free tip of the beam by δ_y and recognize that the objective of this study is to see how accurately *ElmerFoamFSI* captures the variation of δ_y. Table 8 summarizes the properties of the solid and the fluid used in the simulation. The velocity profile is ramped from zero to its fully developed profile (with $\bar{U} = 2\ m/s$) in the transition period of $T_{trans} = 0.1\ s$, after which it is kept constant for the rest of the simulation. Further details about the boundary conditions and the setup of the Hron-Turek problems are provided in Turek and Hron (2006).

Table 8: Properties of fluid and solid domains.

Property	Value	Units
ν_f	1.0×10^{-3}	N s/m^2
ρ_f	1.0×10^{3}	kg/m^3
E_s	5.6×10^{6}	Pa
ν_s	0.4	
ρ_s	1.0×10^{3}	kg/m^3

Benchmark Solution This problem does not have an exact solution: a reference solution provided by Turek and Hron (2006) will be used for the comparison purposes.

ElmerFoamFSI Simulation In the next step, the problem is solved with *ElmerFoamFSI*. *ElmerFoamFSI* utilizes *OpenFOAM* for the fluid and *Elmer* for the solid calculations. The tractions, displacement, and velocities are exchanged between the solvers through *IMPACT*. *Elmer* as the structures agent, solves for non-linear elastic deformations as the order of strains developed in the beam are large. The solid mesh for the *Elmer* agent is generated by the *ElmerGrid*. Figure 27 shows the contour plot for the time-varying fluid velocity, pressure, and the displacement induced on the tip of the beam and captured by the *ElmerFoamFSI*.

Parallel *ElmerFoamFSI* Simulation The same problem was solved using parallel *ElmerFoamFSI*. The simulation domain for both fluid and solid were divided into $n_{cpu} = 4$ partitions. Figure 28 illustrates the partitioning of the simulation domain. It can be observed that all four processes used in the solid solver have fluid-solid interfaces, but only two processes share these boundaries in the fluid domain. In general, *IMPACT* very well handles such mismatches between domain decomposition which is enforced by the participating solvers and exchanges information (in parallel) between the processes as needed.

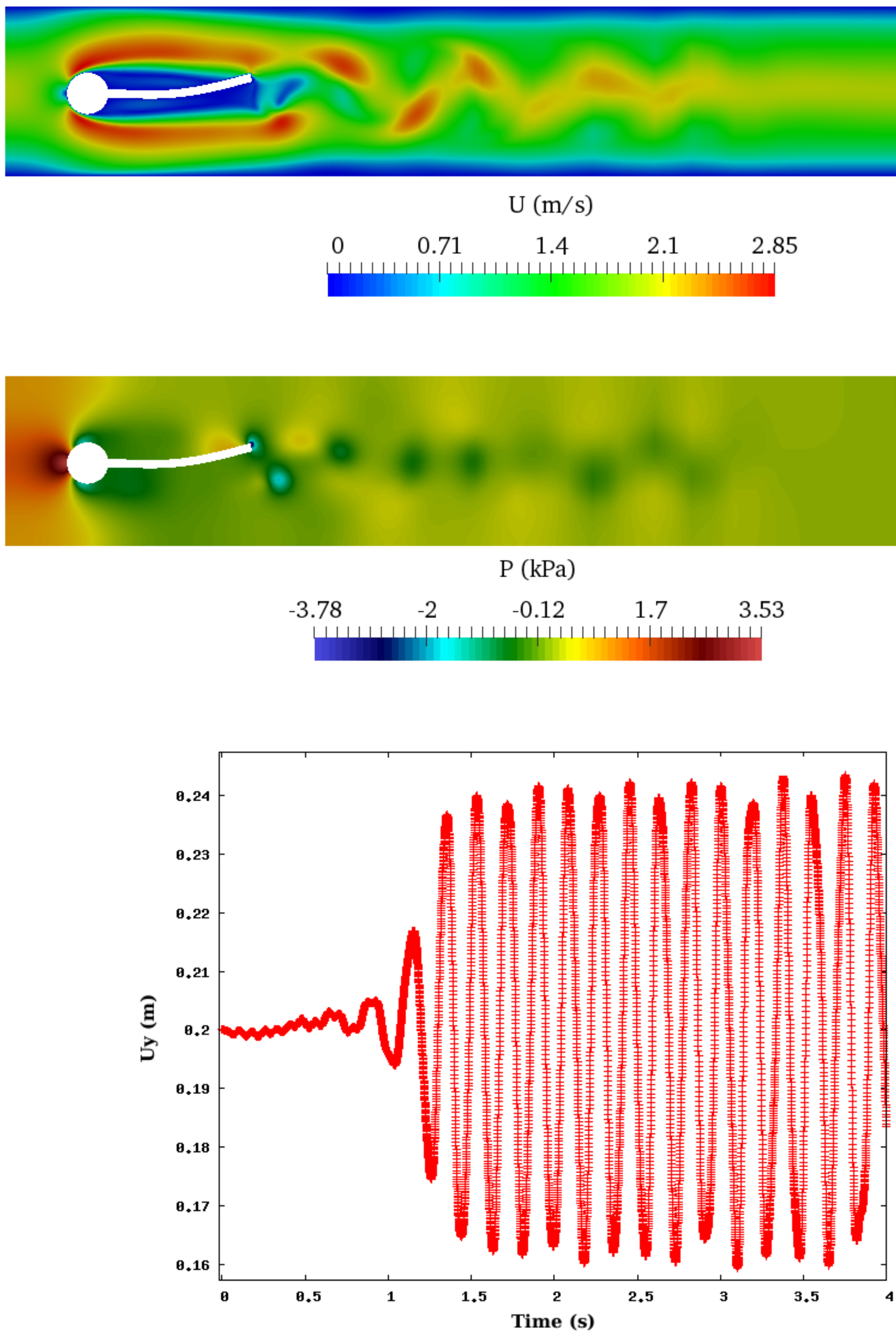


Figure 27: *ElmerFoamFSI* simulation results showing velocity contour (top), pressure contour (middle) for $t = 5.0s$, displacement history for the tip of the beam (bottom).

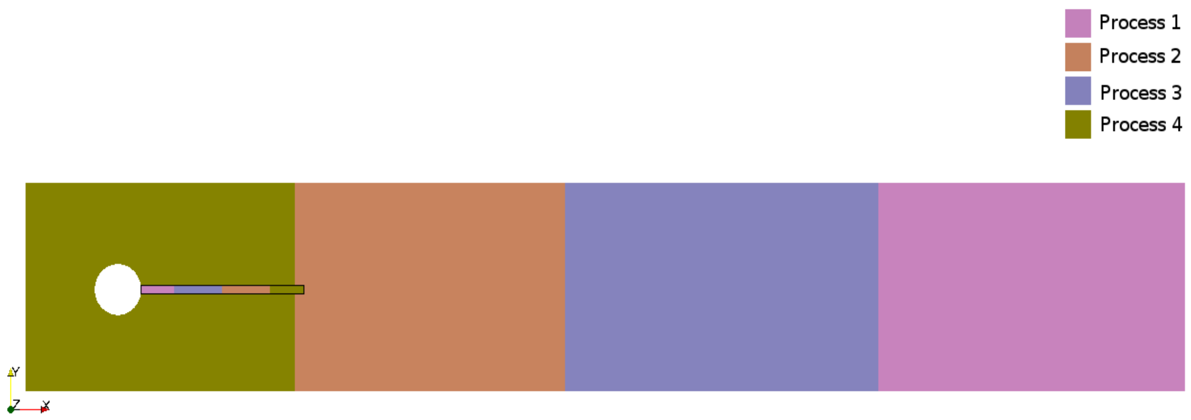


Figure 28: Partitioning of the simulation domain for parallel Heron Turek verification problem.

The displacement for the tip of the beam was also captured by the parallel *ElmerFoamFSI*.

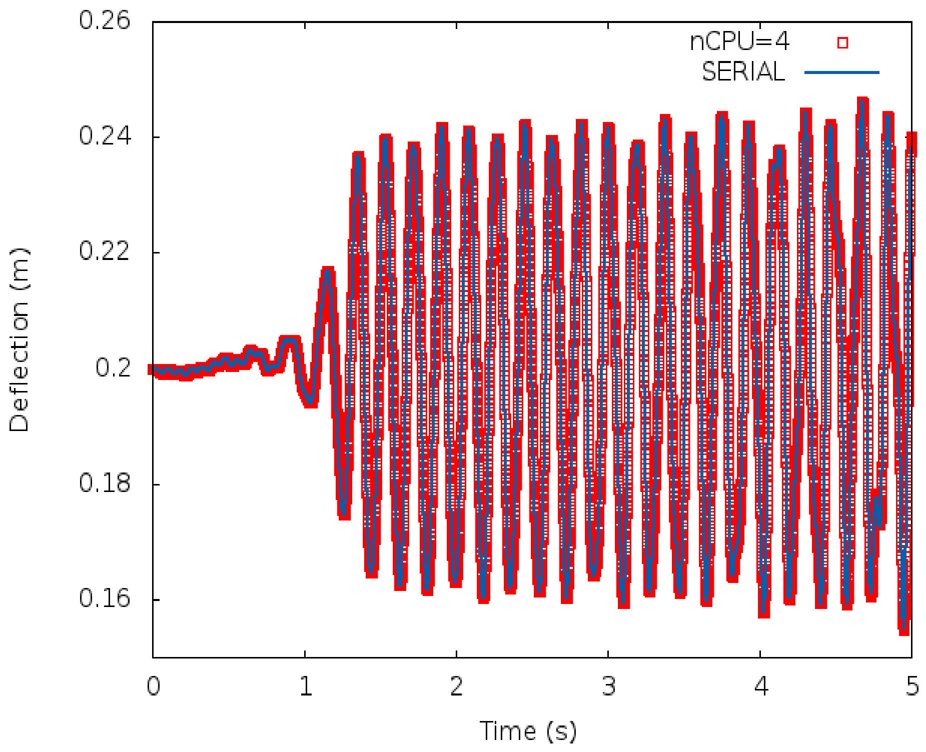


Figure 29: Comparison between serial and parallel *ElmerFoamFSI* predictions for the displacement of the tip of the beam.

Results and Discussion *ElmerFoamFSI* simulation results were compared against those reported in Turek and Hron (2006). The benchmark values for the amplitude and period of the beam deflection δ_y are about $0.035m$ and $0.18s$, respectively. *ElmerFoamFSI* predictions for these quantities were $0.038m$ and $0.18s$, matching the reference values. The error for *ElmerFoamFSI* correspond to:

- numerical errors associated to the temporal and spatial discretization,
- simple non-iterative nature of the solution scheme.

4.5.4 *ElmerFoamFSI* Scaling Study

To measure parallel performance of *ElmerFoamFSI* a scaling study was performed. HronTurek and simple dynamics problems were solved using $N_{cpu} = 1 - 64$ cores. Figure 30-a shows the parallel efficiency for HronTurek problem. In this unbalanced problem the parallel efficiency quickly vanishes. By looking at Figure 30-c it can be observed that for HronTurek problem most of the computation time is spent on *OpenFOAM* solver and therefore the overal performance for *ElmerFoamFSI* is dominated by the performance of this module. Figure 30-b depicts similar efficiency curves for simple dynamic problem. This time again the problem is not well balanced as most of the CPU time is spent in the *Elmer* module, and therefore the performance of *ElmerFoamFSI* is dominated by this module. From this quick study it can be concluded that the parallel efficiency of an *IMPACT*-enabled code is very much related to the performance of its modules, for the case of *ElmerFoamFSI* to *Elmer* and *OpenFOAM* as in a multiphysics problem most of the computation will be happening in the actual physics solvers. Furthermore, Figure 30-c,d indicate that *IMPACT* contributes a negligible overhead to the total computation time.

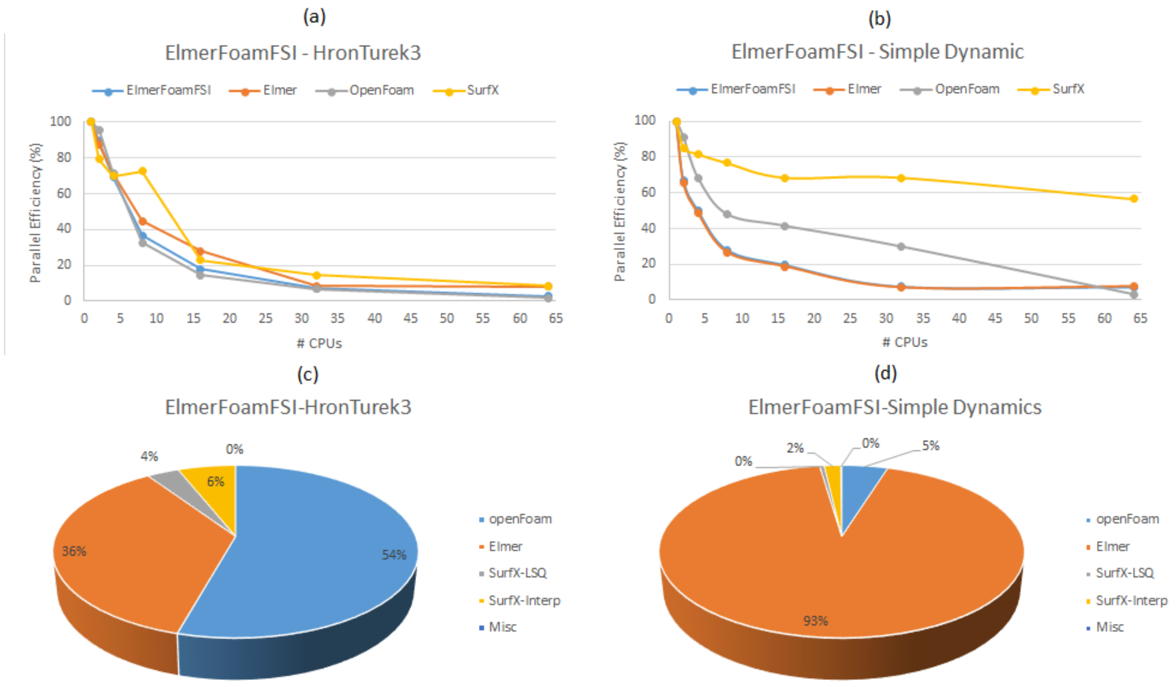


Figure 30: *ElmerFoamFSI* scaling study: parallel efficiency and CPU time partitions for HronTurek problem (a,c) and simple dynamic problem (b,d).

4.6 *Rocstar Multiphysics* Testing

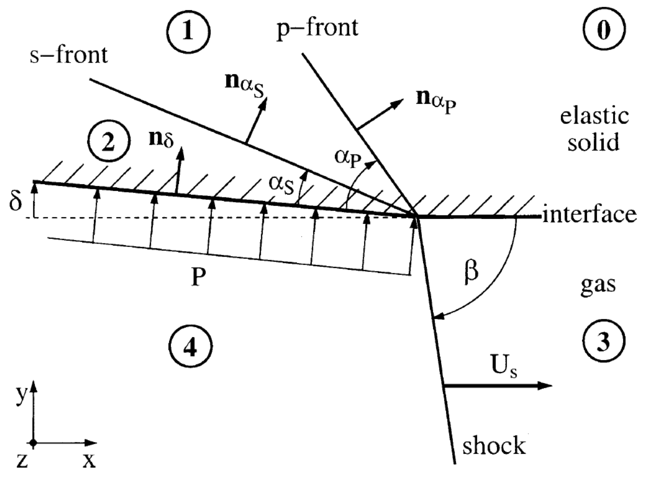
In this section we report the verification tests performed for *Rocstar Multiphysics*. As mentioned earlier, *Rocstar Multiphysics* comes with two compressible fluid dynamics solvers (*Rocflu* and *Rocflo*) and a structures solver (*Rocfrac*). We also have *Elmer* and *OpenFOAM* integrated into *Rocstar Multiphysics*.

We note that all of these modules are MPI-enabled and are fully compatible with *IMPACT*. Testing all of the combinations that can be achieved from these five uniphysics solvers is beyond the scope of this report; here we have limited our analysis to two cases. In the first case, we combined *Rocflo* and *Elmer* to solve a shock propagation compressible flow FSI problem. In the second case, we solve the Hron-Turek problem with the combination of *Rocfrac* and *OpenFOAM*.

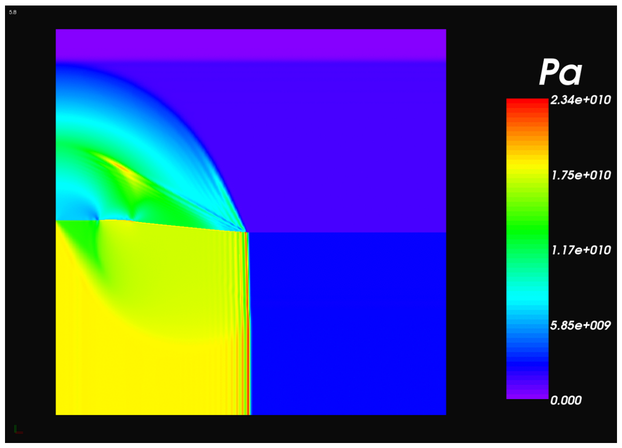
4.6.1 *Rocflo/Elmer* Combination

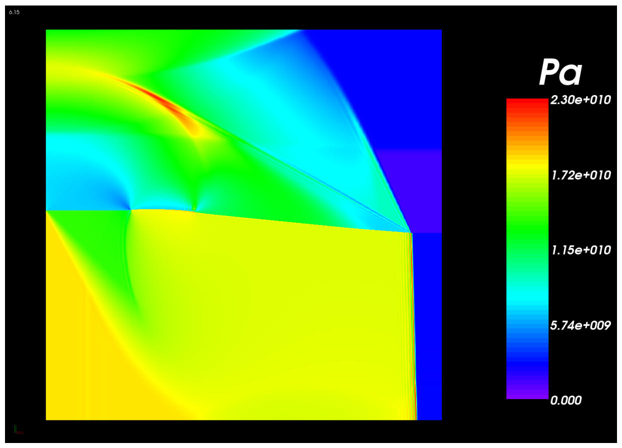
The super-seismic shock example illustrates a problem with aero-elastic FSI. The shock in the compressible fluid travels at a speed that exceeds the dilational wave speed in the solid, causing deformation of the solid. The solid material properties of this example are similar to that of copper. The fluid properties are modified to achieve a sound speed similar to the pressure coefficient (c_p).

This problem can be run on any number of processors and couples the *Rocflo* and *Rocfrac Rocstar Multiphysics* modules. Figure 31 shows images pertaining to the setup and running of this problem. The verification data for this problem are provided by Arienti et al. (2003) and Jaiman et al. (2004).



(a) Illustration of super-seismic step load showing the p and s wave systems (above) and the oblique shock (below).





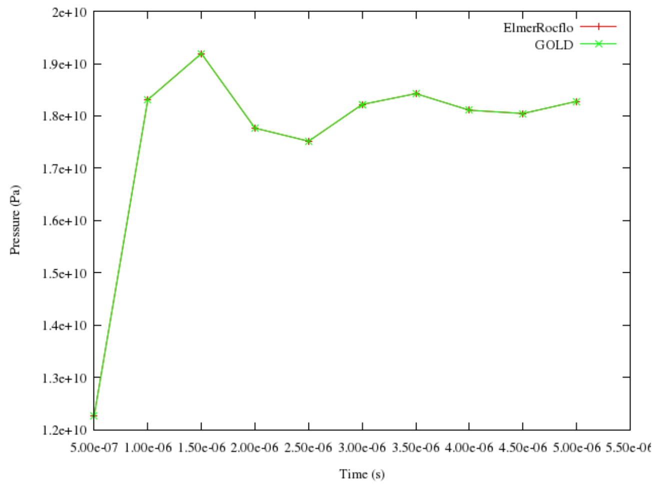
(b) Shock wave partially through domain depicted using *Rocketeer*.

(c) Shock wave almost entirely through domain depicted using *Rocketeer*.

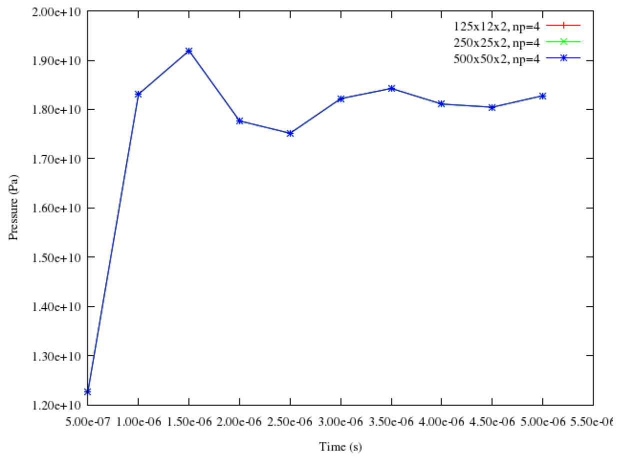
Figure 31: Images from the super-seismic shock example case.

The problem was solved by *Rocflo* and *Elmer* combination. Figure 32 summarizes the verification study outcomes. The solution obtained by *Rocflo/Elmer* combination is compared against that of *Rocflo/Rocfrac* as gold standard. The problem is solved for different mesh densities and with different number of processes. The results match with the gold standard, proving the proper implementation of the modules provided with

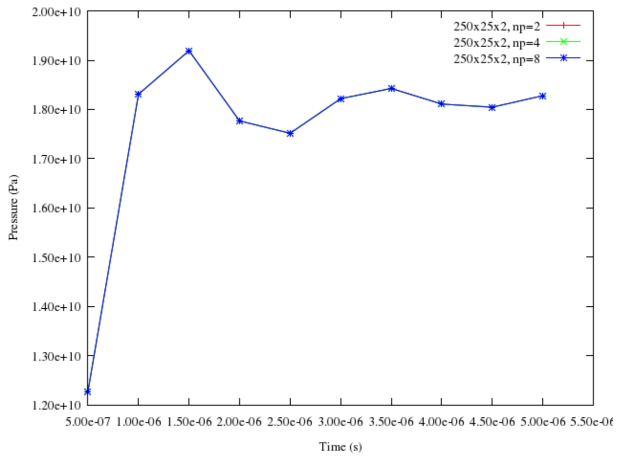
Rocstar Multiphysics and proper functioning of *IMPACT*. This test has been added to the testing framework of *Rocstar Multiphysics* and all input files and settings for the problem are accessible from our public repository.



(a) Comparison between *Elmer/Rocflo* solution and *Rocflo/Rocfrac* (GOLD) solution both obtained by *Rocstar Multiphysics*.



(b) Solution obtained by *Rocflo/Elmer* combination for different mesh densities.



(c) Solution obtained by *Rocflo/Elmer* for different number of processors.

Figure 32: Super-seismic shock problem solved by different modules of *Rocstar Multiphysics* for the verification purpose.

4.6.2 *Rocfrac/OpenFOAM* Combination

The Hron-Turek problem described in Section 4.5.4 was solved in *Rocstar Multiphysics* using *Rocfrac/OpenFOAM* combination. The solids domain was discretized into a finite element mesh composed of tetrahedral elements. The same grid was used for the *OpenFOAM*. The simulation was performed for a few timesteps. Figure 33 compares *Rocfrac/OpenFOAM* solution with *Rocfrac/Rocflo* modules showing comparable solution.

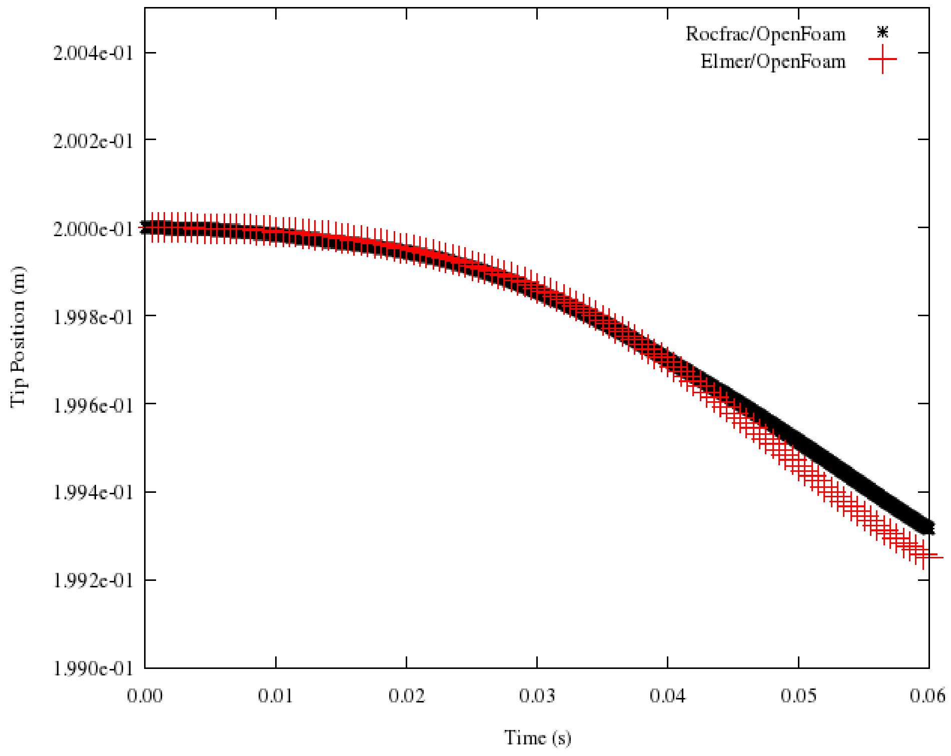


Figure 33: *Rocfrac/OpenFOAM*solution compared with *OpenFOAM/Elmer*for Hron-Turek FSI3 problem.

5 Community Involvement and Usage

5.1 Community Development

Community building is of utmost importance to the success of an open source package. The ultimate goals of standardization, community-driven design, and the nature of our particular, targeted user community make community building an important task to ensure eventual acceptance and success of the product. We need a representative sampling of the potential user-base for this infrastructure to ensure the relevance of its design and implementation. Our overarching community goal of this project is to build a community of stakeholders. During the Phase I and Phase II projects, we initiated the following conversations and contacts to discuss interest in and use of *IMPACT*.

Framework developer contacts

We reached out to developers and development groups of several other multiphysics packages, including *Sierra*, *LIME*, *Trilinos*, *Rocstar*, *PreCICE*, *MpCCI*, and *OpenFSI*. Several of these contacts resulted in meaningful exchanges of ideas.

Inter-institution meetings

We are members of several interdisciplinary government and industry organizations and have participated in several meetings and symposia attended by stakeholders in advanced manufacturing and engineering industries, academia, and government multiphysics endeavors (e.g., the National Center for Supercomputing Applications (NCSA) annual Private Sector Program (PSP) meeting, the Joint Insensitive Munitions Technology Program (JIMTP) review, and the Air Force Research Lab Solid Rocket Motor (SRM) Integrated Product Team Meeting (IPT)). Reception of this project has been quite positive, and we have developed promising relationships with a few companies that are interested in multiphysics capabilities based on *IMPACT* constructs.

Targeted contacts

We contacted, arranged meetings with, and presented our proposed multiphysics infrastructure to specific companies, addressing their Modeling and Simulation (M&S) development groups.

Open source distribution

We set up an open source project for distribution of the infrastructure developments. The project is currently hosted by SourceForge[*].

Consortium for Open Multiphysics

We branded our nascent stakeholders community as the "Consortium for Open Multiphysics." The Consortium mission statement and details can be found online at `http://www.openmultiphysics.org`. We envisioned that the Consortium would help drive the design for the infrastructure; however, it was discovered that most organizations, sans funding, did not have the resources to put into formal group participation for a tool that was under construction. For now, it is up to Illinois Rocstar to continue to generate new multiphysics applications to drive interest in the abilities embodied in *IMPACT*.

5.2 Interactions

To seed the Consortium participant pool, we directly engaged groups from several institutions with known multiphysics needs and interests. These institutions include Boeing, ATK, Aerojet, Caterpillar, Procter &

[*]See `http://www.sourceforge.net`.

Gamble, MSC, AMAX, University of Illinois, NCSA, and Sandia National Laboratory. We have also interacted with several development groups, including developers of *LIME*, *Trilinos*, *Precice*, and *Rocstar*. As members of several government and industry organziations, we have attended several inter-institution meetings, including NCSA's Annual Private Sector Partners (PSP) meeting, the Joint Insensitive Munitions Technical Program (JIMTP) meeting, and the Air Force/Industry SRM Integrated Product Team (IPT) meeting.

Among the groups we have engaged, the response to this project has been very positive. Everyone is interested, but many are reluctant to actively participate before seeing concrete applications of the technology. Most companies are exploring the use of open source simulation and scientific computing packages, but there remains some degree of concern surrounding the use of community-developed projects, their life expectancy, and availability of support. In many of our engagements, several discussions have been centered around "making a business case" for what many call "co-simulation," which seems to be a common terminology for multiphysics in industry. We must be sensitive to the potential reluctance among potential participants.

During the Phase II project, we continued to interact with the groups mentioned above, as well as initiating discussions with new organizations, such as the United Technologies Research Center in CT, the Missile Defense Agency, and further Air Force organizations. It became even more apparent that large organizations did not want to spend resources on a nascent, developing product until it was completed. Given that realization, we suspended attempts to initiate a consortium community around *IMPACT*, and changed tack to look for application and partnering opportunities to generate more example applications and "buzz". Section 5.3 discusses the successes to date in finding applications for *IMPACT*.

5.3 Follow-on *IMPACT* Projects

To drive *IMPACT* into use outside of Illinois Rocstar and start to build the envisioned community of users and possible coupling standardization, *IMPACT* is now being used in several current projects to further illustrate its ability to be applied in diverse use cases. Several of these new projects are listed below.

- *IMPACT* is being used to couple a DoD hydrocode to an existing Air Force simulation tool called *Endgame Framework*. The advanced parallel coupling facilities of *IMPACT* will allow construction of a remote-coupled parallel runtime facility for this hydrocode in Endgame Framework.
- *IMPACT* is being used on a NASA project to couple modules in a new Illinois Rocstar-produced advanced battery modeling tool. The *IMPACT*-enabled code will then be coupled with other tools from Idaho National Laboratory and Oak Ridge National Laboratory to produce an advanced tool for modeling performance and safety of advanced batteries.
- The *IMPACT*-enabed *Rocstar Multiphysics* is being used in a joint Illinois Rocstar - University of Alabama Huntsville (UAH) - Missile Defense Agency (MDA) project to couple a complex-flow lattice Boltzmann tool into *Rocstar Multiphysics* to model an apparatus of interest to MDA.
- *IMPACT* is being designed into the interface on another DOE project to couple certain chemistry and materials science tools together for exposure through the `chemistryhub.org` cloud computing interface.
- United Technology Research Center has been using a non-*IMPACT*-enabled version of the Rocstar code for over a year, and is interested in transitioning to *Rocstar Multiphysics* with the newly capable Elmer solid mechanics tool included through *IMPACT*. As new open-source tools are added to

the newly *IMPACT*-enabled *Rocstar Multiphysics*, a much wider base of multiphysics users will be developed.

- *IMPACT* is being used in a new lattice-Boltzmann-based chemical diffusion and reaction Air Force project to allow coupling into an existing Air Force toolchain in the next phase of that project.
- A current Illinois Rocstar DOE project concerning advanced grid/mesh data transfer and manipulation will result in new service modules to be exposed through *IMPACT*, and ultimately available to *Rocstar Multiphysics*, expanding it's use and reach.
- New projects continue to be proposed that use *IMPACT* in different ways.

The list above is just the beginning of the kinds of new coupled multiphysics applications that can be produced using *IMPACT*. Final documentation, the gitHUB-hosted downloadable *IMPACT* code, and an expanding set of example solutions will enable generating a more interested set of stakeholders, and ultimately the envisioned community of users.

6 Publications and Presentations

Since the beginning of the Phase II this project has been highlighted as a specific topic or as part of other topics in sessions at the following venues:

AFRL SRM M&S IPT Meeting – The *Solid Rocket Motor Modeling and Simulation Integrated Product Team* meeting hosted by the Air Force Research Lab, attended by several DOD prime contractors and other industrial and government organizations interested in solid propellant rocket propulsion. This project is of interest to the IPT community for its potential to enable rocket and rocket-system modeling and simulation.

DOE ASCR SBIR/STTR Workshop – The *Opportunities in Advanced Computing and Networking* hosted by DOE was attended by representatives from several small-to-medium advanced manufacturing and engineering businesses and small businesses interested in SBIR/STTR, ISVs, and DOE.

NASA SBIR Workshop – March 17 – 18, 2015, NASA Glenn Research Center. *IMPACT* was highlighted as an enabling technology for the Illinois Rocstar advanced battery modeling initiative during this discussion and presentation of NASA SBIR technologies to a broad industry and government audience.

Aerojet Rocketdyne – internal meeting to discuss possible use and support of *IMPACT* and *Rocstar Multiphysics*.

United Technology Research Corp – internal meeting to discuss possible use and support of *IMPACT* and *Rocstar Multiphysics*.

Proposals – Several unfunded proposed applications of *IMPACT* have been submitted to the Air Force and Army in addition to the funded initiatives discussed in Section 5.3.

6.1 Website(s) or other Internet site(s)

Illinois Rocstar has established a centralized *GitHub* site for dissemination of software related to this project (`https://github.com/IllinoisRocstar`). This new site is linked to `openmultiphysics.org` and provides a modern git-based repository.

The results, software, documentation, and community associated with this project will be maintained on `openmultiphysics.org` once an active community is built using the software. This site houses the Consortium for Open Multiphysics, documents its mission, and presents a set of multiphysics test cases. The site also provides a forum for user discussions, an issue ticketing system, and a link to the source code repository on *GitHub*.

6.2 Inventions, patent applications, and/or licenses

The *IMPACT* infrastructure is, by design, an open-source software system using a permissive non-copyleft license. As such, no patents or other restrictive intellectual property restrictions are implemented. Copyright for the software is retained by Illinois Rocstar, with the permissive license providing unlimited distribution rights if the license and copyright are maintained.

6.3 Other products

This project will result in a collection of publicly available descriptions of domain-specific problems for multiphysics V&V and a set of working multiphysics applications composed of available open uniphysics domain solvers targeting one or more of these V&V problems. The initial collection of these problems is available at OpenMultiphysics.org, and the software will become available at the time of *IMPACT* Release 1.

7 Conclusions and Path Forward

The project concludes with major successes in construction of the *IMPACT* infrastructure itself, establishment of two parallel multiphysics applications using *IMPACT*, and the creation of several new projects that are also using *IMPACT* to construct new multiphysics tools. Section 7.1 discusses *IMPACT* and the multiphysics tools and their documentation and Section 7.2 outlines the future of *IMPACT* and associated applications.

7.1 Accomplishments

IMPACT, two multiphysics applications: *ElmerFoamFSI* and *Rocstar Multiphysics* and their documentation comprise the major deliverables from this Phase II SBIR. These deliverables are discussed in Sections 7.1.1- 7.1.4.

7.1.1 *IMPACT*

As the major deliverable under this project, Illinois Rocstar presents *IMPACT*, an infrastructure for parallel software integration aimed at facilitating multiphysics simulation. The infrastructure is designed to provide a flexible mechanism for inter-application data exchange and function invocation for use in developing composite parallel software systems. A key objective is minimizing the cost of development and ownership of integrated capabilities by reducing the effort and depth of changes required for integration. It facilitates interoperability between different programming languages (in particular, C++ and Fortran 90) and enables "plug-and-play" of different physics and computer science capability implementations within an integrated module-based system.

The object-oriented design paradigm and component encapsulation allow for clean inter-application interfaces and maximize concurrency in development of different capabilities. A number of multiphysics-specific capabilities are provided by the infrastructure as reusable service utility modules. This design and set of capabilities provide a rich environment that allows for rapid prototyping of software systems composed of multiple integrated applications; it also supports a multitude of architectures for the integrated software system. The software implementation of the infrastructure has been released under an OSI-approved open source license and is distributed through a github repository, `https://github.com/IllinoisRocstar/IMPACT`.

In addition to the working implementation, an abstract model for the infrastructure has been developed and presented. This model is intended for use as an abstract framework around which to discuss standardization and requirements for the infrastructure. We assert that standardization and protocols for low level software integration constructs and procedures will greatly increase utilization of HPC M&S among the nation's advanced manufacturing and engineering industries.

Furthermore, standardization allows for development and deployment of couple-ready capabilities that can be readily leveraged in producing composite software systems. The economic implications and range of commercial applications for this technology are promising: integration-ready software encapsulating couple-ready models are the future of HPC-based M&S and a key to maintaining national competitiveness.

In Appendix A, we have provided a more in-depth document that can be used as a stand-alone reference for the *IMPACT* concept. All modules and services provided in *IMPACT* are also documented in Appendices B to H.

7.1.2 *ElmerFoamFSI*

ElmerFoamFSI is the second product delivered under this project and is our first application program build based on the constructs provided by *IMPACT*. This product couples two well known computational physics codes, *Elmer* and *OpenFOAM*, to provide a standalone FSI solver. *OpenFOAM* is an open source computational fluid dynamic (CFD) code widely adopted by researchers across most areas of engineering and science. The code has a long list of features and facilities to solve for anything from fluid dynamics to heat transfer and solid mechanics, and contains an internal fluid-solid (or fluid-structures) module to solve for problems involving interaction between fluid and structures. *Elmer* is another well-known open source finite element analysis (FEA) software with a sizable user base. Similar to *OpenFOAM*, *Elmer* has physical models and capabilities to simulate a wide range of problems including structural mechanics, fluid dynamics and electromagnetics. For further information about these codes, we refer reader to the comprehensive documentation of these codes.

The *ElmerFoamFSI* project comes with several FSI simulation examples to test its functionality and verify its action. The setup of these examples are referenced in the Section I and within the online documentation of the project. The mathematics of FSI coupling was also researched and is provided in Appendix J.

7.1.3 *Rocstar Multiphysics*

Another product under this project is our rebuilt *Rocstar* now known as *Rocstar Multiphysics*. The *Rocstar Multiphysics* is another full spectrum example of *IMPACT* application. *Rocstar Multiphysics* includes several complementary finite-volume compressible flow solvers that are formulated on moving meshes (ALE) to handle geometric changes. The modules included, *Rocflo* and *Rocflu* are general use CFD solvers, both of which have strengths for specific problems. *Rocflo* uses either a centered or an upwind scheme with Roe flux splitting on multi-block structured meshes. *Rocflu* operates on unstructured tetrahedral or mixed-element meshes and employs a novel, high-order WENO-like approach with the HLLC scheme to handle strong transient flows including shocks. Further, the fluid solvers are integrated with additional physics modules for simulating turbulent and multiphase fluid flows.

Aside from its compressible-flow solvers, *Rocstar Multiphysics* is also equipped with a finite-element structure solver called *Rocfrac*. *Rocfrac* uses an explicit finite-element scheme and supports different element shapes including tetrahedral and hexahedral elements and their 2D analogs. *Rocfrac* is also capable of accounting for moving meshes (ALE), material, and geometric non-linearities. On top of fluid and structures modules, *Rocstar* is also equipped with a module to account for solid-fuel burning during simulation. This module is called *RocburnAPN*. Finally, *Rocstar Multiphysics* also comes with service modules for mesh refinement, solution transfer and many others.

We have also integrated *Elmer* and *OpenFOAM* into *Rocstar Multiphysics*; it is important to note that all of these modules are MPI-enabled and are fully compatible with *IMPACT*. The final test for *Rocstar Multiphysics* was the combination of *Elmer*/*Rocflo* to solve a shock propagation problem in a compressible flow and *OpenFOAM*/*Rocfrac* to solve the Hron-Turek problem. Aside from these tests, a series of unit and regression tests are provided in *Rocstar Multiphysics* which verify the accuracy of the product and can be used as a reference of how to use the code.

7.1.4 Documentation

IMPACT is a complex multiphysics code integration infrastructure and its use requires good documentation. Emphasis has been placed on producing comprehensive documentation during the project and the combination of the web-based documentation for *IMPACT*, *Rocstar Multiphysics*, and *ElmerFoamFSI* and the guides in the appendices to this document provide a solid base for outside organizations to become familiar with and use *IMPACT* and/or the multiphysics tools for their own purposes. In addition, we supply a test infrastructure (IRAD) and many test problems for users of the infrastructure to base robust unit and regression testing protocols on.

7.2 Next Steps

As described in this report, the technical portions of this project have been successful. The *IMPACT* system is complete, tested, available, and documented. Two pertinent parallel multiphysics capabilities have been constructed using *IMPACT*, and are now available to the community. All of these capabilities are open source and available from gitHUB.

As discussed in Section 5, building a community of users around a developing product was not a successful endeavor. We talked with many groups and individuals, many of whom showed interest in the technology during the project. However, providing their own resources to attend meetings and discussions concerning their needs from the technology was difficult to maintain. Thus, now that the technology is complete and available, a different tack has been chosen: to search out and produce as many multiphysics capabilities as possible, adding additional physics modules to *Rocstar Multiphysics* as makes sense, and to build a community around the developing multiphysics ecosystem instead of only around the *IMPACT* infrastructure. This will provide potential users with the ability to connect their own tools together using *IMPACT* the use of a burgeoning set of already *IMPACT*-enabled physics modules; and the *Rocstar Multiphysics* system to start from, extend, or learn from as needed.

In Section 5.3, a list of current *IMPACT*-oriented initiatives is provided where Illinois Rocstar is using the completed *IMPACT* infrastructure to build tools for others in both government and industry. Some rely upon *IMPACT* and *Rocstar Multiphysics*, while others are new developments using *IMPACT* alone. We are working on adding advanced lattice Boltzmann modeling techniques, modern mesh data transfer, adaptation and remeshing capabilities, hydrodynamic FEM modeling and diffusion and chemical reaction models to the *IMPACT*-enabled set of available tools. Some of these can be made available open-source in the expanding *Rocstar Multiphysics* base, while others can only be made available through private or government channels (e.g., the Air Force hydrodynamics code). All, however, will provide bases for future multiphysics code development in commercial or government projects.

The next steps to be taken to expand the user base for *IMPACT* and the *Rocstar Multiphysics* system built around it are to:

- Continue to look for more applications to build using *IMPACT* by responding to RFPs, interacting with technical and business contacts, and beginning a marketing program for *Rocstar Multiphysics*.
- Attend appropriate conferences and technical meetings. Produce and present papers and presentations at conferences and begin to exhibit at conferences and trade shows.
- As community size grows, assess point at which holding user group conferences/meetings/webinars online is viable.
- Continue to make it easier to use *IMPACT* for code coupling. Currently, “wizards” to query the user for information about the codes to be coupled, producing the skeleton for the coupling objects based on input information is envisioned. Ultimately, a graphical programming system to facilitate coupling is planned.

- Continue to expand *IMPACT* service module capabilities, by integrating Illinois Rocstar and other open-source tools into the package.
- Produce a series of training webinars to augment the online and .pdf documentation.
- Integrate portions of *Rocstar Multiphysics* into other commercial systems, potentially through CAD System plugins or other methods provided by broadly accepted modeling and simulation tools.

Broadly, these next steps are designed to get the system into as many hands and use-cases as possible to expand the applicability and acceptance of the open source system. That is the first step to building an interested community, at which point discussions of standards and compliance will be useful. Illinois Rocstar is dedicated to producing the most useful open source multiphysics tools and solutions possible, and will leverage the results of this SBIR to the maximum extent possible to improve access to multiphysics modeling and simulation throughout the US industrial and government sectors.

A IMPACT Core Domain Model

Version 1.0.0

IllinoisRocstar LLC

October 25, 2016

License

The software package sources and executables referenced within are developed and supported by Illinois Rocstar LLC, located in Champaign, Illinois. The software and this document are licensed by the University of Illinois/NCSA Open Source License. (See `opensource.org/licenses/NCSA`.) The license is included below.

For more information regarding the software, its documentation, or support agreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

List of Acronyms

Term	Description
1-D or 1D	one-dimensional
2-D or 2D	two-dimensional
3-D or 3D	three-dimensional
API	application programming interface
CFD	computational fluid dynamics
CSC	component-side client
CSM	computational structural mechanics
CI	component interface
FSI	fluid–structure interaction
HPC	high performance computing
IMPACT	*Illinois Rocstar Multiphysics Application Coupling Toolkit*
I/O	input/output
NCSA	National Center for Supercomputing Applications
MI	model interface
MPI	message passing interface
M&S	modeling and simulation
SIM	Software Integration Manager
SIT	Software Integration Toolkit
V&V	verification and validation

A.1 Introduction

This document serves to define the core domain model of the *Illinois Rocstar Multiphysics Application Coupling Toolkit* (*IMPACT*) software. It aims to provide users and developers with a high level description of *IMPACT*ś purpose and functionality. The idea is that as *IMPACT* is further developed and changed, this document will serve as a basis and a guide to sustain the original vision of the software. The broad overarching goal for *IMPACT* is to be an infrastructure wherein multiphysics technologies can be rapidly conceived, deployed, studied, and ultimately utilized, all by the domain scientist(s) seeking to accomplish the simulation. In essence, it is software to assist in simulating multiphysics problems by integrating other software.

In multiphysics, the system is broken into two (or more) components that interact, such as an air duct consisting of a fluids domain (air) and a structural domain (sheet metal duct) that interact (or intersect) at the vent (or flap). When discussing multiphysics it is important to distinguish between the *physics* of the actual phenomenon, the mathematical *model* used to describe the physics, and the *software* that implements the algorithm that solves the mathematical model. Figure A.34 illustrates the distinction between these three aspects of the multiphysics problem. The full definitions used for these pieces are given below in addition to our definition of multiphysics.

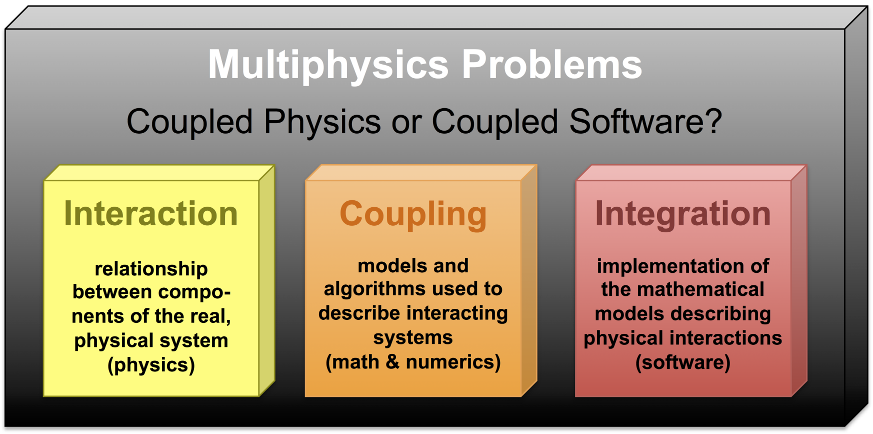

Multiphysics Problems
Coupled Physics or Coupled Software?
Interaction
relationship between compo-nents of the real, physical system (physics)
Coupling
models and algorithms used to describe interacting systems (math & numerics)
Integration
implementation of the mathematical models describing physical interactions (software)

Figure A.34: As precedented by [Keyes et al. (2013)], the preciseness of language is an important facet of the multiphysics discussion. Here we present our three core terms and the distinction between them: interaction (physics), coupling (mathematics and numerics), and integration (software).

Multiphysics
Problem or process that has two or more components that physically interact.

Interaction
Interdependence between components of the real, physical system.

Coupling
Mathematical models and numerical algorithms where the solution of an equation (or set of equations) is dependent upon the solution of another, different (set of) equation(s).

Integration
Implementation of the mathematical models describing the interaction and/or the joining of separate software applications to model multiple physical processes.

Software integration is distinct from numerical coupling and refers to the mechanics of interfacing and operating multiple software components in concert toward a common goal. Software integration is the main focus of *IMPACT*. Before presenting the full background and motivation behind *IMPACT*, it is pertinent to further emphasize the relevant language and color coding used throughout the discussion. This clarification will be followed by the background and motivation, a high level explanation of *IMPACT*, and a description of an example multiphysics problem.

A.1.1 Language, Terminology, and Color Usage

In their final report, the participants of the Institute for Computing in Science multiphysics workshop stress the importance of precise language when discussing multiphysics and its challenges [Keyes et al. (2013)]. This notion resonates strongly with the Illinois Rocstar development team and is important for understanding the subject matter of this work. Keyes et al. makes the crucial distinctions between "strong" and "weak" coupling of multiphysics models. They also distinguish between "loose" and "tight" algorithms used to actuate the coupling. As mentioned previously, we introduce further distinctions between "interact," "couple," and "integrate." Below, more key terminology is defined that will be used throughout this document when discussing different aspects of multiphysics problems, modeling, and simulations.

Strong (weak) interaction
The degree of interdependence among physical systems, a measure of how the changes in one physical system affect another.

Tight (loose) coupling
The degree of interdependence among mathematical models or algorithms, a measure of how much one model or equation set influences another.

Full (partial) integration
The degree to which two or more software applications work cooperatively, a measure of the extent to which software applications have been joined together.

Monolithic vs Partitioned multiphysics
In a *partitioned* approach, each physical domain is simulated by a dedicated solver, where each solver is independently developed and utilizes its own timestepping (staggered timestepping is used for the entire system). In a *monolithic* approach, a single solver is used for the complete set of interacting domains with timestepping performed synchronously for the entire system.

In addition to specific language, a color scheme is used throughout this document that remains consistent across the figures depicting the infrastructure. The color coding corresponds to different layers of the *IMPACT* infrastructure or to aspects that could be present in a general multiphysics software infrastructure. (Note that this color scheme does not apply in figures displaying physical domains.) The color coding is as follows.

Red → Connectors
Red represents components of the infrastructure that work to *connect* applications and/or services to one another. When discussing *IMPACT* specifically, these pieces comprise the Software Integration Toolkit (SIT) that helps transfer data and functionality among applications.

Green → Applications
Green is used to represent software *applications*. These applications usually model some type of physics but can represent any general user applications.

Blue → Drivers
Blue represents the main *driver* for software simulating a multiphysics problem. In relation to *IMPACT* , this piece is termed the orchestrator. The main driver orchestrates the flow to solve the problem.

Purple → Services
Purple is used to represent *services*, which are essentially applications, but they have been provided to the user through the *IMPACT* infrastructure. They can be thought of as applications that provide some extra functionality, such as data transfer and mapping, beyond solving the main physics of the problem.

Grey → SystemOS
Grey is used to represent the operating (base compute) system on which the software is running.

Now that the language and color scheme has been made clear, the background and motivation for creating *IMPACT* will be discussed.

A.1.2 Background

Many of today's important and challenging problems in science and engineering involve multiple, complex, interacting physical systems, and often involve combustion or other sources of energy release. Examples of such systems include fluid-structure interaction (FSI), conjugate heat transfer, thermomechanical coupling, and shock-to-detonation of energetic materials. As mentioned, multiphysics modeling and simulation refers to the advanced, coupled modeling techniques used to simulate these interacting systems. Large-scale modeling and simulation of such multiphysics problems using high performance computing (HPC) has become a crucial component of research and development in the private sector, academia, and the national laboratories.

Multiphysics problems can take a variety of forms which affects the methods used to model and simulate them. In some cases, multiple physical processes can be occurring within the same domain. An example of such a case is a problem involving needing to solve for fluid flow with heat transfer. In other cases, all physical processes are not present throughout the entire system. For example, one domain with specific physical processes could overlap only part of another domain with different physical processes. This overlap would create a third domain with the same (or different) dimensionality where the physical processes (domains) interact. This type of general situation is shown in Figure A.35.

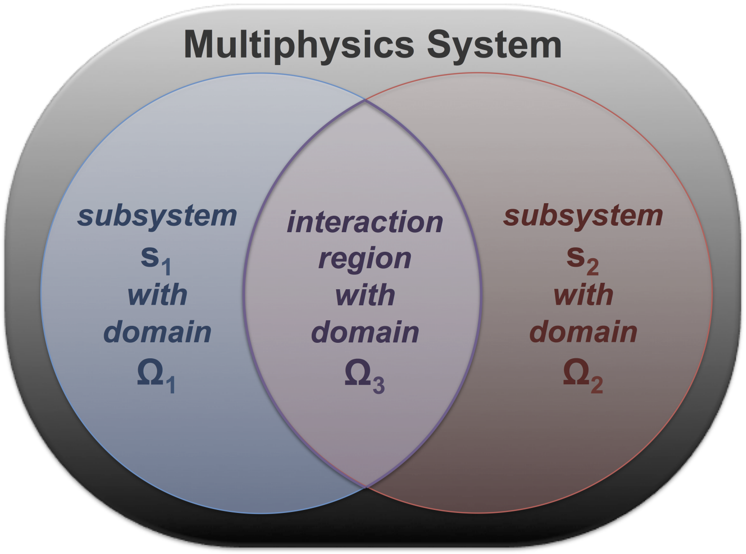
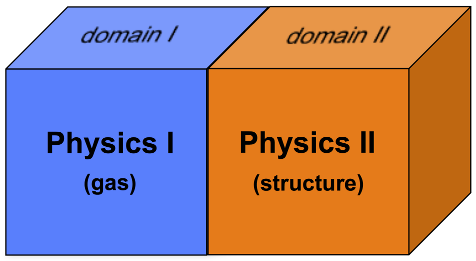
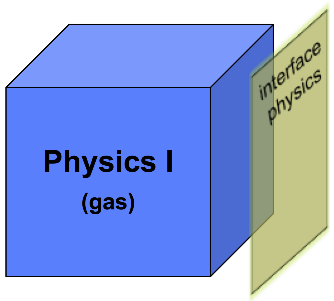

Multiphysics System
subsystem s_1 with domain Ω_1
interaction region with domain Ω_3
subsystem s_2 with domain Ω_2

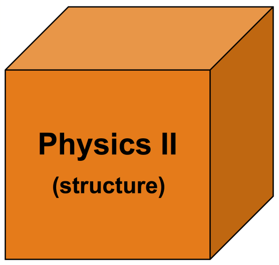
Figure A.35: In a multiphysics problem the different physical domains can interact in the space of another overlapping domain of the same (or different) dimensionality.

In a third case, the physical domains (of dimension n) do not necessarily have to overlap at all, but instead interact at an interface with dimension $n-1$. The setup of an example problem with this type of interaction is illustrated in Figure A.36. Although this figure shows an FSI problem, other multiphysics problems can have the same setup and problem flow. A partitioned approach is shown in this example, wherein distinct solvers would be used for each physical domain.

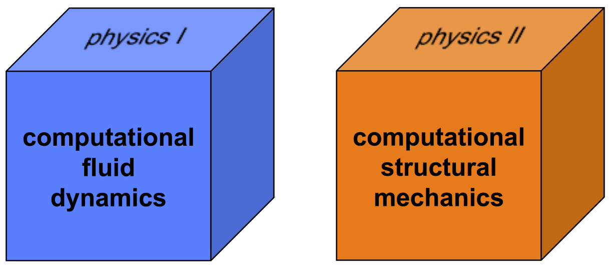


(a) The two domains have differing physical characteristics and interact at the interface.





(b) The interface may have its own physics.


physics I
computational fluid dynamics
physics II
computational structural mechanics

(c) Each domain is simulated numerically by methods observing the respective physics.

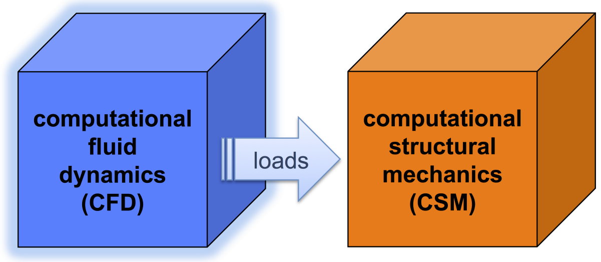
Figure A.36: Two or more 3D domains abut at a common interface. The goal of multiphysics computation is to simulate two or more physical domains that interact. In this problem they interact across a moving, reacting interface. (a) These domains do not overlap, the geometry of the interface is a 2D surface in space, and the domains may move and/or deform, but they do not come apart. Mass, momentum, energy, and charge are conserved across the interface; however, some physical quantities of interest "jump" at the interface (e.g., density). (b) The interface could be reactive with a combustive or other chemical process, or the interface could propagate or move due to some process.

The interaction between the physical domains in this partitioned FSI case can be as thus: the movement of the fluid applies pressure loads to the structure and the deformation of the structure causes displacement velocities to affect the fluid. In simulating this type of interaction, a staggered timestepping scheme can be used, computing the solution in one physical domain and then the solving for the second domain. This process is illustrated in Figure A.37.

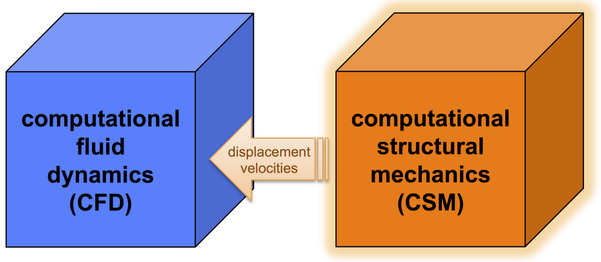
In a partitioned multiphysics simulation approach, each physical domain has a unique solver with its own distinct discretization schemes. Data must be mapped correctly from one discretization to another and the interface discretization moved correctly in the case of a reacting, moving interface. Figure A.38 illustrates this part of the FSI multiphysics simulation.

A.1.3 Motivation

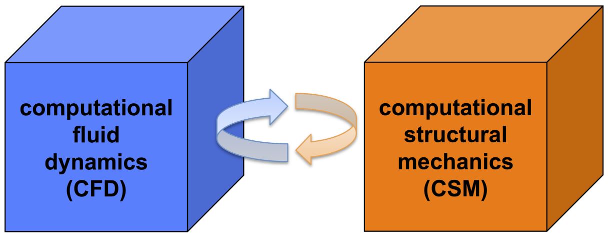
Several factors conspire to drive the cost of development and ownership of multiphysics computational capabilities inordinately high. The private sector entity has limited choices when deciding how to address a multiphysics simulation requirement. One must either invest in expertise for a "build-your-own solution," or purchase external expertise and capabilities. There are currently no existing standards or protocols for tools and interfaces, commercial or otherwise, that support general software and simulation integration and coupling for multiple parallel simulation applications. As a result, both choices incur costly software development and refactoring activities that often result in a single, limited-use capability or a project-specific solution.


computational fluid dynamics (CFD)
loads
computational structural mechanics (CSM)

(a) The fluids domain steps and calculates pressure loads at the interface, which are then passed to the structures domain.


computational fluid dynamics (CFD)
displacement velocities
computational structural mechanics (CSM)

(b) The structures domain steps and calculates the deformation and interface velocities, and the interface motion is sent to the fluids domain.


computational fluid dynamics (CFD)
computational structural mechanics (CSM)

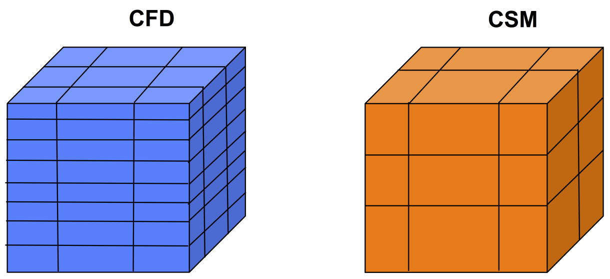
(c) The process is repeated to step the simulation in time.

Figure A.37: A partitioned approach with staggered time stepping. Computation of the interacting domain is centrally synchronized.

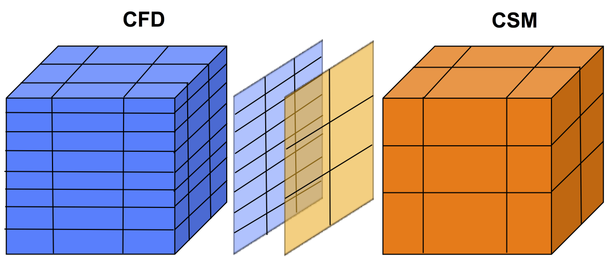
The difficulty of establishing a viable solution with commercial, off-the-shelf software is compounded by prohibitively expensive HPC or multiprocessor licensing models. These difficulties are not confined to the private sector, but are shared by industry, academia, and government entities with large-scale simulation needs. These technological and economic barriers to the quick production and testing of high fidelity, high-performance multiphysics simulation software are a significant bottleneck for research efforts in this and other areas that now, or will soon, rely heavily on multiphysics simulation [Council on Competitiveness (2011)].

The vision for *IMPACT* is to be a general, open-source multiphysics infrastructure that would allow the broader community to develop multiphysics computational capabilities using existing simulation applications. In addition, it would serve as a reference implementation to help guide the development of a new, open standard for highly inter-operable multiphysics software design and execution. In essence *IMPACT* allows other applications to be integrated and work cooperatively. Figure A.39 illustrates this concept. *IMPACT* is designed for use with multiphysics, however, is not limited to this particular situation and, in reality, transcends multiphysics altogether.

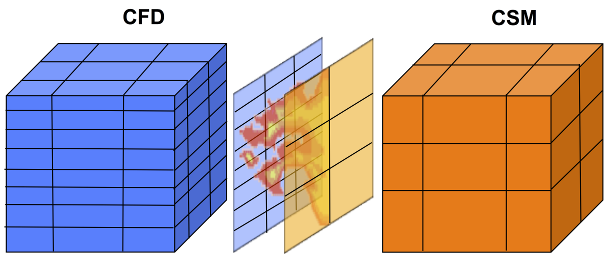
Standardization and specification for software integration interfaces across application, language, and platform boundaries in the HPC environment are necessary for advancing the state-of-the-art predictive multiphysics simulation capabilities as the underlying algorithms and HPC platforms evolve. General data-driven interfaces and their adoption by the community are a key step in establishing these standards and specifications. Modern designs must seek to provide advanced interfaces to scientists, engineers, and analysts that hide many of the details of the platforms and software mechanics. Such an infrastructure will also significantly reduce the barrier to entry for private sector entities endeavoring to develop multiphysics computation capabilities that leverage the nation's HPC resources. To maintain competitiveness in the modern world marketplace, simulation applications must be designed for deployment in integrated environments and coupled models, simulating ever more complex interacting systems.



(a) Each domain is decomposed for distribution among processes.



(b) The domains and shared interface are discretized by each solver application.



(c) The interface is reactive. It catches on fire, ejects materials, and propagates (burns).

Figure A.38: A partitioned multiphysics simulation. (a) In general, the domain decomposition is disparate across the domains. (b) Each solver marches through time according to the domain-specific physics (i.e., timesteps are disparate). Getting the interface data transfer correct is essential for accuracy and stability of the simulation. (c) Combustion depends on solution and geometry information from the other domains. Both geometries change drastically as the material burns away.

In light of the generality presented Figure A.39, Figure A.40 illustrates how a complete integrated system modeling interacting physics might look to the user wanting to solve the FSI problem discussed in Figure A.37. In this case, the relevant solvers are loaded to address the physics in the different domains (fluids, structure, and interface combustion) as well as the necessary services that manipulate and transfer the required data. These components are coupled though the integration interface, and the entire simulation is orchestrated by the main driver.

A.2 Overview of *IMPACT*

Part of our goal for this effort is to spur an inter-organization discussion on standards and protocols for general parallel software integration. For this we require an abstraction that separates out capabilities and data constructs that are domain-independent (i.e., independent of multiphysics and M&S).

A significant portion of the effort required in building integrated composite software systems lies in infrastructure development. Common architectures include those in which the various constituent components are centrally orchestrated or end-to-end workflows with no centralized orchestration. The integration of multiple executables, one of which may have "ownership" of the control flow, is often required. These architectures sometimes involve physical networks between the various components of the integrated system. This situation usually calls for physical network-based communication and often involves complicated, event-driven control flow management.

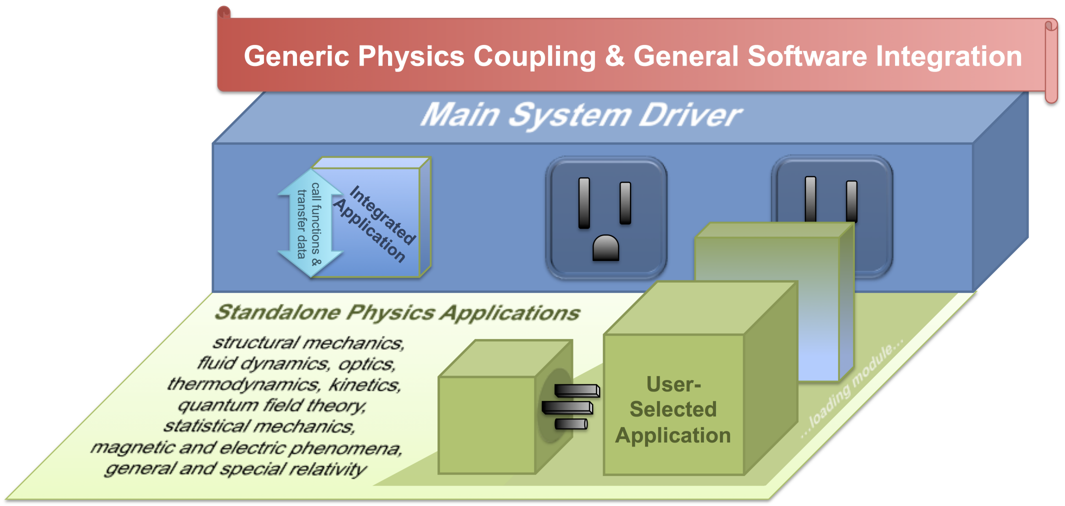

Generic Physics Coupling & General Software Integration
Main System Driver
Integrated Application
call functions & transfer data
Standalone Physics Applications
structural mechanics, fluid dynamics, optics, thermodynamics, kinetics, quantum field theory, statistical mechanics, magnetic and electric phenomena, general and special relativity
User-Selected Application
...loading module...

Figure A.39: Generalized application coupling and integration. From myriad standalone physics applications, scientists and engineers choose two (or more) applications to couple together to solve the complex interactions of their physical systems. The main system driver organizes the plug-and-play nature of the integrated applications, and the coupling is actuated by the infrastructure connectors.

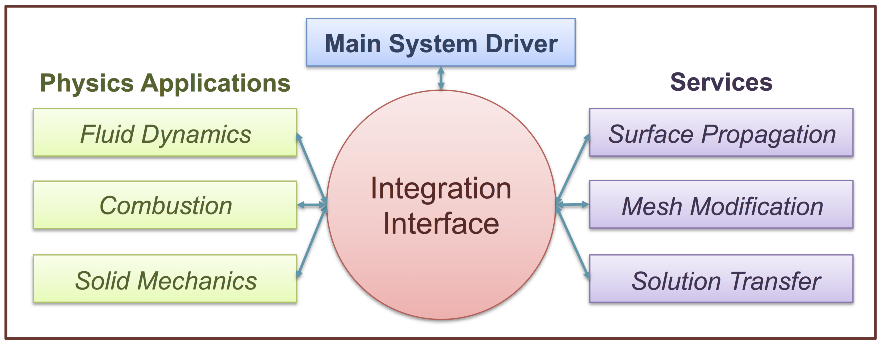


Figure A.40: The integration interface provides the mechanisms by which applications can publish and access methods and data. This interface is the "glue" of the multiphysics simulation.

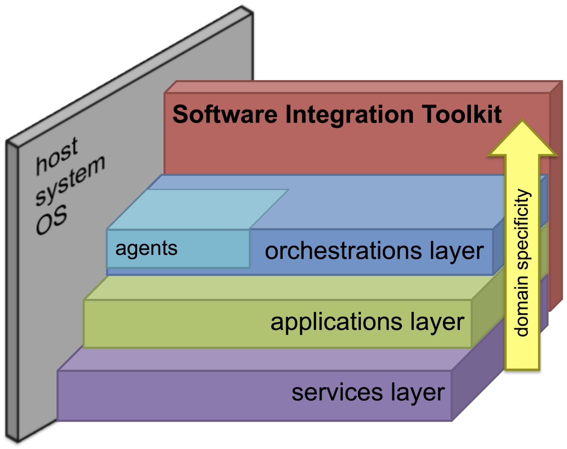


Figure A.41: Infrastructure abstraction model. The Applications Layer can use capabilities from the Services Layer, interface with other Applications Layer components, and participate as part of a composite software system coordinated by the Orchestrations Layer. Agents reside within the Orchestrations Layer and interface with the Applications Layer. All intercomponent exchanges are mediated by the Software Integration Toolkit.

In *IMPACT*, we desire a software integration infrastructure that generalizes the common features of these architectures and encapsulates them into a separate toolkit for use by the development community in software integration endeavors. Further, we want to identify a minimal set of domain-independent, multiphysics simulation capabilities, and factor them onto the infrastructure as "services," such that the services, together with the software integration toolkit, form a general infrastructure for multiphysics capability development. A high level depiction of the *IMPACT* infrastructure layers is shown in Figure A.41.

The back layer in Figure A.41 represents the Software Integration Toolkit that provides the basic constructs allowing applications to publish native data structures and functions. In this context, we use "publish" to mean that the application can describe and provide access to native data structures and functions by outside software. Modules or user Applications are those that provide the "primary" domain-specific capabilities that must be integrated into the composite software system. The abstraction model in Figure A.41 shows the *IMPACT* infrastructure layers in order of decreasing generality. A brief introduction of each layer is discussed here and presented in more detail in the following section.

Software Integration Toolkit The software integration toolkit layer provides the basic constructs necessary for the integration of multiple software components and encapsulates the constructs most suitable for industry-wide standardization.

- Encapsulates applications into component objects called *modules*
- Provides constructs for publishing methods and associated metadata
- Provides constructs for publishing data and associated metadata
- Provides associated control mechanisms for inter-language access to published methods and data
- Provides intercomponent communication

Orchestrations Layer The orchestration layer encapsulates the driver for the integrated composite software system. This layer is the most domain-specific.

- Directs the flow of the problem
- Ensures that the proper functions are called in the proper order
- Organizes the flow of data
- Solves the multiphysics problem at hand utilizing the available tools

Component Agents An agent is responsible for interfacing with an application module's component interface and presenting the required model interface to the coupling object.

- Makes calls to the application's functions
- Accesses application data
- Provides data to the application
- Allows application to access and request other data or function calls

Applications Layer The application layer encapsulates those applications that provide the "primary" domain-specific capabilities that must be integrated in the composite software system and includes single physics domain applications.

- Solves a subset of the physics of the entire multiphysics problem at hand
- Integrated through the SIT by shallowly adapting the source code or wrapping the executable

- Examples: solvers for CFD, CSM, FSI, CHT, thermodynamics, quantum field theory, and magnetic phenomena

Services Layer The service layer provides capabilities through the software integration toolkit that can be used to implement and perform domain-specific capabilities and tasks. An example of this layer is services to accomplish data mapping between disparately discretized surfaces.

- Parallel I/O for general data, geometry, mesh, and mesh-bound solution data
- Parallel mesh-to-mesh data transfer (e.g., interpolation)
- Simple vector algebra on mesh-bound solution data
- Parallel volume mesh smoothing
- Parallel discrete surface operations (e.g., calculate face normals)
- Parallel surface propagation
- Collective operations for decomposed domains (e.g., reduce on shared nodes)

A.3 In-Depth Look at *IMPACT*

The overarching technical objective of the *IMPACT* project is to design and implement a general infrastructure for software integration in support of the development of multiphysics modeling capabilities. Our approach was to form an abstract model wherein all multiphysics-specific capabilities were factored out into layers with increasing domain specificity. The abstraction is shown in Figure A.41. This model forms a framework around which industry-wide standards may be discussed and developed.

A.3.1 Software Integration Toolkit

At the core of *IMPACT* lies an abstraction layer for general software integration in the HPC environment. The layer that we call the Software Integration Toolkit (SIT) defines these primitive software integration constructs which facilitate the sharing of application-native data and methods across the component–component boundary. The SIT and its relationship to the other layers of the abstract model was depicted in the previous section in Figure A.41. All component–component interactions are mediated through the SIT. It is designed to support several different composite software system architectures and a variety of user applications. We identify the following constructs and capabilities as those under the purview of the SIT.

Application encapsulation
The SIT must provide an abstraction that encapsulates or represents an application and its interface to the composite system. All of the applications' interactions with the composite software system are conducted through this abstraction. This abstraction must support serial and parallel applications which may use a variety of parallelization strategies (e.g., OpenMP, MPI, or proprietary). Support should be provided for integrating both library applications as well as applications that must run as stand-alone executables (i.e., users should be able to integrate both stand-alone applications and libraries).

Native method publication
The SIT must provide abstractions and constructs for publishing application-native methods (i.e., functions) and associated metadata. Since applications integrated into the composite system will

typically be coded in a variety of programming languages, allowing for coexistence of multiple programming languages is necessary. This piece of the SIT must also include the mechanisms required to invoke application-native methods across the programming language and component boundaries.

Native data publication
To support the vast disparity in programming languages and data structures among composite system components, a unified, language-independent, abstract view of application-native data is necessary to standardize intermodule exchanges. This abstraction must be flexible and self-describing. General abstractions, constructs, and mechanisms for publishing application-native data and associated metadata must be provided by the SIT, which includes mechanisms for specifying access policies on data (e.g., read-only, read/write, etc.) and layout of the data in memory.

Intercomponent and interprocess communication
An interface must be provided to hide the mechanics and complexity of the communication between components of the integrated software system. This intercomponent communication interface allows for the greatest variety in the architecture for the applications and the composite software system. For example, to support architectures in which some component must run as a stand-alone executable, some form of interprocess communication must be used to actuate communication between the stand-alone component and the rest of the system.

A.3.2 Orchestration

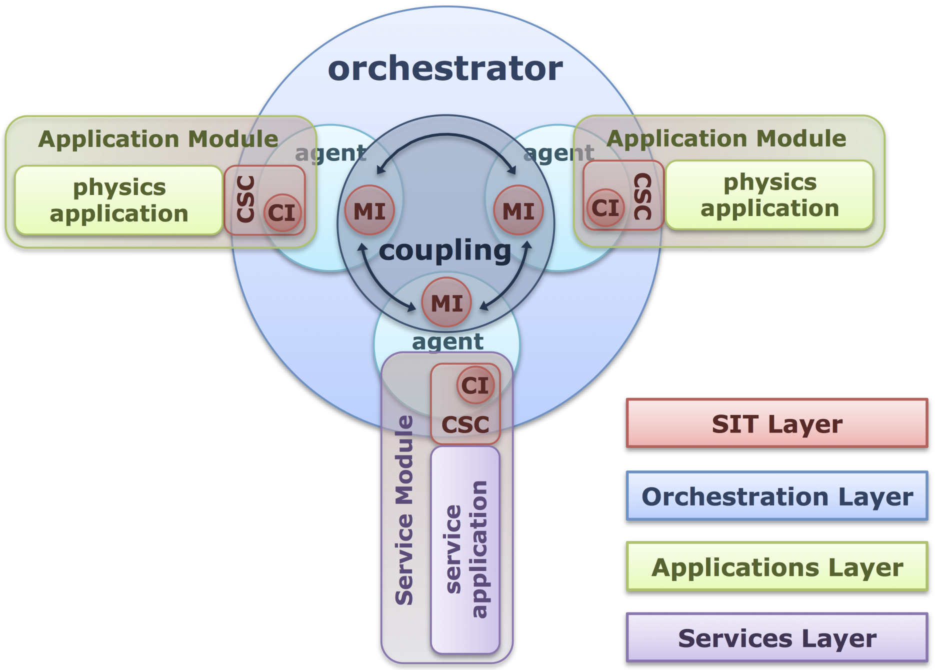


Figure A.42: Orchestration and orchestrator constructs that drive the system. The Orchestrator is the driver of the system. It drives the coupling object, which in turn uses actions (shown here as dark blue arrows) to invoke functions and exchange data between components. The coupling interface defines the model interfaces (MI) that it needs to implement the coupling algorithm. These MI are presented by the application-specific Agents that directly interface with the Applications through their defined component interfaces (CI). The component-side client (CSC) provides access to the Application-native data and functions to the outside world. As before all intercomponent exchanges are mediated by the Software Integration Toolkit.

The Orchestrations Layer of the model integration infrastructure is the piece that implements the integrated composite software system. The Orchestrations Layer as shown in Figure A.41 and expanded upon in Figure A.42, is the most domain-specific piece of the integrated capability. Often, but not always, this piece is implemented as an orchestrating driver, or Orchestrator. Other possibilities for Orchestrations Layer constructs include "passive" or event-driven middleware components. Regardless of implementation, Orchestrations Layer constructs have access to one or more applications through the SIT and use intrinsic functionalities, or those provided by services, to actuate the interactions between system components.

As shown in Figure A.42, once multiple Application modules have been created, an Orchestrator is created to implement the *coupling scheme* and drive the simulation by managing the control flow and the intermodule interactions. The *IMPACT* infrastructure provides an orchestration toolkit called the Software Integration Manager (SIM). SIM provides constructs designed to be used in the implementation of Orchestrators. It is instructive to think of the Orchestrator as having three main functionalities: *driver and driver logistics*, *coupling algorithm*, and *component agents*.

Driver and driver logistics

The Orchestrator is responsible for loading the Application modules, managing their MPI communicators (if necessary), and handling the control flow among the components of the integrated simulation. SIM offers basic constructs for handling these functions for synchronized operation of serial and MPI parallel components.

Coupling algorithm

SIM offers three constructs for building coupling algorithms. The *coupling* objects encapsulate a *scheduler* that executes *actions* in a particular order and/or progression to actuate the coupling algorithm. The actions are a general abstraction and encapsulation for almost any procedural set of instructions. Typical actions include invoking modules and interpolating between meshes. The key consideration in the implementation of this part of the system is that the interface to each domain-specific model (i.e., the so-called *model interface* (MI)) is defined by what the coupling algorithm requires. This consideration puts some restrictions on the possible choices for the domain-specific simulation applications that this coupling algorithm may utilize in the integrated simulation.

Component agents

Once the driver and coupling algorithm are in place, and model interface defined, the domain-specific Application modules are *plugged into* the simulation by using SIM's Agent construct. An Agent is responsible for interfacing with an application module's component interface (CI) and presenting the required model interface to the coupling object.

In multiphysics systems, Orchestrators typically manage the control flow and implement the coupled timestepping schemes. For example, *Rocstar*'s orchestrator managres control flow, implements the coupled timestepping, and actuates the data transfers between physical domain-specific components, including implementation of the jump conditions.

A.3.3 Applications and Services

User Applications, shown in green in Figures A.41 and A.42, are those that provide the "primary" domain-specific capabilities that must be integrated into the composite software system. In the multiphysics case, two example user applications could be an application implementing a computational fluid dynamics (CFD) model and another one implementing a computational structural mechanics (CSM) or transient thermal model, as illustrated in Figure A.36. In this case the target simulation for the integrated software system could include fluid–structure interaction (FSI) or conjugate heat transfer.

User Applications are integrated through the SIT by adapting the application source code or wrapping the application executable or interface using an API provided by the SIT implementation. This API provides the constructs and mechanisms necessary for the application to share its data and functions with the integrated software system. Applications that interact with the coupled system only through the infrastructure are considered fully integrated. All others are considered as only partial integrations. Once integrated, whether fully or partially, the user application becomes a *component* of the integrated system.

Application adaptations for integration should be shallow. The SIT should provide an application programming interface (API) that can understand (perhaps after light massaging) the application-native data structures and interface with application-native methods. The integration adaptations are typically in the application's native programming language and live outside the main source code for the application, which avoids the exposure of any potentially proprietary information to other components of the integrated software system.

In our software integration architecture, shown in Figures A.41 and A.42, Services are specific capabilities that are provided through the constructs of the SIT that support domain-specific integrated software systems. Examples include computational and numerical services, such as parallel I/O, mesh smoothing, and surface propagation.

A.4 Modules, Coupling, and Infrastructures

IMPACT facilitates the integration of in-house, commercial, and open software components for rapid multiphysics application development. Our *IMPACT* project objectives require assembly of working examples of integrated simulation applications using the multiphysics infrastructure. Each developed application should successfully run a subset of the example V&V test cases and be built from freely available, open components. These simulation applications will serve as both an examples of how to use the infrastructure and as demonstrations of feasibility. Development of an integrated multiphysics application using the infrastructure requires two major activities: *application integration* and and *simulation orchestration*.

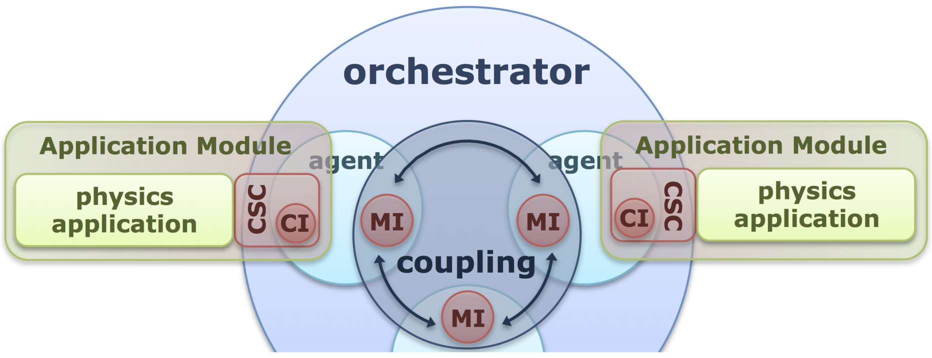


Figure A.43: Application modules. The coupling interface defines the model interfaces (MI) that orchestrator needs to implement the coupling algorithm. These MI are presented by the application-specific Agents that directly interface with the Applications through their defined component interfaces (CI). The component-side client (CSC) provides access to the Application-native data and functions to the outside world. This figures is a subset of Figure A.42 that focuses on Application Modules.

A.4.1 Integrating Applications as Modules

Figure A.41 covers most of the main high level components of the infrastructure design. Infrastructure constructs are used to shallowly modify the user's application, producing an application module. A module is an application-native software construct with an embedded component-side client, as show in Figure A.43.

The component-side client (CSC) is an Application-native software construct that uses an infrastructure application programming interface (API) to provide read/write/execute access to the original user Application's native data structures and computational methods. This access is communicated over an intercomponent communication method to one or more component interfaces (CIs) in the integrated system. Integrated systems are accomplished by the production of the Software Integration Manager (SIM). The SIM manages all of the component interfaces for all of the applications and mediates their interactions.

Application
The Application, or user application, is the software construct that encapsulates the capability that the user wishes to use in an integrated system of multiple components. The Application can be open source, a closed/proprietary library, or a standalone executable.

Component-side client
The component-side client (CSC) is the software construct embedded in the Application that allows the Application to publish its data and computational methods for use in the integrated system. The CSC is written in the Application's native programming language, with access to both the Application's native data structures and the native programming language bindings for the infrastructure API.

Application module
An Application Module is an Application with its associated component-side client. It is an integration-ready software component that encapsulates the Application.

Intercomponent communication
Communication of data, metadata, and commands between system components and the integrated system is provided by intercomponent communication. Specifically, it provides the communication method between the component-side client and component interfaces. Note that the options for the communication substrate of the intercomponent communication (i.e., the low level communication method, e.g., Posix IPC or MPI) are limited by the type of application (e.g., open-source, closed, executable only, etc.).

Component interface
Each Application Module interacts with the integrated system through one or more component interfaces. From the integrated system's perspective, each Application Module and the integrated system's interactions with the module are encapsulated by and done through the component interface.

Software Integration Manager
The Software Integration Manager (SIM) is the software construct that manages all of the component interfaces and their interactions. The SIM typically manages the control flow of the integrated system, but not always. Depending on implementation, the SIM can be driven by other integrated components. A SIM can be written in any language for which the infrastructure implementation offers an API or bindings.

How the Application Module and Software Integration Manager (SIM) map to processes is application and system specific and has implications on the allowed communication substrate for the intercomponent communication. If the SIM and the Application Module are implemented in a single process, then intercomponent communication would be most efficiently implemented as a direct memory access. On the other hand, if the SIM and the Application Module are in separate processes, then the intercomponent communication has to use a particular communication method. Possible choices for this communication method include MPI, MPI2, shared memory, TCP/IP, some other Unix IPC, or even files. The available communication substrates for the intercomponent communication are implementation-specific, but clearly have implications for the types of integrated systems that can be constructed.

A.4.2 Serial vs Parallel Applications

The modern HPC environment typically consists of many multicore nodes together with high speed interconnects, usually with some number of GPU nodes. The current design does not include consideration of GPU parallel applications. Instead, we focus on the need to support MPI, OpenMP, shared memory, and hybrid applications. Since the infrastructure itself is not GPU parallel, there are some resulting design constraints for the integrated system involving GPU parallel user applications, but this design does not preclude use of the infrastructure by GPU parallel applications.

In general, each integrated component will have its own native communication method and parallelization strategy, and the infrastructure constructs need to support the integration of these disparately developed and parallelized application components. The following is meant to be a general description of an integrated system architecture. Several existing multiphysics simulation integration packages, such as *Rocstar*, *PreCICE*, *MCT*, and *LIME*, implement systems very similar to those described here.

The general system architecture with two integrated serial applications is shown in Figure A.43. In general, each Application Module can use any of the implemented intercomponent communication methods. The Software Integration Manager (SIM) manages all of the component interfaces (CIs) for all of the Application Modules, mediates their interactions, and usually (but not always) manages the control flow between the integrated components. Depending on implementation, integrated components can be serial or parallel standalone applications or libraries, spawned separately or as children of the SIM process.

The parallel application typically, but not always, uses MPI for the communication substrate. The Application's native communication method is used to provide interprocess communication between the component-side clients (CSCs) for each processor. Each CSC can communicate with the SIM process via the intercomponent communication, or the SIM–CSC communication can be done by only one CSC. Parallel components with a single communicating CSC are usually OpenMP, or otherwise threaded applications, but this situation can also arise with a master–slave type of parallelization model using MPI or another interprocess communication method. In parallel integrated systems, the SIM can, itself, be a parallel construct. Parallelization of the SIM requires a communication method for SIM interprocess communication, which is usually done by MPI, but is implementation-specific in general.

A.4.3 Relationship to Other Infrastructures

There have been a number of interfaces developed relatively recently that target large scale, scientific software development. While these interfaces do address some of the varied requirements of multiphysics infrastructure for HPC applications, there remains a need for a standardization and interface specification for parallel software coupling. It is also important that new infrastructure efforts allow for relatively easy integration of existing scientific applications to lower the development costs and encourage a high level of software reuse, thereby lowering the entry barriers to HPC multiphysics simulation to industry and the research community at large. Some of the interfaces that we considered were *POOMA* [Reynders et al. (1996)], *ALEGRA* [Budge and Peery (1998)], *Overture* [Bassetti et al. (1998)], *LIME* [Stewart and Edwards (2004)], *Cactus* [Allen et al. (2001)], *CCA* [Allan et al. (2002)], and *OpenFSI*. Additional discussion of the interfaces most relevant to the current endeavor is given below.

OpenFSI

OpenFSI is an open-source, fluid–structure interaction (FSI) infrastructure that is currently under development. *OpenFSI* is built over the Service Component Architecture, which is a relatively new specification for constructing applications using a Service-Oriented Architecture. While *OpenFSI* is a step in the right direction, in that it attempts to provide a standard specification for FSI, it does not address the general multiphysics simulation development problems, nor does it stand ready to make use of HPC, as it provides no inherent parallelism or support for external parallelism. The infrastructure we are discuss here would complement *OpenFSI* by providing mechanisms by which the *OpenFSI* client could communicate across the application-application boundary and perform conservative and accurate data mapping across the “wetted” surface.

LIME

LIME is an infrastructure developed at Sandia National Laboratories and is one of the most advanced parallel interfaces for integrated multiphysics simulation. Unfortunately for application software authors, *LIME* is a development infrastructure that must be adopted throughout the application development effort. While it offers a very powerful environment for development and multiphysics simulation, it is not well suited to integrating existing applications without significant code overhaul.

Cactus

Another powerful parallel simulation infrastructure is found in *Cactus*, originally developed at the National Center for Supercomputing Applications. In this framework, all parallelism is provided by the infrastructure itself, again requiring infrastructure adoption in the development effort.

Common Component Architecture

The multiphysics infrastructure that we discuss here shares several features with this framework. The *Common Component Architecture* (*CCA*) allows one to wrap existing software constructs to create software components that present a standard interface to the underlying infrastructure and other components. The level of integration offered by the *CCA* is fine-grained, and places emphasis on runtime discovery of available services, similar to the Service-Oriented Architecture approach.

Trilinos

Trilinos is a comprehensive framework dedicated to facilitating large-scale multiphysics applications and simulations [Heroux et al. (2005); M. A. Heroux et al. (2012)]. Developed by Sandia National Laboratories, a leader in multiphysics software, *Trilinos* is the current state-of-the-art in advanced multiphysics coupling, offering many constructs designed to allow components to interoperate and cooperate in the HPC environment. The infrastructure we discuss here is complementary to *Trilinos*, providing the constructs needed to take advantage of the capabilities *Trilinos* offers, while offering additional services for the applications that must stand alone (e.g., commercial/closed packages from independent software vendors).

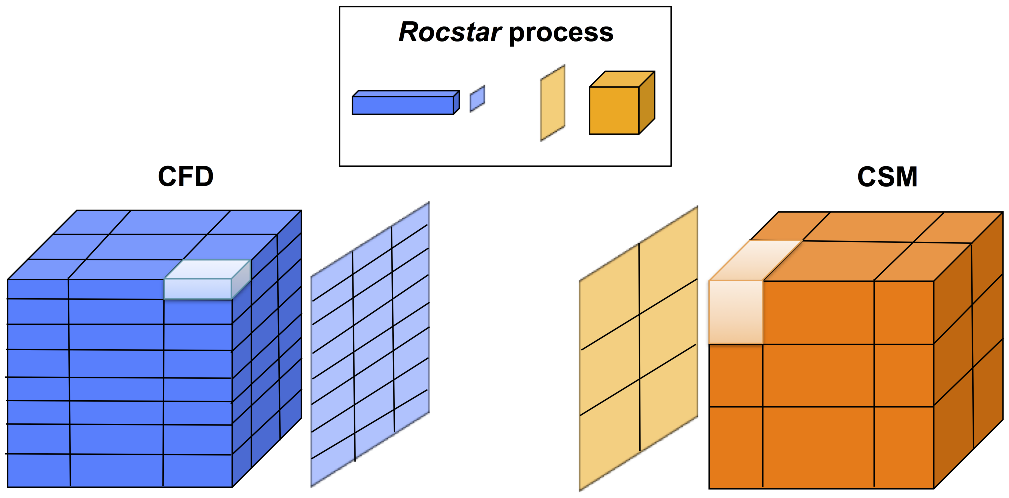


Figure A.44: *Rocstar* process – Domain decomposition. Each *Rocstar* process has one or more sections of one or more of the solvers' domains.

A.5 *Rocstar* Multiphysics Example

In this section, we present a higher level, pictorial explanation of multiphysics coupling and integration, with a focus on the *Rocstar* package. *Rocstar* is an open-source, multiphysics simulation suite distributed by Illinois Rocstar and its core constructs form the basis for *IMPACT*. We use *Rocstar* to illustrate the components *IMPACT* and their relationship to partitioned multiphysics through use of a generalized, coupled computational fluid dynamics (CFD), computational structural mechanics (CSM) problem, similar to that first introduced in Figures A.36 – A.38. The simulation is done in parallel, so the CFD and CSM domains are decomposed and distributed among processors. The problem setup and domain decomposition is illustrated in Figure A.44. Note that the domains are not decomposed in the same manner.

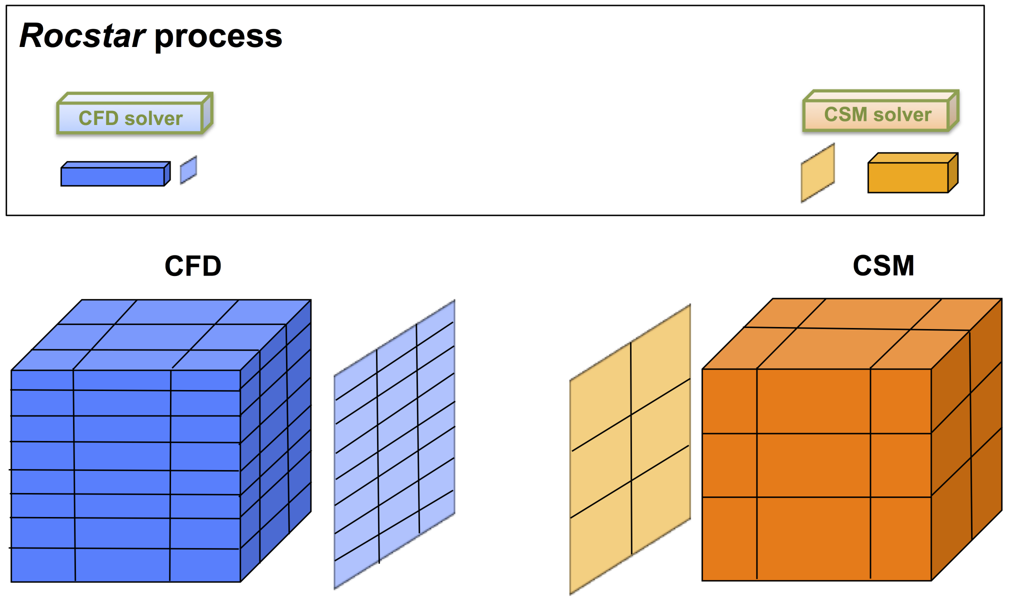


Figure A.45: *Rocstar* process – Solver Applications. Each domain is simulated numerically by methods observing the respective physics. The domain-specific (CFD and CSM) solvers are physics Applications and are therefore outlined in green.

Figure A.45 illustrates that this problem is solved using a partitioned approach. Therefore, distinct solvers are used for the CFD domain and the CSM domain. These solvers are the physics applications discussed in the previous sections, which are integrated using *IMPACT*.

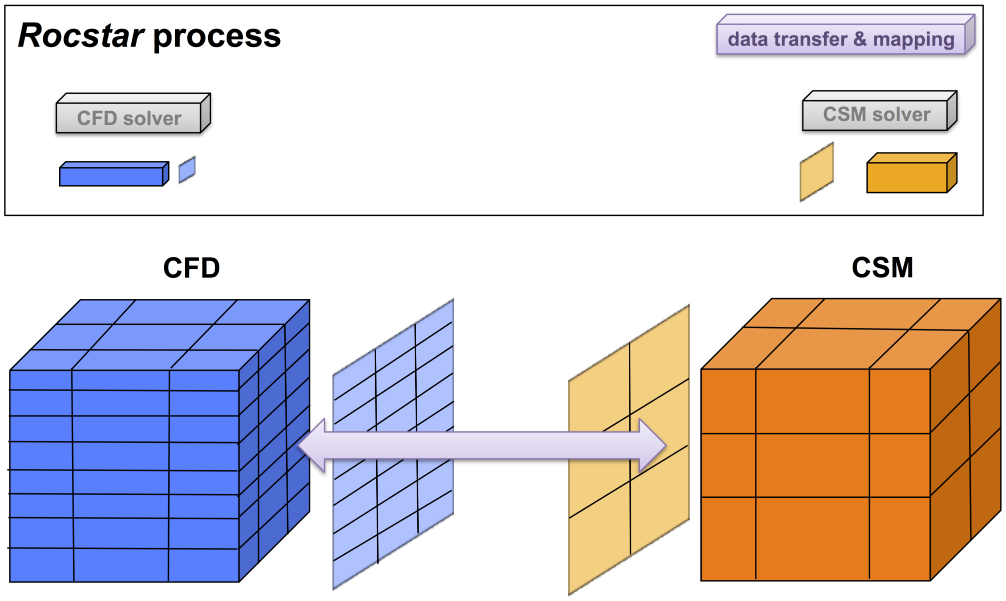


Figure A.46: *Rocstar* process – Data Mapping. Accurate, conservative data mapping across the interface and processor-geometry mapping is required. Data mapping is provided by the Services Layer and is therefore shown in purple.

Data mapping between the solvers is a critical issue that is handle by the *IMPACT* services. This mapping is shown in Figure A.46. Data must be mapped across the different domain decompositions, and perhaps adjustments made for different meshing techniques between the solvers.

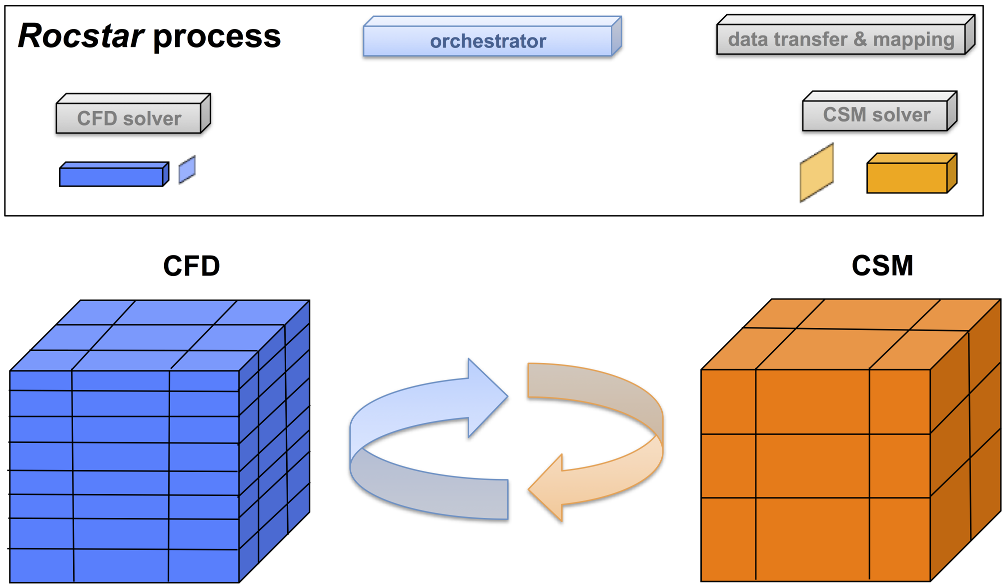


Figure A.47: *Rocstar* process – Orchestrator. There must be a control flow manager for timestepping, handling some jump conditions, and converting units. The Orchestrator is the main system driver.

Figure A.47 illustrates the job of the orchestrator in this context. It would control the entire problem flow including passing of data from one solver to another, ensuring that timesteps are synchronized or properly adjusted if staggered, and that proper function calls are made in order.

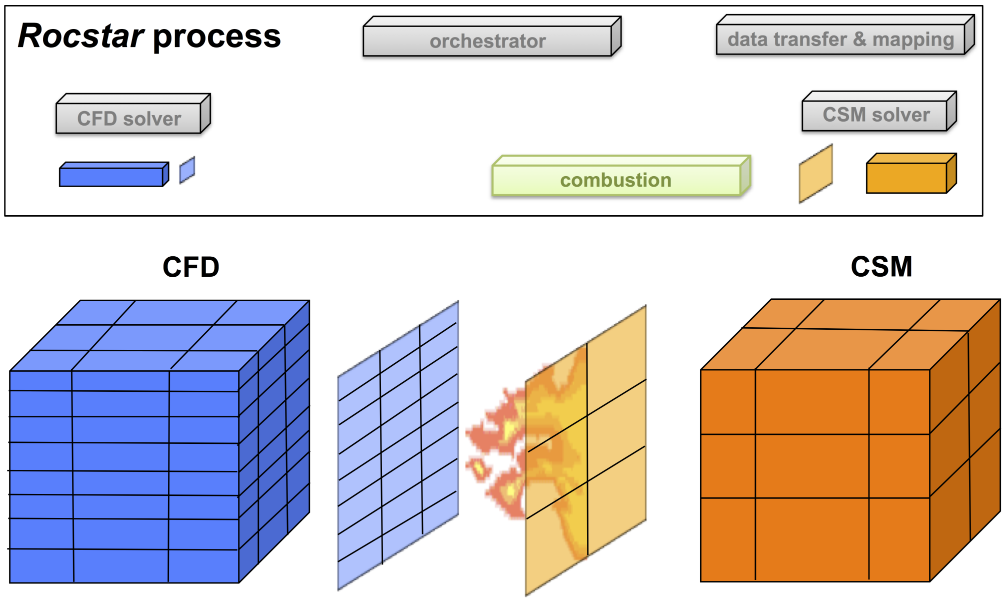


Figure A.48: *Rocstar* process – Combustion. To handle burning, we need a combustion solver capable of operating on geometry and data from other solvers and their domains. Combustion is a physical process addressed by an Application and is therefore shown in green.

There may be unique physics occurring at the interface of the two domains. In this case, combustion occurs at the CFD–CSM interface. This physical process could be handled by another separate physics application or by one of the existing applications. Figure A.48 illustrates this concept.

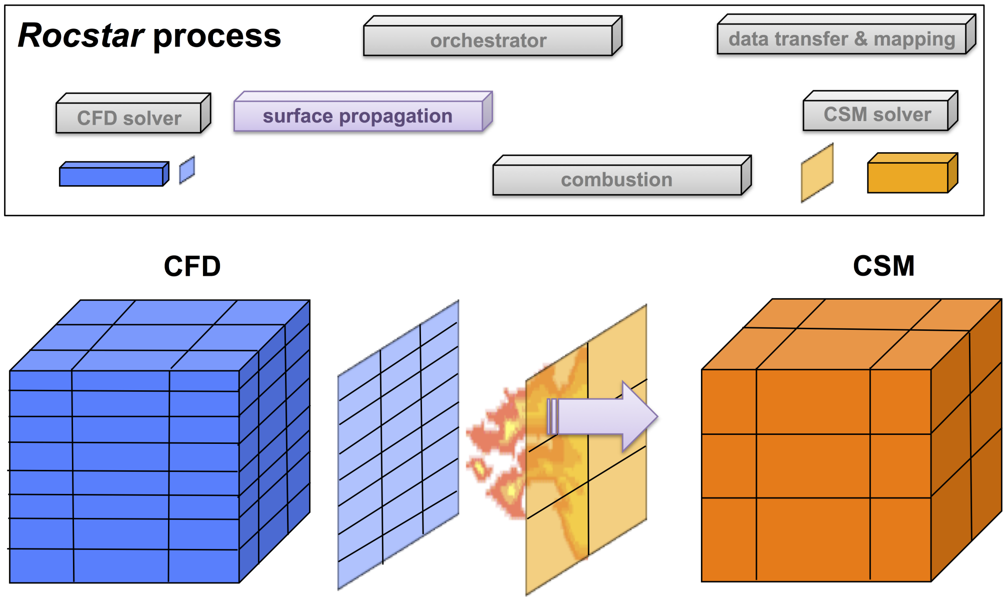


Figure A.49: *Rocstar* process – Surface Propagation. We need sophisticated surface propagation capabilities to handle the interface motion due to burning. Surface propagation is provided by the Services Layer and is therefore shown in purple.

Surface propagation is required to model the regressing, burning interface. *IMPACT* has a service module for surface propagation as illustrated in Figure A.49.

Figure A.50 shows that mesh modification may be needed after altering the problem geometry due to surface propagation. *IMPACT* has a service module for mesh modification as well. In this case, both meshes will have to be altered, and it is critical that the correct mapping between the domains is ensured.

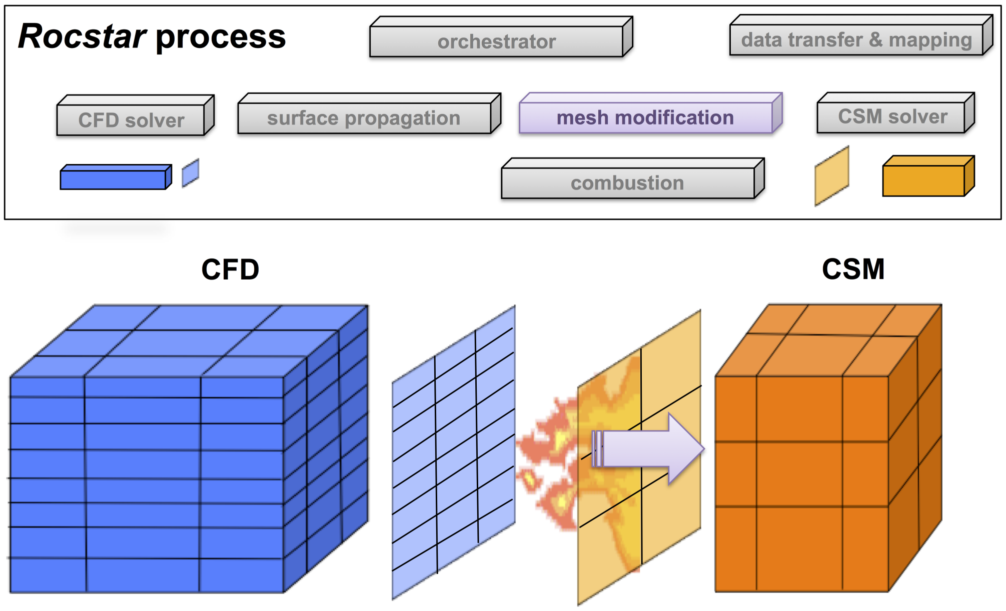


Figure A.50: *Rocstar* process – Mesh Modification. Mesh modifications will be required for handling the extreme changes in geometry due to burning and deformations. Mesh modification is provided by the Services Layer and is therefore shown in purple.

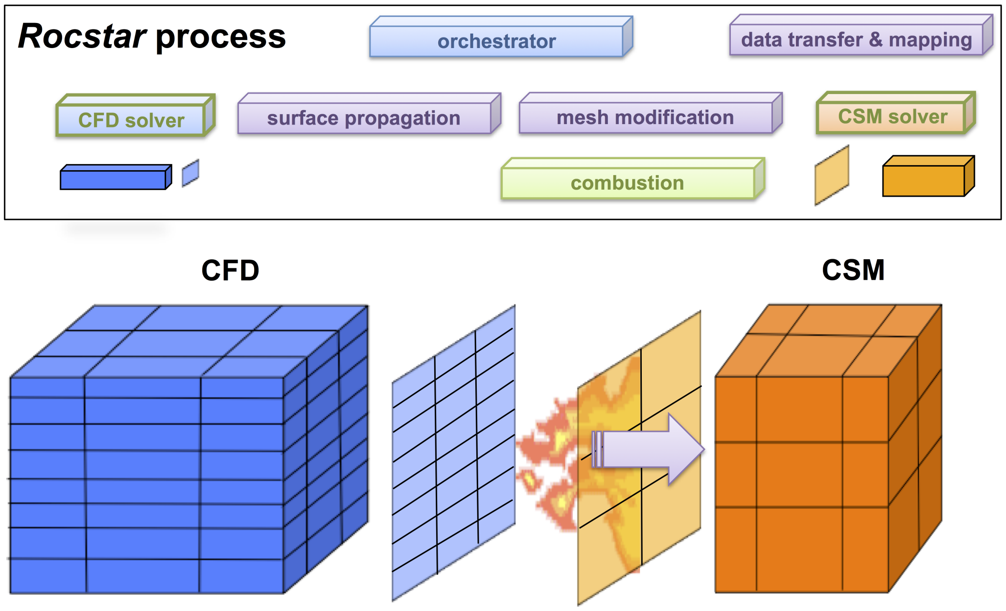


Figure A.51: Complete *Rocstar* process. All of the pieces have to interact in an efficient manner, sharing data and methods and working together, to simulate the complete system. The color coding of the different pieces represents to which *IMPACT* abstraction layer they belong.

Finally, Figure A.51 shows all the pieces of the *Rocstar* simulation discussed above, each with the appropriate color coding used throughout the document.

B IMPACT User's Guide

***IMPACT* User's Guide**
Version 0.1.0 IllinoisRocstar LLC October 25, 2016

License

The software package sources and executables referenced within are developed and supported by Illinois Rocstar LLC, located in Champaign, Illinois.The software and this document are licensed by the University of Illinois/NCSA Open Source License (see `opensource.org/licenses/NCSA`). The license is included below.

For more information regarding the software, its documentation, or support agreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

B.1 Overview

IMPACT is a developing suite of software packages with an integrated build system. It is designed to be used as an infrastructure for composing multiphysics simulation capabilities from multiple, disparately developed simulation applications. The software is now fully functional and can be used to build multiphysics capabilities in its current state. The following quick start guide will help the user get started with obtaining, building, and using *IMPACT*.

B.2 How to Get *IMPACT*

IMPACT is distributed from its online repository, `https://github.com/IllinoisRocstar/IMPACT`. This distribution package includes the source and all available documentation. There are other files available for download which are not part of this integrated package.

This distribution directory structure for *IMPACT* contains:

```
IMPACT/
 |__ AUTHORS
 |__ LICENSE
 |__ README
 |__ CMakeLists.txt
 |__ Documentation/
 |    |__ IMPACT_User.pdf
 |    |__ COM_User.pdf
 |    |__ SurfX_User.pdf
 |    |__ SurfMap_User.pdf
 |    |__ SIM_User.pdf
 |    |__ SimIO_User.pdf
 |    |__ Simpal_User.pdf
 |__ COM/
 |__ SIM/
 |__ SimIO/
 |__ Simpal/
 |__ SurfMap/
 |__ SurfUtil/
 |__ SurfX/
```

This directory is the *source directory* of the distribution. It is highly recommended, and this document will assume, that the user will set an environment variable to indicate the *IMPACT*source directory:

```
MPACT_SOURCE=/the/full/path/to/IMPACT
```

The software packages and their documentation are included in the *IMPACT*distribution. In addition, the user will find the AUTHORS file which indicates the primary architects and developers for the suite of tools included in *IMPACT*. The LICENSE file contains the license text, which can also be found at `http://opensource.org/licenses/NCSA`. The README file is a very quick-and-dirty instruction on building *IMPACT*.

B.3 Build *IMPACT*

IMPACT uses Kitware's *CMake* build system, and should build on just about any Unix or Unix-clone system. We've tested it under the GCC, and Intel compilers on several flavors of Linux and MAC OS X. *IMPACT* requires *CMake*-2.8+.

B.4 Prerequisites and TPLs

IMPACT and its various packages have very few dependencies on outside packages. *IMPACT* requires the following support software and third-party libraries:

- C, C++, and Fortran 90 compilers. GCC and Intel are tested.
- *MPICH*-derived MPI can be obtained from `http://www.mpich.org` or `http://mvapich.cse.ohio-state.edu/` for MPI over InfiniBand. As a side note, the system should work with *OpenMPI* but is not well tested. MPI can also be installed by using a Linux system's software package manager.
- *HDF4* can be obtained from `http://www.hdfgroup.org/products/hdf4/` or installed by using Linux distribution package managers.
- *CGNS* (optional) can be obtained from `http://cgns.sourceforge.net/` or installed by using Linux distribution package managers.

These packages will need to be installed before building *IMPACT*.

B.5 Run *CMake*

With the `MPACT_SOURCE` directory all set up and prerequisites installed, the *IMPACT* software is ready to be configured and built. Out-of-source builds are highly recommended for *IMPACT*. To accomplish this, create a build directory that is not a subdirectory of `IMPACT_SOURCE`. It is assumed the user will create an environment variable to store the build directory:

```
IMPACT_BUILD=/full/path/to/mpact_build_directory
```

The user should set the following variables for specification of the compilers:

`CC=mpicc` (or equivalent)
`CXX=mpicxx` (or equivalent)
`FC=mpif90` (or equivalent)

If *HDF4* is installed somewhere other than a system location (e.g., /usr, or /usr/local), then an additional variable should be set:

```
CMAKE_PREFIX_PATH=/path/to/hdf4/install
```

Where *HDF4* libraries should be found in /path/to/hdf4/install/lib. Once these are set up, the user can go into the build directory and run *CMake*:

```
> cd ${IMPACT_BUILD}
> cmake ${IMPACT_SOURCE}
```

If all of the necessary prerequisites are satisfied, the stated commands should complete without error, resulting in *Makefile*s customized for the host environment. To build *IMPACT*, just issue "make":

```
> make
```

This command should complete the build without errors. Presuming this happened, the user will find the following libraries in the build tree:

```
lib/
 |__libSITCOM.so
 |__libSITCOMF.a
 |__libRHDF4.so
 |__libSimIN.so
 |__libSimOUT.so
 |__libSimpal.so
 |__libSIM.so
 |__libSurfMap.so
 |__libSurfUtil.so
 |__libSurfX.so
```

B.6 Use *IMPACT*

IMPACT is not an application, but a software development infrastructure. It is designed to be used from the build directory, and provides its capabilities in libraries that are linked by the user's applications. Assuming the user's software package is named *UserFoo*, the following general steps are taken to integrate *UserFoo*:

1. Prepare the application for integration.
   (a) Massage *UserFoo* architecture so that it consists of a library and a driver which links and drives the library.
   (b) Major part: Make necessary changes to represent interacting interface surfaces as stand-alone, self-descriptive surface mesh.
   (c) Major part: Make necessary changes to support externally supplied boundary solution on interface meshes. The source of the solution and any transformations may be neglected.
2. Implement the Component-side Client in *UserFoo*'s library.
   (a) Implement `UserFoo_load_module` and `UserFoo_unload_module` (see *COM* User's Guide).
   (b) Using *COM* API, create *UserFoo*'s ComponentInterface and register *UserFoo*-native data and functions as needed.
3. Implement a driver making sure it does the following:
   (a) Initializes MPI, if necessary
   (b) Initializes *COM*
   (c) Loads *UserFoo* with `COM_LOAD_STATIC_DYNAMIC(UserFoo,"userfoowindow_name")`

(d) Can access registered DataItems through the CI
(e) Can call *UserFoo* functions through the CI

The *COM* API is the most used part of *IMPACT* for preparing and integrating an existing application. This API is documented in the *COM* User Guide. Once ready to develop a driver, *SIM* may be optionally used to create multiphysics drivers, or the driver can be entirely designed by the user. Again, the *COM* API is used to access integrated applications' data and methods through the *COM* CI. A user's driver may load and use any of the service modules included in *IMPACT* in creating a multiphysics capability. Each package is described in its own documentation included in the *IMPACT* distribution in IMPACT_SOURCE/Documentation.

C COM User's Guide

Component Object Manager Users Guide
Version 0.1.0 IllinoisRocstar LLC October 25, 2016

License

For more information regarding the software, its documentation, or support agreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

C.1 Introduction

Large-scale numerical simulation of a complex system, such as a solid rocket motor (SRM), requires consideration of multiple, interacting physical components, such as fluid dynamics, solid mechanics, and combustion. Numerical simulations of such systems are commonly called *multiphysics* simulations. Because of their multidisciplinary nature, development of such a multiphysics simulation capability typically involves a broad range of expertise and collaboration among many groups or institutions. It is common to take a partitioned approach, in which the individual physics codes are developed more or less independently of one another, and the software integration effort required to orchestrate them into a coherent system.

The objective of the Component Object Manager (COM) is to ease the integration of such independently developed applications into a coherent, integrated software system, particularly in a distributed parallel setting. It is designed to maximize concurrency in development of different applications and components, minimize user effort in software integration, and provide interoperability between different programming languages (in particular, C, C++, and Fortran 90).

The motivating application for the this integration infrastructure is Illinois Rocstar's flagship multiphysics simulation application, *Rocstar*. Originally developed by the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois DOE ASCI Center (`http://www.csar.uiuc.edu`), for simulating SRM, *Rocstar* is now a general multiphysics application and is applicable to fluid-structure interaction (FSI) across a moving, reacting interface.

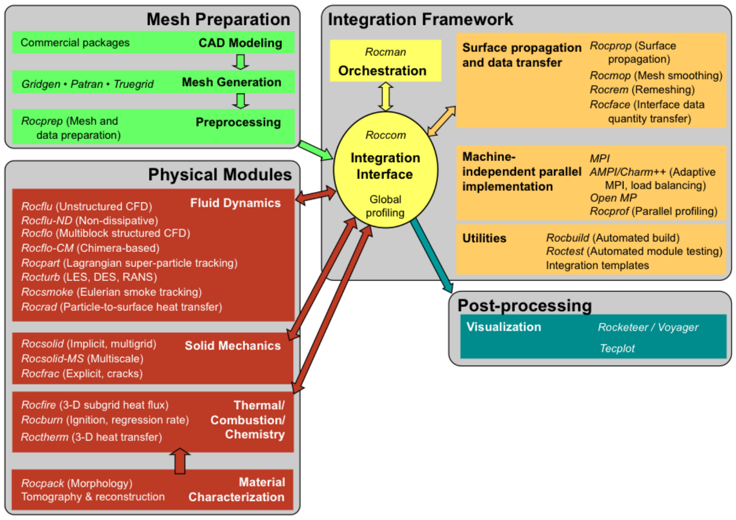

Mesh Preparation
Commercial packages
CAD Modeling
Gridgen • Patran • Truegrid
Mesh Generation
Rocprep (Mesh and data preparation)
Preprocessing
Physical Modules
Rocflu (Unstructured CFD)
Fluid Dynamics
Rocflu-ND (Non-dissipative)
Rocflo (Multiblock structured CFD)
Rocflo-CM (Chimera-based)
Rocpart (Lagrangian super-particle tracking)
Rocturb (LES, DES, RANS)
Rocsmoke (Eulerian smoke tracking)
Rocrad (Particle-to-surface heat transfer)
Rocsolid (Implicit, multigrid)
Solid Mechanics
Rocsolid-MS (Multiscale)
Rocfrac (Explicit, cracks)
Rocfire (3-D subgrid heat flux)
Rocburn (Ignition, regression rate)
Roctherm (3-D heat transfer)
Thermal/ Combustion/ Chemistry
Rocpack (Morphology)
Tomography & reconstruction
Material Characterization
Integration Framework
Rocman
Orchestration
Roccom
Integration Interface
Global profiling
Surface propagation and data transfer
Rocprop (Surface propagation)
Rocmop (Mesh smoothing)
Rocrem (Remeshing)
Rocface (Interface data quantity transfer)
Machine-independent parallel implementation
MPI
AMPI/Charm++ (Adaptive MPI, load balancing)
Open MP
Rocprof (Parallel profiling)
Utilities
Rocbuild (Automated build)
Roctest (Automated module testing)
Integration templates
Post-processing
Visualization
Rocketeer / Voyager
Tecplot

Figure C.52: *Rocstar* architecture. Many software components (i.e. modules) interact through the integration infrastructure.

Figure C.52 shows the overall structure of the current generation of *Rocstar*. In the *Rocstar* architecture, user applications are built as *modules* which are integrated into the composite software system through the *Rocstar* integration interface, *Roccom*. *COM* is the integration interface of *Rocstar* extracted and generalized for use in arbitrary parallel software integration efforts.

COM categorizes modules into two types: *application modules* (including *computation (physics) modules* and *orchestration modules*) and *service modules* In Figure C.52, the boxes on the left show the physics modules. On the top is the *orchestration* module, *Rocman*, which manages the coupling algorithms. On the right are the computer science modules that provide services to the physics and orchestration modules through *COM*. Typically, the physics modules are written in Fortran 90 and the service modules are written in C++. The parallel implementation uses the standard Message Passing Interface (MPI) for all modules.

Although it was motivated specifically by the needs of the rocket simulation application described above, the integration infrastructure we have developed is quite general, and should be equally applicable to many other multiphysics simulations involving multiple, interacting software modules representing various physical components, especially those based on spatial decomposition into geometric domains with associated meshes. *COM* provides systematic methods for modules in a complex simulation to keep track of their data and to access data defined by other modules. Besides declaring variables and allocating buffers, each computation module registers its datasets with *COM*. These datasets can later be retrieved from *COM* by the same module or other modules, using parameters such as data block number, attribute name, etc. Functions can be registered and invoked in a similar way through *COM*. This scheme allows great independence in design and development of individual modules, hides the coding details and potential IP of different modules, and can enable plug-and-play of different modules.

COM is composed of three parts: a simple API (Application Programming Interface) for application modules, a C++ interface for developing service modules, and a runtime system. The API provides subroutines for registering the public data and functions of a module, querying a publicized data of a module, and invoking registered functions. In general, the API is the only part that application code developers need to learn in order to use *COM*. After an application code registers its data with *COM*, it can easily take advantage of the service utilities built on top of *COM*'s developers interface (such as parallel I/O). *COM* also provides support for eliminating global variables from application codes, which is highly desirable in threaded environments. In this documentation, we address only the general concepts of *COM* and its API. For a more in-depth discussion on the developers interface or runtime system, please see the (upcoming) *COM* Developers Guide.

C.2 Overview

COM (standing for Component Object Manager) is a component-based, object-oriented, data-centric software integration infrastructure, which provides a systematic, object-oriented, data-centric approach for inter-module interaction. Using infrastructure constructs, a computation module implements a *Component-side Client* (CSC) which creates distributed objects called *ComponentInterfaces* (CI) and registers its datasets into ComponentInterface instances called *Windows*. With the authorization of their owner modules or the orchestration module, these datasets can later be retrieved from *COM* by other modules using handles provided by *COM*. Functions can be registered and invoked similarly through *COM*. This scheme allows great independence in design and development of individual modules, hides the coding details of different research subgroups, and provides additional features such as automatic tracing and profiling.

C.2.1 Object-Oriented Interfaces

to simplify inter-module interfaces, *COM* utilizes an object-oriented methodology for abstracting and managing the data and functions of a module. This abstraction is mesh- and physics-aware and supports encapsulation, polymorphism, and inheritance.

ComponentInterface Windows and Panes *COM* organizes data and functions into distributed objects called *ComponentInterface* Windows. A CI Window (or simply window) encapsulates a number of *DataItems* (such as the mesh and some associated field variables) and public *functions* of a module, any of which can be empty. DataItems may be gathered together into groups called *DataGroup*s. A window can be partitioned into multiple frames called *Panes*, each of which instantiate a DataGroup. Panes and their DataGroups are useful for exploiting parallelism or for distinguishing different material or boundary-condition types. In a parallel setting, a pane belongs to a single process, while a process may own any number of panes. All panes of a given window must have the same DataGroup, although the total sizes of the Pane's DataGroup DataItems may vary. A module constructs windows inside its CSC at runtime by creating DataItems and registering the addresses of the DataItems and functions. Typically, the DataItems registered with *COM* are *persistent* (instead of temporary) datasets, in the sense that they live throughout the simulation (except that CI windows may need to be reinitialized at some events, such as remeshing). Different modules can communicate with each other only through their CI windows, as illustrated in Figure C.53.

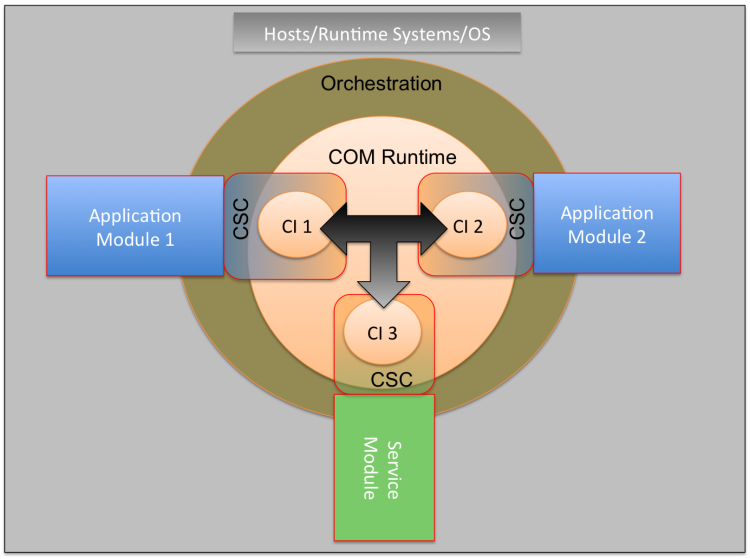


Figure C.53: *COM* architecture with multiple software modules. Each module is loaded at runtime and shares the same process with the orchestrator. All module-module interactions are conducted through the CI and mediated by *COM*.

A code module references CI windows, DataItems, or functions using their names, which are of character-string type. Window names must be unique across all modules, and an DataItem or function name must be unique within a window. A code module can obtain an integer *handle* of (i.e., a reference to) an DataItem/function from *COM* with the combination of the window and DataItem/function names. The handle of an DataItem can be either *mutable* or *immutable*, where an immutable handle allows only read operations to its referenced DataItem, similar to a const reference in C++. Each pane has a user-defined positive integer ID, which must be unique within the window across all processors but need not be consecutive.

DataItems DataItems of a module can include mesh data, field variables, and other data associated with the CI window or pane. The former two types of DataItems are associated with nodes or elements. A nodal or elemental DataItem of a pane is conceptually a two-dimensional dataset: one dimension corresponds to the nodes/elements, and the other dimension corresponds to the data within a node/element. The dataset can be stored in a row- or column-major two-dimensional array, or be stored in separate arrays for each component of the dataset. *COM* allows users to specify a stride (the distance in the base data type, such as int or double precision) between the same component of two consecutive items (such as nodes/elements).

Mesh Data Mesh data include nodal coordinates, pane connectivity, and element connectivity (or simply connectivity), whose DataItem names and data types are predefined by *COM*. The nodal coordinates are double-precision floating-point numbers, with three components per node. The pane connectivity specifies the communication patterns between nodes shared by two or more panes, and is a pane DataItem packed in a 1-D array. The registration of pane connectivity is desirable for many purposes, but it is optional and can be computed automatically from coordinates using Rocmap.

COM supports both surface and volume meshes, which can be either multi-block structured or unstructured with mixed elements. For multi-block meshes, each block corresponds to a pane in a window. For unstructured meshes, each pane has one or more connectivity tables, where each connectivity table contains consecutively numbered elements (i.e., their corresponding field variables are stored consecutively) of the same type. Each connectivity table must be stored in an array with contiguous or staggered layout. To facilitate parallel simulations, *COM* also allows a user to specify the number of layers of ghost nodes and cells for structured meshes, and the numbers of ghost nodes and cells for unstructured meshes.

Field Variables Field variables are nodal or elemental DataItems that have no designated names or data types. A user must first define such an DataItem in the window and then register the addresses of the DataItem for each pane. For a specific pane, if a field variable is stored in one single array, then the array is registered with a single call; if it is stored in multiple arrays, then the user must register these arrays separately.

Windowed and Panel DataItems A data member can also be associated with either the CI window or a pane. Examples of windowed DataItems include data structures that encapsulate the internal states of a module, its CI, or some control parameters. An example of a pane DataItem is an integer flag for the boundary condition type of a surface patch. Similar to field variables, these DataItems do not have designated names or data types, and must be created within a CI window and then registered, or allocated.

Aggregate DataItems In *COM*, although DataItems are registered as individual arrays, DataItems can be referenced as an aggregate. For example, the name "mesh" refers to the collection of nodal coordinates and element connectivity; the name "all" refers to all the data DataItems in a window. For staggered DataItems, one can use "*i*-DataItem" ($i \geq 1$) to refer to the *i*th component of the DataItem or use "DataItem" to refer to all components collectively.

Aggregate DataItems enable high-level inter-module interfaces. For example, one can pass the "all" DataItem of a window to a parallel I/O routine to write all of the contents of a window into an output file with a single call. As another example, it is sometimes more convenient for users to have *COM* allocate memory for data DataItems and have application codes retrieve memory addresses from *COM*. *COM* provides a call for memory allocation, which takes a window DataItem name pair as input. A user can pass in "all" for the DataItem name, which will have *COM* allocate memory for all the unregistered DataItems.

C.2.2 Functions

A CI can contain not only data members but also function members. A module can register a function into its CI window, to allow other modules to invoke the function through *COM*. Registration of functions enables a limited degree of runtime polymorphism. It also overcomes the technical difficulty of linking object files compiled from different languages, where the mangled function names can be platform and compiler dependent.

Member Functions Except for very simple functions, a typical function needs to operate with certain internal states. In object-oriented programs, such states are encapsulated in an "object", which is passed to a function as an argument instead of being scattered into global variables as in traditional programs. In some modern programming language, this object is passed implicitly by the compiler to allow cleaner interfaces.

In mixed-language programs, even if a function and its context object are written in the same programming language, it is difficult to invoke such functions across languages, because C++ objects and F90 structures are incompatible. To address this problem, we introduce the concept of member functions of DataItems into *COM*. Specifically, during registration a function can be specified as the member function of a particular data DataItem within one of its CI windows. *COM* keeps track of the specified DataItem and passes it implicitly to the function during invocation, in a way similar to C++ member functions. Because the caller no longer needs to know the context object of the callee, this concept overcomes the incompatibility without sacrificing object-orientedness.

Optional Arguments *COM* supports the semantics of optional arguments similar to that of C++ to allow cleaner codes. Specifically, during function registration a user can specify the last few arguments as optional. *COM* passes null pointers for those optional arguments whose corresponding actual parameters are missing during invocation.

C.2.3 Inheritance

In multiphysics simulations, inheritance of CI data on the interface surface between domains is useful in many situations. First, the orchestrator sometimes needs to create data buffers associated with a computation module for the manipulation of jump conditions. Inheritance of windows allows the orchestration module to create a new window for extension or alteration, without altering an existing application's CI. Second, a module may need to operate on a subset of the mesh of another module. In rocket simulation, for example, the combustion module needs to operate on the burning surface between the fluid and solid. Furthermore, the orchestrator sometimes needs to split a user-defined CI into separate windows based on boundary-condition types, so that these subwindows can be treated differently (e.g., written into separate files for visualization). Figure C.54 depicts a scenario of inheritance among three windows.

To support these needs, *COM* allows inheriting the mesh from a parent window to a child window in either of two modes. First, the mesh can be inherited as a whole. Second, only a subset of panes that satisfy a certain criterion are inherited. After inheriting mesh data, a child window can inherit data members from its parent window, or other windows that have the same mesh (this allows for *multiple inheritance*). The child window obtains the data only in the panes it owns and ignores other panes. During inheritance, if an DataItem already exists in a child window, *COM* overwrites the existing DataItem with the new DataItem.

COM supports two types of inheritance for data members: cloning (with duplication) and using (without duplication). The former allocates new memory space and makes a copy of the data DataItem in the new window, and is safer in terms of data integrity. The latter makes a copy of the references of the data member, which avoids the copying overhead associated with cloning and guarantees data coherence between the parent and child, and is particularly useful for implementing orchestration modules.

C.2.4 Data Integrity

In complex systems, data integrity has profound significance for software quality. Two potential issues can endanger data integrity: dangling references and side effects. *COM* addresses these issues through the mechanisms of persistency and immutable references, respectively.

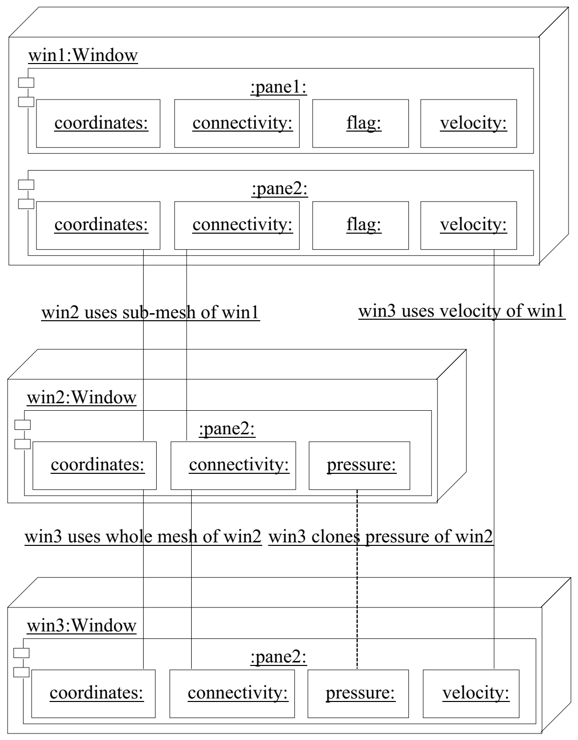


Figure C.54: Scenario of inheritance of mesh and field DataItems among three CI windows.

Persistency *COM* maintains references to the datasets registered with its CI. To avoid dangling references associated with data registration, *COM* imposes the following persistency requirement: the datasets registered with a CI window must outlive the life of the window. Under this model, any persistent object can refer to other persistent objects without the risk of dangling references. In a heterogeneous programming environment without garbage collection, persistency cannot be enforced easily by the runtime systems instead, it is considered as a design pattern that application code developers must follow.

Immutable References Another potential issue for data integrity is side effects due to inadvertent changes to datasets. For the internal states of the modules, *COM* facilitates the traditional integrity model through member functions described earlier. In *COM*, a service module can obtain accesses to another module's data DataItems only through its function arguments, and *COM* enforces at runtime that an immutable handle cannot be passed to mutable arguments.

C.3 Architecture of *COM*

The core of *COM* is composed of three parts: an Application Programming Interface (API), a C++ class interface for development of service modules, and a runtime system for the bookkeeping associated with data objects and invocation of functions.

C.3.1 *COM* API

The *COM* API supplies a set of primitive function interfaces to physics and service modules and orchestrators for system setup, CI management, information retrieval, and function invocation. The subset of the API for CI management serves essentially the same purpose as the Interface Definition Language (IDL) of other frameworks (such as CCA), except that *COM* parses the definitions of the CI at runtime. *COM* provides different bindings for C++ and F90, with similar semantics. See Section C.5 for details.

C.3.2 C++ Class Interfaces

COM provides a unified view of the organization of distributed data objects for service modules through the abstractions of CI windows and panes. Internally, *COM* organizes windows, panes, DataItems, functions, and connectivities into C++ objects, whose associations are illustrated in Figure C.55,on a UML class diagram.

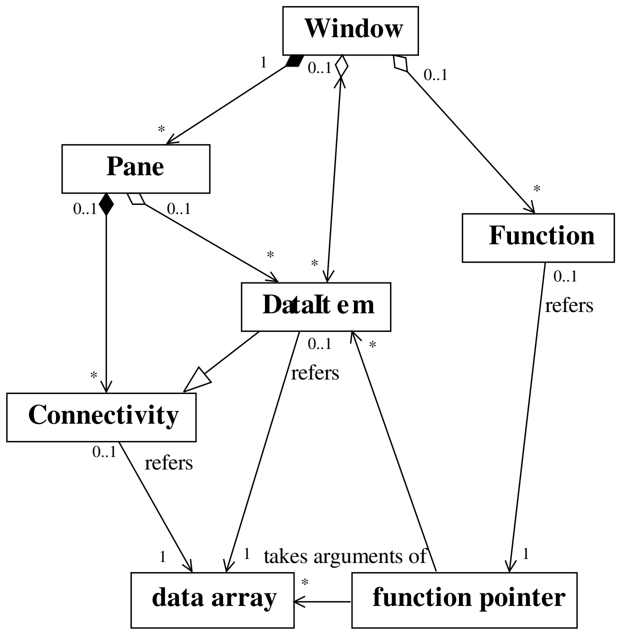


Figure C.55: UML associations of *COM*'s classes.

A ComponentInterface window maintains a list of its local panes DataItems, and functions; a Pane object contains a DataGroup which is a list of DataItems and connectivities; a DataItem object contains a reference to its owner window. By taking references to DataItems as arguments, a function can follow the links to access the data DataItems in all local panes. The C++ interfaces conform to the principle of deeply immutable references, ensuring that a client can navigate through only immutable references if the root reference was immutable. Through this abstraction, the developers can implement service utilities independently of application codes, and ensure applicability in a heterogeneous environment with mixed meshes, transparently to physics modules.

C.3.3 *COM* Runtime System

The runtime serves as the middleware between modules. It keeps track of the user-registered data and functions. During function invocation, it translates the function and DataItem handles into their corresponding references with an efficient table lookup, enforces access protection of the DataItems, and checks whether the number of arguments of the caller matches the declaration of the callee. Furthermore, the runtime system also serves as the middleware for transparent language interoperability. For example, if the caller is in F90 whereas the callee is in C++, the runtime system will null-terminate the character strings in the arguments before passing to the callee.

Through the calling mechanism, *COM* also provides tracing and profiling capabilities for inter-module calls to aid in debugging and performance tuning. It also exploits hardware counters through PAPI to obtain performance data such as the number of floating-point instructions executed by modules. A user can enable such features using command-line options without additional coding.

C.4 Module Requirements

A *COM* application has a driver or an orchestrator, which is responsible for system setup and invoking the registered functions in turn. Each *COM*-compliant module must provide a load-module routine, which creates a CI window to encapsulate its interface functions and context objects, and an unload-module, which destroys the window, where the window name is typically the same as that of the module. By calling the load-module routines, the driver dynamically loads a set of modules into the runtime system. Through *COM*'s calling mechanism, the orchestrator then invokes the functions of the physics and service modules, which in turn can also invoke functions provided by other modules.

C.5 COM API

COM provides different bindings for C, C++, and Fortran 90, with similar semantics, except that C/C++ is case-sensitive whereas Fortran is case-insensitive, and C/C++ passes arguments by value whereas Fortran passes by reference. Another subtle difference is that Fortran character strings, which are not null terminated by default, must be interpreted differently from C/C++ character strings. These differences are apparent in the prototype definitions of the subroutines, but are mostly transparent to users. COM's interface prototypes are defined in "com.h" (for C/C++) and "comf90.h" (for Fortran 90), which must be included by the codes in corresponding languages, respectively.

COM's interface subroutines follow the following conventions: They all start with the prefix COM_, followed by lower-case letters (for C/C++). Most subroutines return no values unless otherwise specified. If a non-fatal error occurred inside a COM subroutine, an error flag will be set. For the C and Fortran interfaces, the error code can be obtained by calling **COM_get_error_code**. For the C++ interface, the error code will be thrown as an exception.

Although COM's API has about 40 functions, a simple computation module needs to use only about 10 of them, mostly in Section C.5.2. The other functions are more advanced and provided mostly for the orchestrators and for more complex physics modules.

C.5.1 Initialization and Finalization

Startup and Shutdown of COM Implementations of COM's runtime system require some setup operations before any other COM operations can be performed. To provide for this, COM includes an initialization subroutine **COM_init**.

C:	**COM_init**(int *argc, char ***argv)
Fortran:	**SUBROUTINE COM_INIT**

This subroutine must be called exactly once from every process before any other COM subroutine (apart from **COM_initialized**) is called. It is typically called from the driver routine of the application. The C version accepts the arguments argc and argv, which are the arguments of the main routine of C. **COM_init** parses the following options:

- "-com-v *n*": Set the verbose level of all processes to *n* (see **COM_set_verbose**);
- "-com-v*p n*": Set the verbose level of process *p* to *n* (see **COM_set_verbose**);
- "-com-mpi": Call MPI_Init within COM_init.
- "-com-home <directory>":Search for shared libraries under <directory>/lib. Alternatively, one can pass the directory by setting either COM_HOME or ROCSTAR_HOME enrionment variables.

The Fortran version **COM_INIT** takes no arguments. A corresponding subroutine **COM_finalize** is also provided for COM to clean up its state after the execution of the program. It also needs to be called on every process. Once this subroutine is called, no COM subroutine may be called.

C: **COM_finalize**(void)

Fortran: **SUBROUTINE COM_FINALIZE**

The following is a piece of code in C that illustrates its usage.

```
int main(int argc, char **argv) {
  COM_init(&argc, &argv);
  /* main program */
  COM_finalize();
}
```

COM provides a subroutine **COM_initialized** for checking whether **COM_init** has been called.

C: int **COM_initialized**(void)

Fortran: **FUNCTION COM_INITIALIZED()**
INTEGER :: **COM_INITIALIZED**

The argument flag is set to nonzero if **COM_init** has been called and zero otherwise.

Furthermore, COM runtime environment can also be shut down abnormally by calling **COM_abort**.

C: void **COM_abort**(int ierr)

Fortran: **SUBROUTINE COM_ABORT**(IERR)
INTEGER, INTENT(IN) :: IERR

This function terminates a COM program and returns the error code ierr to the invoking environment. If MPI was initialized, then **COM_abort** calls MPI_Abort internally on MPI_COMM_WORLD. Otherwise, it calls the exit function of the standard C library. For Fortran codes, COM also provides a related subroutine **COM_CALL_EXIT**, which we describe in Section C.6.6.

Loading and Unloading of Modules In the COM infrastructure, user applications can be built into a dynamic library named "libfoo.so" (where "foo" is arbitrary). The dynamics libraries are called *modules*, and COM provides the following interface to load and unload the dynamic library for a module.

C:	**COM_load_module**(const char *modName, const char *winName)
Fortran:	**SUBROUTINE COM_LOAD_MODULE**(MODNAME, WINNAME) CHARACTER(*), INTENT(IN) :: MODNAME, WINNAME
C:	**COM_unload_module**(const char *modName, const char *winName=NULL)
Fortran:	**SUBROUTINE COM_UNLOAD_MODULE**(MODNAME,WINNAME) CHARACTER(*), INTENT(IN) :: MODNAME,WINNAME OPTIONAL :: WINNAME

The argument modName is the main part of the library name (e.g., "foo" for libfoo.so). The "foo" module needs to supply two subroutines, **foo_load_module** and **foo_unload_module**, which takes winName as its argument. When **COM_load_module/** or **COM_unload_module** is called, COM locates the symbol **foo_load_module/foo_unload_module** in libfoo.so, respectively, and invokes these user-provided routines by passing winName to load/unload the module. For **COM_unload_module**, the winName argument can be omitted if the module is loaded only once.

For ease of debugging, sometimes it is desirable to build COM modules and applications statically. In this case, an application need to define the prototypes of **foo_load_module** and **foo_unload_module** and call them directly instead of through **COM_load_module** or
COM_unload_module. COM provides the following macros to C/C++ codes (defined when "com.h" is included):

COM_EXTERN_MODULE(modName_noquotes)

COM_LOAD_MODULE_STATIC_DYNAMIC(modName_noquotes, winName)

COM_UNLOAD_MODULE_STATIC_DYNAMIC(modName_noquotes, winName)

Depending on whether the macro **STATIC_LINK** is defined (e.g., by passing -DSTATIC_LINK to the compiler) or not, these macros expands to different statements. When **STATIC_LINK** is defined, then **COM_EXTERN_MODULE**(foo) defines **foo_load_module** and **foo_unload_module**. Otherwise, it expands to noop. For C++ codes, since **COM_EXTERN_MODULE** uses the extern "C" modifier, it cannot be used inside a function. **COM_LOAD_MODULE_STATIC_DYNAMIC**(foo, "FOO") expands to **foo_load_module**("FOO") if the **STATIC_LINK** is defined, and to **COM_load_module**("foo","FOO"), otherwise; similarly for **COM_UNLOAD_MODULE_STATIC_DYNAMIC**.

C.5.2 Data and Function Registration

COM organizes data and functions into CI windows. A window encapsulates a number of data members (such as the mesh and some associated data DataItems) and public functions of a module. A module can create any number of CI windows. Technically, an application has only one CI with multiple windows, but this distinction is currently irrelavent. All panes of a window must have the same DataGroup, although the size of each DataItem may vary.

C.5.3 Creation of CI Window

A call to **COM_new_window** creates an empty window with a given name.

C++: **COM_new_window**(const std::string wName, MPI_Comm comm=MPI_COMM_SELF)

C: **COM_new_window**(const char *wName, MPI_Comm comm=MPI_COMM_SELF)

Fortran:
```
SUBROUTINE COM_NEW_WINDOW(WNAME, COMM)
CHARACTER(*), INTENT(IN) :: WNAME
OPTIONAL, INTEGER, INTENT(IN) :: COMM
```

The wName argument is a character string and must be unique across all modules, and comm is the MPI communicator of the owner processes of the window. Because a window is a collective concept, this subroutine should be called on all processes within the MPI communicator. The communicator can be retrieved by calling **COM_get_communicator** (see subsection C.8).

After a window is created, a user can create data DataItem members and register addresses of data and functions to it as described later, followed by calling **COM_window_init_done** to mark the end of the registration of a window.

C++: **COM_window_init_done**(const std::string name, int clct=1)

C: **COM_window_init_done**(const char *name, int clct=1)

Fortran:
```
SUBROUTINE COM_WINDOW_INIT_DONE(NAME, CLCT)
CHARACTER(*), INTENT(IN) :: NAME
INTEGER, INTENT(IN), OPTIONAL :: CLCT
```

It also takes the window name as its first argument. The second argument clct specifies whether the function is being called collectively on all the owner processes of the window, so that the mapping from panes to processes can be determined. If any pane was created or deleted, then **COM_window_init_done** must be called collectively before the window is used, by passing a non-zero value (the default) to clct. After calling **COM_window_init_done**, the sizes and arrays of the DataItems can be changed without calling **COM_window_init_done** again. DataItems and/or panes can be added or deleted, given that **COM_window_init_done** is called after the changes. If the DataItems were changed but no panes were added or deleted from any process, then **COM_window_init_done** can be called with clct equal to zero, to avoid recomputing the pane-to-process mapping.

DataItems

Declaration of New DataItems Besides mesh data, a window can have other data members, which can be associated with the window, a pane, nodes, or elements of the pane. Different from keywords, these data DataItems do not have designated names or data types. Therefore, a user must first define an DataItem by calling **COM_new_dataitem** before registering the addresses of the DataItem. Again, this subroutine must be called collectively on all owner processes of a window.

C++: **COM_new_dataitem**(const std::string aName, char loc, COM_Type type,
int ncomp, const char *unit)

C: **COM_new_dataitem**(const char *aName, char loc, COM_Type type, int ncomp, const char *unit)

Fortran: **SUBROUTINE COM_NEW_DATAITEM**(ANAME, LOC, TYPE, NCOMP, UNIT)
CHARACTER(*), INTENT(IN) :: ANAME, FNAME, UNIT
CHARACTER*1, INTENT(IN) :: LOC
INTEGER, INTENT(IN) :: TYPE, NCOMP

In the argument list, aName is the DataItem name in the format of "window DataItem", similar to mesh data names; loc can be either '**w**', '**p**', '**n**', or '**e**', corresponding to *windowed*, *panel*, *nodal*, or *elemental* data; type specifies the base datatype of the DataItem, which can be one of the following constant C/C++ Data types:

- COM_CHAR
- COM_UNSIGNED_CHAR
- COM_BYTE
- COM_SHORT
- COM_UNSIGNED_SHORT
- COM_INT
- COM_UNSIGNED
- COM_LONG
- COM_UNSIGNED_LONG
- COM_FLOAT
- COM_DOUBLE
- COM_LONG_DOUBLE
- COM_BOOL

or Fortran Data types:

- COM_CHARACTER
- COM_LOGICAL
- COM_INTEGER
- COM_REAL
- COM_DOUBLE_PRECISION
- COM_COMPLEX
- COM_DOUBLE_COMPLEX

or other:

- COM_MPI_COMMC
- COM_MPI_COMM(=COM_MPI_COMMC)
- COM_MPI_COMMF
- COM_MAX_TYPEID(=COM_MPI_COMMF)
- COM_STRING
- COM_RAWDATA
- COM_METADATA
- COM_VOID
- COM_F90POINTER
- COM_OBJECT
- COM_MIN_TYPEID

ncomp is the number of components of the DataItem (for example, the number of entries associated with each node/element for nodal/elemental data); and unit is the unit of the DataItem, which can be the empty string "" if the DataItem is unitless.

One can call COM_new_dataitem on an existing DataItem to re-define an DataItem (including the keyword "nc"). However, one cannot increase the number of components of the predefined DataItems (see next subsection). After calling COM_new_dataitem, the previously registered data are reset, and its handles and inherited DataItems become invalid. It is the user's responsibility to ensure the consistency for the DataItem.

Predefined Mesh Data Mesh data, including nodal coordinates, pane connectivity, and element connectivity (or simply connectivity), have predefined DataItem names and data types. Nodal coordinates ("nc") are predefined as double-precision nodal DataItems with three components (corresponding to x, y, and z, respectively) per node and a default unit "m". However, nodal coordinates may be redefined by calling COM_new_dataitem to have less than 3 components, a different base data type, or a different unit. Pane connectivity ("pconn") is predefined as a 1-D integer pane DataItem with no unit.

"nc" for nodal coordinates
"pconn" for pane connectivity

For each pane, the pane-connectivity array can have multiple blocks:

1. shared nodes;
2. real nodes to send;
3. ghost nodes to receive;

4. real cells to send;
5. ghost cells to receive.

The first block goes into the real part of pconn and blocks 2–5 go into the ghost part. Blocks 2 and 3 must be present together, so are blocks 4 and 5. The ghost part of pconn is optional, and within the ghost part of pconn, blocks 4 and 5 are optional. Each block has the following content:

<number of communicating-pane blocks to follow>
<communicating pane id 1>
<#local nodes to follow>
<list of local nodes>...
... ! repeat for other remote panes

The lists of nodes for a pair of communicating panes are stored in the same order in their corresponding tables. Furthermore, the panes are stored in increasing order of pane IDs. If a node is shared by more than two panes, then every pair of shared nodes is stored in pconn. Note that it is possible for a single pane to have duplicated nodes, for example, in the case of branch-cut for structured meshes. In this case, in the block for shared nodes, the list of local nodes is composed of a series of node pairs, where the first node in the pair always has a smaller node ID than the second, and the number of local nodes is equal to twice the number of pairs. Note that in the case of partial inheritance, where a subwindow may inherit a subset of panes from a parent window, pconn may be inherited by the subwindow for its inter-pane communication, and as a consequence pconn may refer to some remote panes that no longer exist. In this case, it is important to note that the first number in each block is no longer the actual number of communicating panes, and a traversal of pconn should skip the nonexisting remote panes.

The names of element connectivities have the format of "**:***elementtype***:***aname*", where the ":*aname"* part is optional and is useful when there are multiple connectivity tables for one type of elements. Note that element connectivities are not regular DataItems, in that different panes may contain different types of elements, and an element connectivity must not be created by calling COM_new_dataitem but by setting its size and registering its address.

"**:st1**:*aname*", "**:st2**:*aname*", "**:st3**:*aname*" for structured mesh of 1, 2 and 3 dimensions.
"**:b2**:*aname*" and "**:b3**:*aname*" for 2- and 3-node bar elements.
"**:t3**:*aname*", "**:t6**:*aname*", "**:q4**:*aname*", "**:q8**:*aname*", and "**:q9**:*aname*" for connectivity tables of 3- and 6-node triangles, and 4-, 8-, and 9-node quadrilaterals, respectively.
"**:T4**:*aname*", "**:T10**:*aname*", "**:B8**:*aname*" ("**:H8**:*aname*"), and "**:B20**:*aname*" for connectivity tables of 4- and 10-node tetrahedra, and 8- and 20-node bricks, respectively
"**:P5**:*aname*", "**:P14**:*aname*", "**:P6**:*aname*" ("**:W6**:*aname*"), "**:P15**:*aname*" ("**:W15**:*aname*"), and "**:P18**:*aname*" ("**:W18**:*aname*") for connectivity tables of 5- and 14-node pyramids and 6-, 15-, and 18-node prisms (aka pentahedra or wedges), respectively.

For elements of unstructured meshes, COM uses the same numbering convention as the CFD General Notation System (CGNS), of which a detailed description can be found in Section 3.3 of CGNS Standard Interface Data Structures (`http://www.grc.nasa.gov/WWW/cgns/sids/sids.pdf`). If a pane has multiple connectivity tables, these tables must be registered in increasing order of the element numbering (i.e., the elements with smaller indices in field-variable arrays must be registered earlier), and ghost elements must be registered last. Note that for structured meshes, a pane can register only one connectivity using **COM_set_array_const** described in the next subsection, by passing in the numbers of nodes of all directions in a single array listed in Fortran convention (See example code in subsection C.5.5). We will allow for users to add new element types in future releases.

Registration of Sizes One sets the sizes of an DataItem using the following routine.

C++: **COM_set_size**(const std::string aName, int pane_id, int size, int ng=0)

C: **COM_set_size**(const char *aName, int pane_id, int size, int ng=0)

Fortran: **SUBROUTINE COM_set_size**(ANAME, PANE_ID, SIZE, NG)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID, SIZE, NG
OPTIONAL :: NG

In the arguments, aName is the data name in the format of "window.aname" or "window.:elementtype:aname" for connectivity tables paneID is a user-defined *positive* integer identifier of the pane, which must be unique within the window across all processors but need not be consecutive. Window DataItems should be registered with pane-ID 0. The argument size is either the total number of nodes (including ghost nodes) in the pane (for nodal coordinates), or the number of elements (including ghost elements, for a connectivity table), or the length of the dataset for panel or windowed DataItems. ng (optional in F90 and C++; default is 0) is either the number of ghost nodes in the pane for nodal data or the number of ghost elements for a connectivity table.

The default size of a window DataItem is 1, but is undefined for other types of DataItems. Note that setting the number of nodes for one nodal DataItem affects all other nodal DataItems, and it is more efficient to set size for "nc". Typically, one should set the number of elements for each element connectivity.

Registration of Preallocated Array After creating an DataItem, a user can register the address or addresses of the DataItem using the following subroutine.

C++: **COM_set_array**(const std::string aName, int paneID, void *addr,
int stride=0, int cap=0)

C: **COM_set_array**(const char *aName, int paneID, void *addr,
int stride=0, int cap=0)

Fortran: **SUBROUTINE COM_SET_ARRAY**(ANAME, PANEID, ADDR, STRIDE, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
<TYPE> :: ADDR
OPTIONAL :: STRIDE, CAP

As for **COM_set_size**, the aName is either an DataItem name or the name to a connectivity table, and the paneID is a positive integer ID for the owner pane or 0 for window DataItems. The addr argument specifies the address of the array for the DataItem. If the components of each item are stored contiguously in an array of Array(stride,cap) in Fortran convention with stride>=ncomp and cap>=size, one can register the array by with a single call. If it is stored in an array of Array(cap, ncomp), then the stride should be set to 1. The stride argument can be omitted if stride is equal to ncomp, and it is invalid if stride is greater than 1 but smaller than ncomp. **In Fortran 90, it is very important not to register a scalar variable defined locally in a subroutine or function (i.e., a stack variable), unless it has the TARGET or POINTER property.**

The cap argument can be omitted if cap==size. Otherwise, the user must register an array for each individual component of the DataItem using DataItem name in the format "window.*i*-DataItem", where *i* is an integer between 1 and the number of components of the DataItem (Not applicable for connectivity tables). One can change the sizes and the arrays by calling **COM_set_size** and **COM_set_array**.

To protect data integrity, COM allows registration of a read-only data by calling **COM_set_array_const**, which takes the same arguments as **COM_set_array**.

C++: **COM_set_array_const**(...)

C: **COM_set_array_const**(...)

Fortran: **SUBROUTINE COM_SET_ARRAY_CONST**(...)

For Fortran 90, the types supported are scalars and pointers to 1-, 2-, and 3-dimensional integer, single-precision, and double-precision arrays. For other types of variables (such as a function pointer), one can register using one of the following two functions.

Fortran:
```
SUBROUTINE COM_SET_EXTERNAL(ANAME, PANEID, VAR)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
EXTERNAL VAR
```

Fortran: **SUBROUTINE COM_SET_EXTERNAL_CONST**(ANAME, PANEID, VAR)

One can obtain a pointer set by **COM_set_array_const** or **COM_SET_EXTERNAL** only through **COM_get_array_const**.

Registration of Bounds A user can register the lower and upper bounds of a specific DataItem. One can register two sets of bounds: one set of hard bounds, which specifies the universal limits that the dataset must satisfy at all times and whose violation would result in runtime errors; the second set corresponds to soft bounds, whose violations would result in printing of warning messages at runtime.

C++:
```
COM_set_bounds(const std::string aName, int pane_id,
const void *lbnd, const void *ubnd, int is_soft=0)
```

C:
```
COM_set_bounds(const char *aName, int pane_id,
const void *lbnd, const void *ubnd, int is_soft=0)
```

Fortran:
```
SUBROUTINE COM_set_bounds(ANAME, PANE_ID, LBND, UBND, IS_SOFT)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
<TYPE>, INTENT(IN) :: LBND, UBND
INTEGER, INTENT(IN), OPTIONAL :: IS_SOFT
```

When pane_id is 0, then the given bounds will be applied to all panes; if it is greater than 0, then they will be applied to the specific pane with the given pane ID. If the function is called for a vector DataItem, then the bounds apply to the magnitude of the vectors. One can also set the bounds for individual components by calling the function on the corresponding component DataItems (using DataItem names "window.*i*-DataItem"). The fifth argument, is_soft, which is optional with default value 0, specifies whether the bounds are hard (is_soft==0) or soft (is_soft$\neq$0).

C.5.4 Functions

A window can contain not only data members but also function members. A function is registered into a window by calling **COM_set_function** on all owner processes of the window.

C: **COM_set_function**(const char *fName, void (*faddr)(),
const char* intents, COM_Type types[])

Fortran: **SUBROUTINE COM_SET_FUNCTION**(FNAME, FADDR, INTENTS, TYPES)
CHARACTER(*), INTENT(IN) :: FNAME, INTENTS
EXTERNAL FADDR
INTEGER, INTENT(IN) :: TYPES

Similar to DataItem names, fName has the format of "window.function". The argument faddr takes the actual function pointer. For the C interface, to register a function that takes at least one argument (note that all arguments must be pointers), a user code must cast the pointer to the void (*)() type, which is predefined as COM_Func_ptr. The argument intents is a character string of length equal to the number of arguments taken by the registered function, and its *i*th character indicates whether the *i*th argument is for input, output, or both if intents$_i$ is '**i**'/'**I**', '**o**'/'**O**', or '**b**'/'**B**', respectively (see the Optional Arguments paragraph of this section for more discussion). The argument types is an integer array of length also equal to the number of arguments, and its *i*th entry indicates the data type of the *i*th argument. All arguments of a registered function must be passed by reference. If a function is expecting an integer pointer/reference for its *i*th argument, for example, the *i*th entry should be either COM_INT (for C/C++) or COM_INTEGER (for Fortran).

See the section on DataItems above for a list of supported data types.

Member Function Many functions perform operations in a specific context. In object-oriented programs, such contexts are typically encapsulated in objects instead of being scattered into global variables as in traditional programs. Such an object is passed into a function as an argument, and frequently is passed implicitly by the compiler to allow cleaner interfaces in modern programming languages.

To encourage object-oriented programming and cleaner interfaces of application codes, COM supports the concept of member functions of DataItems. A user registers a member function using the interface **COM_set_member_function**, which takes an DataItem name as an additional argument.

C: **COM_set_member_function**(const char *fName, void (*faddr)(),
const char* aName, const char* intents, COM_Type types[])

Fortran: **SUBROUTINE COM_SET_MEMBER_FUNCTION**(FNAME, FADDR, ANAME, INTENTS, TYPES)
CHARACTER(*), INTENT(IN) :: FNAME, ANAME, INTENTS
EXTERNAL FADDR
INTEGER, INTENT(IN) :: TYPES(:)

The given DataItem should encapsulate the context of the registered function. The first entries in intents and types should specify the intention and data type of this DataItem, respectively. When an application code invokes a registered member function through COM, it will not list this DataItem in the arguments, but COM will pass it implicitly as the first argument to the function.

In addition, COM also provides a function for registering C++ member functions of a class, which must be a derived class of COM_Object. An object of a derived class of COM_Object, especially those with virtual functions, must be registered and retrieved using **COM_set_object**() and **COM_get_object**(), which takes the same arguments as **COM_set_array**() and **COM_get_array**(). The member functions are registered using the following interface,

C++: **COM_set_member_function**(const char *fName, void (COM_Object::*faddr)(), const char* aName, const char* intents, COM_Type types[])

and must be casted to the void (COM_Object::*faddr)() type, which is predefined as COM_Member_func_ptr. For example, a member function func of a class Rocfoo can be casted as

reinterpret_cast<COM_Member_func_ptr>(&Rocfoo:func).

Optional Arguments If a registered function is written in C or C++, the last few arguments can be specified as *optional*. COM will pass in null pointers for them if the caller omit these arguments. To specify an argument to be optional, a user should use uppercase letters 'I', 'O', or 'B' instead of 'i', 'o', or 'b' in its corresponding entry in intents.

Data Types As we noted earlier, all arguments of a registered data must be passed by reference. A primitive data type (such as COM_INT) used in the argument types would indicate that the function is expecting a pointer or reference to that type. There are three special cases, however. First, if a function is expecting a character string (vs. a single character), which must be null terminated for C/C++ functions or whose length must be passed in implicitly for Fortran functions, then the corresponding data type of the argument must be set to **COM_STRING**. This data type tells COM to adapt the string if necessary (such as null-terminating the charactering string) to bridge C/C++ and Fortran transparently from users. Second, if a function is a service utility written in C++ and is expecting a C++ object that contains the description of an DataItem, the corresponding data type of the argument must be set to **COM_METADATA**. If the function is expecting the physical address of a window DataItem, then the corresponding datatype should be **COM_RAWDATA**. In general, two types of arguments should use **COM_RAWDATA**: the implicit argument for a member function, and an argument that is a function pointer.

Limitations and Special Notes For language interoperability, a registered function must return no value, and all its arguments must be passed by reference (i.e., must be pointers/references for C/C++ functions). Due to technical reasons, COM has to impose a limit on the maximum number of the arguments that a registered function can take, and the limit is currently set to 7, including the implicit arguments passed by COM, i.e., the first argument of member functions and character lengths for Fortran functions. This preset limit is large enough for most applications, but can be enlarged by changing COM's implementation if desired. Similar to DataItems, a function can be registered multiple times, but only the address of the last registration will be used.

C.5.5 Example Code

The following is a piece of Fortran code segment that demonstrates the registration of data and functions.

```
INTEGER           ::  nn1, ni2, nj2    ! sizes of nodes
INTEGER           ::  ne1              ! sizes of elements
INTEGER           ::  types(2), dims(2)
INTEGER, POINTER           ::  conn1(3,ne1)
DOUBLE PRECISION, POINTER  ::  coors1(3,nn1), coor2(3,ni2, nj2)
DOUBLE PRECISION, POINTER  ::  disp1(3,nn1), disp2(3,ni2, nj2)
DOUBLE PRECISION, POINTER  ::  velo1(ne1,3), velo2(ni2-1, nj2-1,3)
EXTERNAL   fluid_update

CALL COM_NEW_WINDOW("fluid", MPI_COMM_WORLD)

! Create a node-centered double-precision dataset
CALL COM_NEW_DATAITEM( "fluid.disp", "n", COM_DOUBLE, 3, "m")

! Create a element-centered double-precision dataset
CALL COM_NEW_DATAITEM( "fluid.velo", "e", COM_DOUBLE, 3, "m/s")

! Create a pane with ID 11 of a triangular surface mesh
CALL COM_SET_SIZE("fluid.nc", 11, nn1)
CALL COM_SET_ARRAY("fluid.nc", 11, coors1, 3)
CALL COM_SET_SIZE("fluid.:t3:", 11, ne1)
CALL COM_SET_ARRAY("fluid.:t3:", 11, conn1, 3)

! Create a pane with ID 21 of a structured surface mesh
dims(1)=ni2; dims(2)=nj2;
CALL COM_SET_ARRAY_CONST("fluid.:st2:actual", 21, dims)
CALL COM_SET_ARRAY("fluid.nc", 21, coors2, 3)

! Register addresses of DataItems for both panes
CALL COM_SET_ARRAY("fluid.disp", 11, disp1)
CALL COM_SET_ARRAY("fluid.velo", 11, velo1, 1) ! Staggered layout
CALL COM_SET_ARRAY("fluid.disp", 21, disp2)
CALL COM_SET_ARRAY("fluid.velo", 21, velo2, 1) ! Staggered layout

! Register a function that takes two input arguments
type(1)=COM_DOUBLE; type(2)=COM_DOUBLE
CALL COM_SET_FUNCTION("fluid.update", fluid_update, "ii", types)

CALL COM_WINDOW_INIT_DONE("fluid")
......
CALL COM_DELETE_WINDOW("fluid")
```

C.6 Procedure Calls

C.6.1 DataItem and Function Handles

A handle is an integer from which COM can obtain the actual data about DataItems and functions. A user can obtain a mutable handle to an DataItem using **COM_get_dataitem_handle**, or an immutable handle using **COM_get_dataitem_handle_const**, which take the same arguments.

C++: int **COM_get_dataitem_handle**(const std::string aName)

C: int **COM_get_dataitem_handle**(const char *aName)

Fortran: **FUNCTION COM_GET_DATAITEM_HANDLE**(ANAME)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER :: **COM_get_dataitem_handle**

The function can be called on user-defined DataItems, or a pre-defined DataItem "**nc**", "**conn**", "**pconn**", "**mesh**", "**pmesh**", "**data**", and "**all**", which refer to nodal coordinates, element connectivity, pane connectivity, mesh data (including coordinates and element connectivity), parallel mesh data (including mesh and pane connectivity), all field DataItems (excluding parallel mesh), and all DataItems, respectively. Note that it is illegal to call **COM_get_dataitem_handle** on connectivity tables, whose scopes are within panes instead of within windows.

To obtain a handle to a function, one should use **COM_get_function_handle** instead, whose prototype is essentially the same as **COM_get_dataitem_handle**.

If the function or DataItem exists, then a positive integer ID will be returned; otherwise, 0 will be returned. So these functions can be used to detect the existence of a function or DataItems. Similarly, one can detect the existence of a window by calling **COM_get_window_handle**.

C.6.2 Invocation

To invoke a function registered with COM in C or Fortran, a user need to use the following function.

C: **COM_call_function**(int fHandle, int argc, void *arg1, ...)

Fortran: **SUBROUTINE COM_CALL_FUNCTION**(FHANDLE, ARGC, ARG1, ...)
INTEGER, INTENT(IN) :: FHANDLE, ARGC
<TYPE> :: ARG1, ...

The first argument is a function handle, and the second DataItem is the number of arguments to be passed, followed by the pointers (or references) to the data values or DataItem handles.

For C++, we take advantage of the function overloading feature of the language to provide a cleaner interface **COM_call_function**.

C++: **COM_call_function**(int fHandle, void *arg1, ...)

It does not require passing the number of arguments.

C.6.3 Call Tracing

To help debugging application codes, COM allows users to trace the procedure calls by setting a nonzero verbose level.

C: **COM_set_verbose**(int v)

Fortran: **SUBROUTINE COM_SET_VERBOSE**(V)
INTEGER, INTENT(IN) :: V

If v is a positive number, then COM will print out traces of the calls up to depth $(v+1)/2$. If v is an odd number, COM will print only the names of the functions if v even, COM will also print the data types and values of the arguments passed to the functions.

C.6.4 High-Level Profiling

COM contains a simple profiling tool for timing the execution times of the functions invoked through COM.

C: **COM_set_profiling**(int enable)

Fortran: **SUBROUTINE COM_SET_PROFILING**(ENABLE)
INTEGER, INTENT(IN) :: ENABLE

If enable is zero, it disables profiling; otherwise, it enables profiling and resets all the counters of the profiler.

In a parallel run, the timing results are typically more accurate if MPI_Barrier is called before and after a function call, but putting too many barriers may also affect performance. COM allows a user to control where barriers should be placed by the following call.

C: **COM_set_profiling_barrier**(int fHandle, MPI_Comm comm)

Fortran: **SUBROUTINE COM_SET_PROFILING_BARRIER**(FHANDLE, COMM)
INTEGER, INTENT(IN) :: FHANDLE, COMM

This routine will enable COM to call MPI_Barrier on the given communicator before and after the given function for the processes of the given communicator.

The profiling results can be printed by calling

C: **COM_print_profile**(const char *fname, const char *header)

Fortran: **SUBROUTINE COM_PRINT_PROFILE**(FNAME, HEADER)
CHARACTER(*), INTENT(IN) :: FNAME, HEADER

This routine will append the header and the timing results to the file with name fname. If fname is NULL or the empty string, the standard output will be used instead. A typical timing result looks as follows.

```
                   Function      #calls    Time(tree)    Time(self)
-----------------------------------------------------------------------
     Rocflu.update_solution         100       43.1856       42.8221
    Rocfrac.update_solution         100       30.4038       30.3861
 RFC.least_squares_transfer         400      0.957599      0.957599
......
-----------------------------------------------------------------------
      Total(top level calls)                                 74.806
```

In the output, the Time(tree) indicates the sum of the elapsed wall-clock time during the execution of a function since the last call to **COM_init_profiling**, and Time(self) subtracts the elapsed time of the calls made with the function.

C.6.5 Calling System Calls in Fortran

To allow Fortran to execute a shell command using the system call interface of C, COM provides the following **COM_CALL_SYSTEM** function.

Fortran: **FUNCTION COM_CALL_SYSTEM**(COMMAND)
CHARACTER(*), INTENT(IN) :: COMMAND

It will execute the command and return the return status of the command after the command has been completed; if the command fails to execute due to fork failure, then -1 will be returned.

C.6.6 Calling AtExit and Exit Functions In Fortran

A Fortran code can also call the atexit and exit functions of the C standard through COM.

Fortran: **FUNCTION COM_CALL_ATEXIT**(FUNC)
EXTERNAL FUNC

Fortran: **FUNCTION COM_CALL_EXIT**(IERR)
INTEGER, INTENT(IN) :: IERR

COM_CALL_ATEXIT registers a subroutine to be executed when the program terminates normally.
COM_CALL_EXIT causes the program to end and supplies a status code to the calling environment.

C.7 Advanced Window Management

C.7.1 Memory Management

Sometimes, it is more convenient to let COM allocate arrays instead of registering user-allocated arrays. This approach avoids having to duplicate the data structures of windows and panes in application codes for multi-block meshes, and is particularly beneficial for implementing complex orchestration modules. COM provides the following subroutines for memory allocation.

C++: **COM_allocate_array**(const std::string aName, int paneID=0, void **addr=NULL, int strd=0, int cap=0)

C: **COM_allocate_array**(const char *aName, int paneID=0, void **addr=NULL, int strd=0, int cap=0)

Fortran: **SUBROUTINE COM_ALLOCATE_ARRAY**(ANAME, PANEID, ADDR, STRIDE, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
<TYPE>, POINTER :: ADDR
OPTIONAL :: PANEID, ADDR, STRIDE, CAP

C++: **COM_resize_array**(const std::string aName, int paneID=0, void **addr=NULL, int stride=-1, int cap=0)

C: **COM_resize_array**(const char *aName, int paneID=0, void **addr=NULL, int stride=-1, int cap=0)

Fortran: **SUBROUTINE COM_RESIZE_ARRAY**(ANAME, PANEID, ADDR, STRIDE, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
<TYPE>, POINTER :: ADDR
OPTIONAL :: PANEID, ADDR, CAP, STRIDE

These functions take arguments similar to **COM_set_array**, except that addr is returned passed out instead of passed into the procedure. They allocate memory for a specific DataItem in a given pane if paneID is nonzero or all panes if paneID is zero (the default value). The differences between **allocate** and **resize** are that the latter allocates memory only if the array was not yet initialized, or was previously allocated by COM but the current capacity is increased or the stride is no longer the same. During resize, values of the old array will be copied automatically to the new array. If strd is -1, which is the default for **COM_resize_array**, the current value registered with COM (or the number of components if not yet registered) will be used; if strd is 0, then the number of components of the DataItem will be used. If cap is 0, then the larger of the current capacity and the number of items will be used. Note that it is an error to resize an inherited or user-allocated (i..e, not allocated by COM) DataItem. For the Fortran interface, only scalar, 1-D and 2-D pointers are allowed. If a scalar pointer is used, the data itself must be a scalar and the argument STRD and CAP must not be present. If a 1-D pointer is given, then the size of the array will be STRD*CAP. If a 2-D pointer is given, then the sizes of the array will be (STRD,CAP) if STRD is no smaller than the number of components of the DataItem (NCOMP), or be (CAP,NCOMP) if STRD is 1.

A user can use the keyword **all** in the form of "window.**all**" for aName to have COM allocate memory for all DataItems (including the mesh) in a window. The capacity must be no smaller than the size specified by **COM_set_size**; if a value smaller than the actual size is passed to **COM_resize_array**, then the actual size will be used instead.

Furthermore, using **COM_append_array**, COM provides a function to append a series of values to the end of an array associated with a pane or window DataItem that has no ghost items.

C++: **COM_append_array**(const std::string aName, int paneID, const void *addr, int strd, int size)

C: **COM_append_array**(const char *aName, int paneID, const void *addr, int strd, int size)

Fortran: **SUBROUTINE COM_APPEND_ARRAY**(ANAME, PANEID, ADDR, STRD, SIZE)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRD, SIZE
<TYPE>, INTENT(IN) :: ADDR

This function is equivalent to calling **COM_resize_array** to increase the capacity of the array if necessary using the stride currently registered with COM, calling **COM_set_size** to increase the number of items by size, and then copying data from user buffer addr with a stride strd. This function is particularly useful for packing a series of values into a big array in COM. Note that after calling **COM_append_array**, the array in COM may have been reallocated if its capacity was increased, in which case the address previously obtained from COM becomes invalid and the user must reobtain the address by calling **COM_get_array**.

Allocated memory should be deallocated by calling **COM_deallocate_array**.

C++: **COM_deallocate_array**(const std::string aName, int paneID=0)

C: **COM_deallocate_array**(const char *aName, int paneID=0)

Fortran: **SUBROUTINE COM_DEALLOCATE_ARRAY**(ANAME, PANEID)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
OPTIONAL :: PANEID

If the deallocation routine is not called, the memory will be freed automatically when the window is destroyed.

C.7.2 Pointer DataItems

COM provides two special data types, **COM_VOID** and **COM_F90POINTER**. The former means a void pointer in C or C++, and the latter a Fortran 90 pointer. A F90 pointer is different from C/C++ pointers, in that it is a structure containing the descriptor of the data that are referenced, and the exact size of the structure is compiler dependent and may vary with the types that it references. These two data types are particularly useful in conjunction with **COM_allocate_array** to store pointers to some objects, which allows a module to eliminate global variables completely, so that they can take advantage of Charm++.

When COM allocates a F90 pointer, it allocates a piece of memory that is large enough to hold any type of F90 pointers. A F90 application code can copy a pointer to or from COM using **COM_SET_POINTER** and **COM_GET_POINTER**, respectively.

Fortran: **SUBROUTINE COM_SET_POINTER**(ATTR, PTR, ASSO)
CHARACTER(*), INTENT(IN) :: ATTR
<TYPE>, POINTER :: PTR
EXTERNAL ASSO

Fortran: **SUBROUTINE COM_GET_POINTER**(ATTR, PTR, ASSO)
CHARACTER(*), INTENT(IN) :: ATTR
<TYPE>, POINTER :: PTR
EXTERNAL ASSO

These functions are particularly useful for registering the context variable of member functions, similar to registering COM_Object associated with the C++ member functions. For that reason, COM also provides two F90 interface functions, **COM_set_object** and **COM_get_object**, which are essentially aliases of **COM_set_pointer** and **COM_get_pointer**. The argument ASSO is a user-defined subroutine which looks like follows.

```
SUBROUTINE ASSOCIATE_POINTER( attr, ptr)
   <TYPE>, POINTER  :: attr
   <TYPE>, POINTER  :: ptr

   ptr => attr
END SUBROUTINE ASSOCIATE_POINTER
```

Because the arguments of **COM_set_pointer** and **COM_get_pointer** are pointers whose types are unknown to COM, the user must explicitly define the prototypes of these functions in the application codes using the specific data types.

C.7.3 Inheritance

Inheritance is a key concept of object-oriented programming. In current *IMPACT* release, inheritance is very useful under a few situations. First, the orchestration module (*SIM*) sometimes needs to create intermediate data associated with a window owned by another module. Inheritance allows *SIM* to extend the window by adding additional DataItems, or altering the definitions of some of the DataItems. Second, a module (e.g., *Rocburn* in *Rocstar Multiphysics*) may need to operate on a subset of the mesh of another module (e.g., *Rocflo* or *Rocflu*). COM facilitates such special needs by allowing a window to inherit (a subset of) another window without incurring the memory overhead of data duplication. Furthermore, *SIM* sometimes needs to split user-defined windows into separate windows based on boundary-condition types, so that they can be handled differently (such as written into separate files for visualization).

COM supports two types of inheritance: using and cloning. For the former, COM does not duplicate the dataset; for the latter, COM does. For each type, it allows inheriting the mesh from a parent window to a child window in two modes. First, the mesh can be inherited as a whole. Second, only a subset of panes that satisfy a certain criterion are inherited. The following two subroutines support these two modes of use-inheritance, respectively.

C++:	**COM_use_dataitem**(const std::string wName_to, const std::string wName_from, int with_ghost=1, const char *aName=NULL, int val=0)
C:	**COM_use_dataitem**(const char *wName_to, const char *wName_from, int with_ghost=1, const char *aName=NULL, int val=0)
Fortran:	**SUBROUTINE COM_USE_DATAITEM**(WNAME_TO, WNAME_FROM, WITH_GHOST,ANAME, VAL) CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME INTEGER, INTENT(IN) :: VAL, WITH_GHOST

OPTIONAL WITH_GHOST, ANAME, VAL

C++: **COM_clone_dataitem**(const std::string wName_to, const std::string wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

C: **COM_clone_dataitem**(const char *wName_to, const char *wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

Fortran: **SUBROUTINE COM_CLONE_DATAITEM**(WNAME_TO, WNAME_FROM, WITH_GHOST,ANAME, VAL)
CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME
INTEGER, INTENT(IN) :: VAL, WITH_GHOST
OPTIONAL WITH_GHOST, ANAME, VAL

In the arguments, the wName are window names and the aName are DataItem names. The argument with_ghost indicates whether the ghost nodes and elements should be inherited. The next argument is a panel DataItem of integer type, and only the panes whose corresponding values of the DataItem equal to the argument val will be inherited. In practice, aName is most likely to correspond to a boundary-condition type for panes, and val correspond to a boundary condition ID. Note that if aName is empty (i.e., either a NULL pointer or an empty string) and val is nonzero, then condition is considered to be "paneID==val", so that only the pane whose ID is equal to val is inherited.

If a child window needs to contain panes of more than one boundary-condition types, then a user can call **COM_use_dataitem** multiple times with different boundary condition ID. Note that in both routines, the child window does not duplicate memory space for the mesh but inherit the memory addresses of the parent window. If a pane in the parent window does not exist in the target window, a new pane is inserted into the derived window if the DataItem being inherited contains the element connectivities (i.e,. "conn", "mesh", or "all"), or the pane is ignored for other types of elements. If the DataItem being inherited already exists in the child window, then the data type and layout of the new DataItem must be the same as the existing one, and COM will overwrite other information of the existing DataItem with the new DataItem. Note that one must not delete or redefine an DataItem that is being used by another window.

A related function of inheritance is **COM_copy_dataitem**, which copies data from one DataItem onto another.

C++: **COM_copy_dataitem**(const std::string wName_to, const std::string wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

C++: **COM_copy_dataitem**(const int wName_to, const int wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

C: **COM_copy_dataitem**(const char *wName_to, const char *wName_from, int with_ghost=1, const char *aName=NULL, int val=0)

Fortran: **SUBROUTINE COM_COPY_DATAITEM**(WNAME_TO, WNAME_FROM, WITH_GHOST,ANAME, VAL)
CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME
INTEGER, INTENT(IN) :: VAL, WITH_GHOST
OPTIONAL WITH_GHOST, ANAME, VAL

C.7.4 Deletion of Entities

When a window is not needed anymore, it should be destroyed by calling **COM_delete_window**, which takes the window name as its only argument.

C: **COM_delete_window**(const std::string wName)

C: **COM_delete_window**(const char *wName)

Fortran: **SUBROUTINE COM_DELETE_WINDOW**(WNAME)
CHARACTER(*), INTENT(IN) :: WNAME

This subroutine allows COM to clean up its internal data created for a window. It also automatically deallocates all the datasets allocated using **COM_allocate_array** or **COM_resize_array** but not yet deallocated.

Furthermore, one can delete a single pane from a window by calling **COM_delete_pane**, which takes the window name and a pane ID as its arguments.

C++: **COM_delete_pane**(const std::string wName, int pandID)

C: **COM_delete_pane**(const char *wName, int pandID)

Fortran: **SUBROUTINE COM_DELETE_PANE**(WNAME, PANEID)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(IN) :: PANEID

One can also delete an existing DataItem (except for predefined DataItems) by calling **COM_delete_dataitem**.

C++: **COM_delete_dataitem**(const std::string aName)

C: **COM_delete_dataitem**(const char *aName)

Fortran: **SUBROUTINE COM_DELETE_DATAITEM**(ANAME)
CHARACTER(*), INTENT(IN) :: ANAME

The only keyword that can be used with **COM_delete_dataitem** is "data", which will removed all user-defined DataItems and leave only the mesh. Note that after deleting some panes or DataItems, one must call **COM_window_init_done** on all processes collectively before using the window. In addition, deleting a window, pane, or DataItem may invalidate DataItem and function handles and the structure of inheritance, so they should be used with extreme care.

C.8 Information Retrieval

C.8.1 Window and panes

Typically, data registered by application modules need to be accessed only by service modules through the C++ interface described in the Developers Guide. However, some application modules (e.g., *Rocburn*) need to obtain the information about a window created by another module (e.g., *Rocflo*/*Rocflu*). *IMPACT* provides functions to support information retrieval, under the assumption that the caller knows about the DataItem names and base data types of the DataItems.

C: **COM_get_communicator**(const char *wName, MPI_Comm *comm)

Fortran: **SUBROUTINE COM_GET_COMMUNICATOR**(WNAME, COMM)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(OUT) :: COMM

This subroutine obtains the MPI communicator of a window.

The following subroutine obtains the IDs of the panes in a window local to a process:

C/C++: **COM_get_panes**(const char *wName, int *np, int **pane_ids=NULL, int rank=myrank)

C++: **COM_get_panes**(const char *wName, vector<int> &pane_ids, int rank=myrank)

Fortran:
```
SUBROUTINE COM_GET_PANES(WNAME, NP, PANE_IDS, RANK)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(OUT) :: NP
OPTIONAL, INTEGER, POINTER :: PANE_IDS(:)
OPTIONAL, INTEGER, INTENT(IN) :: RANK
```

It sets the number of panes to np and loads an array of IDs into pane_ids, whose memory is allocated by COM and should be deallocated by calling **COM_free_buffer** (except for the vector interface). The rank is in the scope of the MPI communicator of the window. If the rank is not present or is -2, then the default value is that of the current process. If the rank is -1, then the function will load the panes on all the processes within the communicator. Note that this function can only be called **after** calling **COM_window_init_done**.

C/C++: **COM_get_dataitems**(const char *wName, int *na, char **names)

C++: **COM_get_dataitems**(const char *wName, int *na, string &names)

Fortran:
```
SUBROUTINE COM_GET_DATAITEMS( WNAME, NA, NAMES)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(OUT) :: NA
CHARACTER, POINTER :: NAMES(:)
```

It sets na to be the number of DataItems in the window and allocates a space-delimited string names to store the names of the DataItems. Except for the string interface, names must be deallocated by calling **COM_free_buffer** (see below) after use.

C/C++: **COM_get_connectivities**(const char *wName, const int *pid, int *nc, char **names)

C++: **COM_get_connectivities**(const char *wName, const int *pid, int *nc, string &names)

Fortran:
```
SUBROUTINE COM_GET_CONNECTIVITIES( WNAME, PID, NC, NAMES)
CHARACTER(*), INTENT(IN) :: WNAME
INTEGER, INTENT(IN) :: PID
INTEGER, INTENT(OUT) :: NC
CHARACTER, POINTER :: NAMES(:)
```

It sets nc to be the number of connectivity tables in a pane and allocates a space-delimited string names to store the names of the connectivity tables. Again, names must be deallocated by calling **COM_free_buffer** (except for the string interface) after use.

C: **COM_free_buffer**(char (or int) **buf)

Fortran:
```
SUBROUTINE COM_FREE_BUFFER( BUF)
CHARACTER (or INTEGER), POINTER :: BUF(:)
```

C.8.2 DataItem and Connectivity

One can obtain the information of an DataItem by calling **COM_get_dataitem**, whose arguments correspond to those of **COM_new_dataitem**.

C: **COM_get_dataitem**(const char *aName, char *loc, COM_Type *type, int *ncomp, char *unit, int n)

C++: **COM_get_dataitem**(const char *aName, char *loc, COM_Type *type, int *ncomp, string *unit)

Fortran: **SUBROUTINE COM_GET_DATAITEM**(ANAME, LOC, TYPE, NCOMP, UNIT)
```
CHARACTER(*), INTENT(IN) :: ANAME
CHARACTER*1, INTENT(OUT) :: LOC
INTEGER, INTENT(OUT) :: TYPE, NCOMP
CHARACTER(*) :: UNIT
```

One can also use **COM_get_dataitem** on a connectivity, which will set ncomp to the number of nodes per element for that particular type of element.

One can also check the status of a window, a pane, an DataItem, or a connectivity table by calling the function **COM_get_status**.

C: int **COM_get_status**(const char *aName, int paneID)

Fortran: **FUNCTION COM_GET_STATUS**(ANAME, PANEID)
```
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
INTEGER :: COM_GET_STATUS
```

If aName is a window name (i.e., containing no '.') and paneID is 0, then it checks whether the window exists, and returns 0 if so and -1 otherwise. If aName is a window name and paneID is nonzero, then it checks whether the given pane exist in the window, and returns 0 if so and -1 otherwise. If aName is in the form of "window.DataItem", then it checks the status of the given DataItem, and returns one of the following values:

- -1: does not exist;
- 0: exists but not initialized;
- 1: set by the user using set_array or set_object;
- 2: set by the user using set_array_const;
- 3: use from another DataItem;
- 4: allocated using resize_array, allocate_array, or cloned from another DataItem.

If an DataItem uses another, one can get the full name (window.DataItem) of its parent DataItem using the following interface:

C/C++: **COM_get_parent**(const char *waName, int paneid, char **parent)

C++: **COM_get_parent**(const char *waName, int paneid, string &parent)

Fortran: **SUBROUTINE COM_GET_PARENT**(WANAME, PANEID, PARENT)
CHARACTER(*), INTENT(IN) :: WANAME
INTEGER, INTENT(IN) :: PANEID
CHARACTER, POINTER :: PARENT(:)

The storage for the parent name will be allocated by COM and must be freed using **COM_free_buffer** after usage, except for the C++ interface.

C.8.3 Sizes

The following function can be used to obtain the sizes of an DataItem. Note that the size corresponds to the total size (including ghost items).

C++: **COM_get_size**(const std::string aName, int pane_id, int *size, int *ng=NULL)

C: **COM_get_size**(const char *aName, int pane_id, int *size, int *ng=NULL)

Fortran: **SUBROUTINE COM_get_size**(ANAME, PANE_ID, SIZE, NG)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
INTEGER, INTENT(OUT) :: SIZE, NG
OPTIONAL :: NG

Note that for structured meshes, its dimensions should be obtained using **COM_get_array_const** instead of **COM_get_size**.

C.8.4 Arrays

One can obtain an array by either obtaining a pointer to the array, or copying the data into a user provided buffer. The first mode is provided by **COM_get_array** and **COM_get_array_const**, which can be used to obtain a pointer to an array registered or allocated in *COM*.

C: **COM_get_array**(const char *aName, int paneID, void **addr,
int *strd=NULL, int *cap=NULL)

Fortran: **SUBROUTINE COM_GET_ARRAY**(ANAME, PANEID, ADDR,
STRD, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
<TYPE>, POINTER :: ADDR
INTEGER, INTENT(OUT) :: STRD, CAP
OPTIONAL :: STRD, CAP

C: **COM_get_array_const**(...)

Fortran: **SUBROUTINE COM_GET_ARRAY_CONST**(...)

Note that if the DataItem-name and the pane-ID do not identify a unique array, then a NULL pointer will be returned. Furthermore, if an array was registered with **COM_set_array_const**, then it can be retrieved only by **COM_get_array_const**. For the Fortran interface, only scalar, 1-D and 2-D pointers are allowed. If a scalar pointer is used, the data itself must be a scalar and the argument STRD and CAP must not be present. If a 1-D pointer is given, then the size of the array will be STRD*CAP. If a 2-D pointer is given, then the sizes of the array will be (STRD,CAP) if STRD is no smaller than the number of components of the DataItem (NCOMP), or be (CAP,NCOMP) if STRD is 1.

The second mode is provided by **COM_copy_array**.

C: **COM_copy_array**(const char *aName, int paneID, void *val, int v_strd=0, int v_size=0, int offset=0)

Fortran: **SUBROUTINE COM_GET_ARRAY**(ANAME, PANEID, VAL, V_STRD, V_SIZE, OFFSET)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, V_STRD, V_SIZE, OFFSET
<TYPE>, POINTER :: VAL
OPTIONAL :: V_STRD, V_SIZE, OFFSET

It copies up to v_size items of the DataItem starting from the offset-th item of the DataItem into the given buffer with stride v_strd. If v_strd=0 (the default value), then the number of components will be used as the stride. If v_size=0 (the default value), then the number of items will be used as the size. The default value of offset is 0. Note that a runtime error occurs if offset is negative or offset+v_size is larger than the actual capacity of the DataItem.

C.8.5 Bounds

The lower and upper bounds of a specific DataItem can be obtained by calling **COM_get_bounds**.

C: **COM_get_bounds**(const char *aName, int pane_id, void *lbnd, void *ubnd, int is_soft=0)

Fortran: **SUBROUTINE COM_get_bounds**(ANAME, PANE_ID, LBND, UBND, IS_HARD)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
<TYPE>, INTENT(OUT) :: LBND, UBND
INTEGER, INTENT(IN), OPTIONAL :: IS_SOFT

Furthermore, COM also provides functions to check the DataItems against pre-set bounds.

C: int **COM_check_bounds**(const char *aName, int pane_id, int nprint=0)

Fortran: INTEGER **FUNCTION COM_check_bounds**(ANAME, PANE_ID, NPRINT)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANE_ID
INTEGER, INTENT(IN), OPTIONAL :: NPRINT

If pane_id is 0, then the bounds will be checked on all panes; otherwise, they will be checked only on the pane with the given pane ID. If there are only soft-bound violations or no violations, then the function returns the number of soft-bound violations. If the verbose level of COM is 0, then no information will be printed. If the verbose level is nonzero and nprint is 0, then a summary of soft-bound violations will be printed. If nprint is greater than 0, then the first few (where the number to be printed is nprint) soft-bound violations for each DataItem in each individual pane will be printed. A violation of hard bounds will terminate the execution of the code, and a summary of hard-bound violations will be printed upon termination, along with information about any soft-bound violations if the verbose level is nonzero.

C.9 Sample Codes

A few sample application codes of COM are provided in the test subdirectories of a few service modules. In particular, sample codes are available in Simpal/test and SurfMap/test.

D SIM User's Guide

Simulation Integration Manager
Version 0.1.0 IllinoisRocstar LLC October 25, 2016

License

The software package sources and executables referenced within are developed and supported by Illinois Rocstar LLC, located in Champaign, Illinois.The software and this document are licensed by the University of Illinois/NCSA Open Source License (see `opensource.org/licenses/NCSA`). The license is included below.

For more information regarding the software, its documentation, or support agreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

D.1 Overview

The *Simulation Integration Manager* (*SIM*) is an orchestrator implementation in the *IMPACT* suite. It is the front-end of the code that directly interacts with end-developers and end-users of coupled simulations. *SIM* is a higher-level infrastructure, built on top of the *COM* integration framework, but it is possible to use it independently of *COM*. The overall objectives of *SIM* are to provide **high-level abstractions** of the operations in multiphysics coupling, facilitate **flexible** and **easy construction** of complex coupling algorithms from predefined building blocks, and deliver **readability** of the orchestration module.

D.2 Capabilities

The next-generation of *SIM* will support the following capabilities, and allow potential extensions:

- different types of applications
  - unsteady-state calculations (with time-marching schemes)
  - steady-state calculations
- different types of coupling (from simplest to most complex)
  - stand-alone fluid or solid
    - with or without combustion and flexible control of initial ignition
    - with or without surface regression and time zooming
    - thermal boundary conditions, chemical reactions, etc.
    - with or without propagation constraints
  - fully-coupled fluid-solid interaction
    - with or without predictor-corrector iterations
    - with or without subcycling
    - flexible execution order (e.g., alternating orders and concurrent executions)
  - fully-coupled fluid-solid-combustion interaction
    - all suboptions listed above
- numerical and geometrical capabilities
  - mesh modification
  - surface propagation
  - sliding interfaces
- infrastructure runtime support
  - flexible, user-friendly control mechanism in choosing different options
  - concurrent execution of independent tasks, possibly on different sets of processors
  - exact restart in all coupling modes
  - asynchronous I/O using different file formats, and separation of visualization and restart files

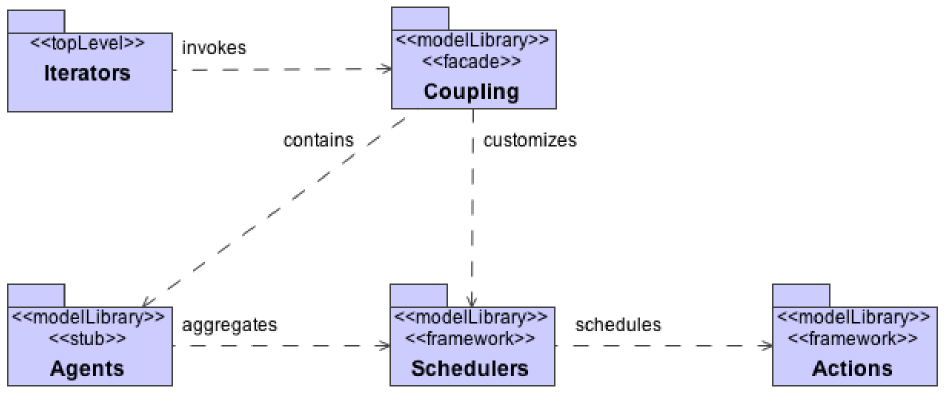


Figure D.56: Overview of system architecture of *SIM*.

D.2.1 Design Features

To facilitate the diverse needs of different applications and coupling schemes, we design a new software infrastructure for *SIM*, which has the following features:

- Tiered architecture: the layers are *top-level iterations*, *coupling schemes* (with or without predictor-corrector iterations), *agents* for physics modules, and manipulation of *jump conditions*;
- Action-centric specification: coupling schemes are described as *actions*, with well-defined input and output to specify the *data flow*;
- Automatic scheduling: based on the data flow among actions, the *control flow* is derived automatically to determine the scheduling of actions by the runtime system, which allows potential concurrent execution of actions;
- Visual aid: *SIM* will provide *visualization of data flow* of actions, to help users comprehend and debug coupling algorithms.

D.3 System Architecture

SIM contains five types of key components, as depicted in Figure Figure D.56: top-level iterations, coupling schemes (with or without predictor-corrector iterations), agents for physics modules, schedulers, and actions. We will explain these components in the following subsections.

D.3.1 Top-level Iterations

Time-marching Schemes For unsteady-state calculations, the time-marching scheme is a simple driver code, which can be used with different types of coupling schemes. The pseudo-code in Procedure 1 outlines the time-marching procedure, whose core is the coupling schemes.

Steady-state Iterations Steady-state calculations are sometimes used in fluid-solid interactions and for stand-alone simulations, and hence are desirable features for *SIM*. The pseudo-code in Procedure 2 outlines the top-level iterations of a steady-state calculation, which is very similar to the time-marching scheme.

Procedure 1 Top-level time-marching scheme.

```
construct coupling object
invoke input of coupling object
invoke initialization of coupling object
while not yet reached designated time do
  invoke time integration of coupling scheme by passing in current time and obtaining new time
  if reached time for restart dumps then
    invoke restart output (as well as visualization data) of coupling scheme
  else if reached time for visualization dump then
    invoke visualization output of coupling scheme
  end if
end while
invoke finalization of coupling object
```

Procedure 2 Top-level iteration for steady-state calculations.

```
construct coupling object
invoke input of coupling object
invoke initialization of coupling object
while not yet converged do
  invoke solver of coupling scheme
  if reached stage for restart dumps then
    invoke restart output (as well as visualization data) of coupling scheme
  else if reached stage for visualization dump then
    invoke visualization output of coupling scheme
  end if
end while
invoke finalization of coupling object
```

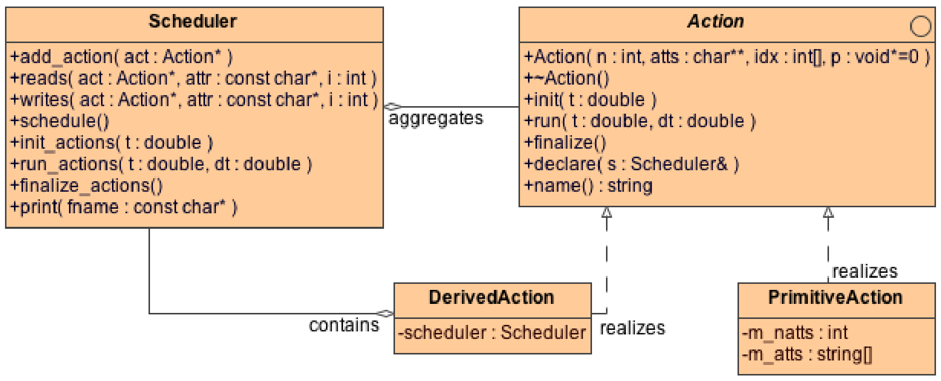

Scheduler
+add_action(act : Action*)
+reads(act : Action*, attr : const char*, i : int)
+writes(act : Action*, attr : const char*, i : int)
+schedule()
+init_actions(t : double)
+run_actions(t : double, dt : double)
+finalize_actions()
+print(fname : const char*)
aggregates
Action
+Action(n : int, atts : char**, idx : int[], p : void*=0)
+~Action()
+init(t : double)
+run(t : double, dt : double)
+finalize()
+declare(s : Scheduler&)
+name() : string
contains
DerivedAction
-scheduler : Scheduler
realizes
realizes
PrimitiveAction
-m_natts : int
-m_atts : string[]

Figure D.57: Overview of actions and schedulers.

D.3.2 Actions and Schedulers

As depicted in the class diagram in Figure D.57, actions and schedulers are interdependent on each other. An *action* is a functional object. There are two types of actions: *primitive* actions, which perform some basic calculations, and *derived* actions, which are composed of collections of subactions. An action has a constructor, and three core procedures: init(), run(), and finalize(), for performing initialization, execution, and finalization, respectively. The constructor typically takes a number of DataItem names and their corresponding *time indices* as arguments. The time indices are important to denote whether the DataItem is from a previous iteration of the top-level loop. Optionally, the constructor may also take a void pointer as its final "wildcard" argument, which can point to a specific structure to encapsulate the additional input and output of the action.

The initialization procedure allocates the intermediate data needed to perform the action, and finalization deallocates these intermediate data. In general, when the constructor is invoked, the DataItem names listed in the arguments may not have been associated with actual DataItems; when initialization is invoked, they will have been associated. Depending on whether the action is primitive or derived, its execution procedure performs some basic calculations, or invokes the execution of subactions. In addition, an action has two procedures for interacting with schedulers: declare(), which registers the read and write operations performed on the DataItems passed into the constructor, and name(), which provides a descriptive name for the action for visualization.

A *scheduler* is a container of actions, and is responsible for determining the orders of initialization, execution, and finalization of its actions. It is also part of a derived action for scheduling the subactions. A scheduler provides a procedure add_action() to its user for registering actions. The scheduler invokes the declare() member function of an action when it is being added, and declare() registers data access with the scheduler by calling the reads() and writes() member functions of the scheduler. After all the actions have been registered with a scheduler, it then constructs a directed acyclic graph (DAG) for the actions, in which each edge is identified by a DataItem name and its corresponding time index. Typically, the pair of DataItem and time indices should identify one unique provider, but a DataItem may be used by multiple actions.

If a DataItem has multiple providers, the scheduler will try to identify the actual provider using other dependencies, and execution will abort if this identification process fails. The schedule() member function determines the orders of initialization, execution, and finalization of its actions, and recursively invokes the scheduling of the derived actions. In a parallel computation, the schedulers on all processes must return the same orders. The member functions init_actions(), run_actions(), and finalize_actions() invoke the

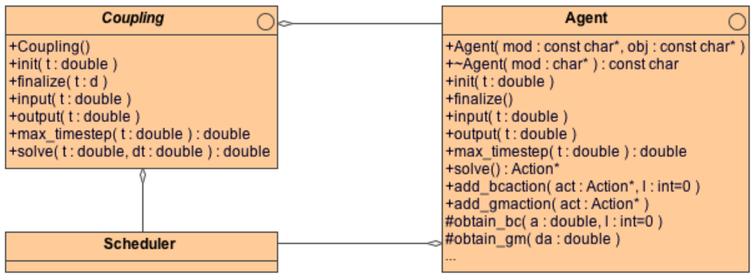

Coupling
+Coupling()
+init(t : double)
+finalize(t : d)
+input(t : double)
+output(t : double)
+max_timestep(t : double) : double
+solve(t : double, dt : double) : double
Agent
+Agent(mod : const char*, obj : const char*)
+~Agent(mod : char*) : const char
+init(t : double)
+finalize()
+input(t : double)
+output(t : double)
+max_timestep(t : double) : double
+solve() : Action*
+add_bcaction(act : Action*, l : int=0)
+add_gmaction(act : Action*)
#obtain_bc(a : double, l : int=0)
#obtain_gm(da : double)
...
Scheduler

Figure D.58: Overview of agents and coupling schemes.

corresponding procedures of the actions. Furthermore, the scheduler provides a print() function, for printing out the DAG and its scheduling for visualization.

In general, an action is present in at least one scheduler, and typically is in only one scheduler. The user (such as a derived action) of a scheduler should create actions by calling the C++ new operator, but the owner schedulers of the actions will be responsible for deleting the actions when they are no longer in use. If an action is used by multiple schedulers, the user must ensure the action is *reentrant*, in the sense that its initialization and finalization procedures do not allocate and deallocate intermediate data multiple times, and it is safe for multiple instances of the action to run concurrently.

D.3.3 Agents and Coupling Schemes

An *agent* serves a physics module. The most basic task of an agent is to load a physics module ComponentInterface (CI) into *COM* in the constructor. A more critical set of tasks of an agent is to manage the persistent buffer data on behalf of the physics module. These data are typically defined on the mesh of the physics module, may be shared by multiple actions, and need to be saved for visualization or restart. The constructor of an agent defines the buffer data, and its initialization procedure is responsible for allocating these buffer data, after invoking the initialization of the physics module.

The finalization of an agent deallocates these buffers and invokes the finalization of the module. Since these buffer data may need to be read and written along with the data of the physics module, the agent is responsible for providing the input and output functions, to be called by the coupling schemes.

In addition to providing data services for a physics module, the agent also interacts with scheduler on its behalf, and and feeds boundary conditions to the physics module. An agent usually provides one action to coupling schemes, for invoking the solver of the module. It also provides a function for obtaining the maximum time step constrained by the module.

For supply boundary conditions, it provides two callback procedures to the physics module, one for obtaining boundary conditions, and the other for obtaining grid motion. These procedures may be called by the physics module during its iterations of calculation (such as subcycling or Runge-Kutta iterations) without returning the control back to the coupling scheme. These callback procedures are implemented as derived actions in the agent, whose subactions are defined by the coupling schemes and registered by calling add_bcaction() and add_gmaction(), respectively. The scheduling of these callback procedures should be performed when the solve action is being scheduled by the main scheduler.

A *coupling scheme* is composed of a number of agents and a scheduler. Its constructor invokes the constructors of the agents and the scheduler to define the coupling algorithm, and then determines the orders that must be followed for invoking initialization, execution, and finalization. The initialization (init) and

Procedure 3 Coupling with predictor-corrector iterations.

```
for $i = 1$ to maxiter do
    invoke time integration of base coupling scheme
    check convergence
    if converged then
        store current solution
        break from loop
    else
        restore previous solution
    end if
end for

if $i \geq$maxiter and not converged then
    throw exception
end if
```

finalization (finalize) procedures of the coupling scheme initialize and finalize the agents and actions in the scheduled order, respectively. Its execution (solve) procedure takes the current time and the pre-determined time step, runs the actions, and then returns the new time. The coupling scheme also provides interfaces to the top-level iterators for input, output, and obtaining the maximum time step, which invoke their counterparts of the agents.

D.3.4 Predictor-corrector Iterations

Coupling with predictor-corrector (PC) iterations generalizes the basic coupling scheme described above. Besides the standard actions, coupling with PC iterations requires an inner loop to repeat the base coupling algorithm, until the interface conditions have converged, or a maximum number of iterations have been reached. Its execution requires some additional services, such as checking convergence and storing and restoring data. Procedure 3 describe the sequence of such a PC iteration, in which different base coupling schemes can be plugged.

D.4 Predefined Actions

To support implementations of coupling schemes, we define four types of actions: the execution of a physics module, interpolation of boundary conditions, manipulation of jump conditions, and operations for supporting PC iterations.

.

D.4.1 Solve

This action is physics dependent, and is defined in an agent. It takes the input and output of the physics module as the DataItem lists of the arguments, and takes a pointer to the parent agent as its wildcard argument.

D.4.2 Interpolate

This action in general handles interpolation or extrapolation in time for one DataItem. Except for time zooming, it is the sole building block for update_bc() and update_gm() functions of an Agent instance. There can be different types of interpolation schemes. In general, its constructor takes following arguments:

- The interpolation scheme, passed in as the wildcard argument, which can be one of the EXTRAP_LINEAR, INTERP_LINEAR, EXTRAP_CENTRAL, INTERP_CENTRAL, INTERP_CONST;
- Three DataItem names: current data, interpolated data, and old data. The old DataItem is optional and needed only for INTERP_CENTRAL and INTERP_LINEAR. For INTERP_LINEAR and EXTRAP_LINEAR, the action may construct a gradient DataItem internally.

D.4.3 Jump Conditions

These are the high-level abstractions of the jump conditions in each type of coupling, such as motion transfer, load transfer, heat transfer, mass transfer, momentum transfer, etc..

D.4.4 Actions for PCCoupling

PCCoupling requires some special services, including checking convergence and storing and restoring data. These special services are defined as protected classes within PCCoupling. These actions are manually scheduled and are called directly by the Procedure 3, instead of by a scheduler.

PCService PCService serves as the base class for the implementation of the other services in this subsection. Its constructor takes a list of DataItem names. The given DataItems may or may not belong to the same user window. The init() operation groups the DataItems received by the constructor based on their owner windows, and creates one buffer window for each owner window by using the connectivity tables (to replicate the panes) and then cloning the listed DataItems in it. The buffer data created will not be saved for restart. The action provides a protected utility function copy(dir) to help the implementation of subclasses. The function copies data DataItems from user windows to buffer windows or vice versa, depending on the argument.

CheckConvergence The class CheckConvergence is a subclass of PCService, in charge of managing memory space for backing up user-specified interface quantities, to check whether PC iterations have converged. Besides inheriting the init() operation from PCService, it has two public interface functions:

- set_tolerance(attr, tol, norm): sets the tolerance of a given DataItem for a specific norm.
- check(ipc) takes the current iteration index of the PC-iterations as input, and compares the norms of the DataItems against the preset tolerances. If all tolerances are met, it returns true; otherwise, it copies the current solution for later comparison and then returns false.

Backup The class Backup is also a subclass of PCService, in charge of managing memory space for backing up user-specified data DataItems after PC iterations have converged, and recovering data if not yet converged. Besides inheriting the init() operation from PCService, it has two public interface functions:

- store(): copies the data from CI windows to buffer windows;
- restore(): copies data from buffer windows back to CI windows.

D.5 Schedulers

D.5.1 Sequential

The simplest type of scheduler is to schedule the operations of the actions based on the order of their registration. It does not require constructing DAGs. It is particularly convenient for implementing simple derived actions, and should suffice for many simpler coupling schemes.

D.5.2 Concurrent

More sophisticated schedulers can allow execution of multiple independent actions concurrently and on different sets of processors. The concurrent scheme requires construction of DAGs, and may make use of sophisticated scheduling algorithms for optimal performance.

D.5.3 Interprocess

The most sophisticated schedulers can allow execution of independent actions to run on different processes, and potentially allow migration of actions.

D.6 Predefined Agents

Agents represent each of the physics modules which are typically written in a separate library in Fortran 90. Agents provide support to initialize the physics modules and drive their simulations. The base Agent class implements common features required by all physics modules including file I/O (using *SimIO*), and initialization of *COM* function handles.

In general, a given physics code will present a CI with a set of windows. The functions and data presented in application-specific CI are, in general, arbitrary. This situation necessitates multiple Agents. An application-specific Agent derives from a domain-specific Agent and uses the application-specific CI to present the desired interface to the *SIM* coupling constructs. Domain-specific agents are discussed below.

D.6.1 Fluid agent

The class FluidAgent is designed to present a computational fluid dynamics (CFD) interface that can be used in the advanced *SIM* coupling constructs. An arbitrary CFD application publishes its native data and functions through the CI window, and an application-specific Agent derives from FluidAgent and massages the application's CI so that it conforms. Together, the application-specific Agent and the FluidAgent create all necessary *COM* windows for boundary condition data that are used in coupled simulations. It defines the subroutine to write restart data files using *SimIO/SimOUT*. It also creates window for convergence check.

D.6.2 Solid agent

The class SolidAgent is designed to present a computational structural mechanics (CSM) interface that can be used in the advanced *SIM*. It creates all necessary *COM* windows for boundary condition data that are used in coupled simulations. It defines the subroutine to write restart data files using Rocout. It also compute integrals for conservation check.

D.6.3 Burn agent

The class BurnAgent is designed to work with transient thermal solvers, ignition, and burn-rate providers. It creates all necessary *COM* windows for boundary condition data that are used in coupled simulations. It defines the subroutine to write restart data files using *SimIO*.

D.7 Predefined Coupling Schemes

The class Couple implements the basic functions of a coupling scheme, which is composed of a number of agents and a scheduler. The definition of a particular coupling scheme is defined in the constructors of a derived Couple class. The physics details of each coupling scheme are defined in an (upcoming) manual - Numerical Coupling Interface in *SIM*.

D.7.1 Fluid-alone

Fluid alone simulation includes fluid only with or without combustion.

D.7.2 Solid-alone

Solid alone without combustion. It is implemented in derived class SolidAlone.

D.7.3 Fluid-solid interaction

Fluid solid interaction includes both fluid and solid modules, but without combustion. The following derived classes are implemented: SolidFluidSPC for solid, fluid, no combustion, simple staggered scheme with P-C; SolidFluidBurnSPC for solid, fluid, combustion and simple staggered scheme with P-C; FluidSolidISS for fluid, solid and no combustion using improved staggered scheme.

D.7.4 Fluid-solid-combustion interaction

Similar to fluid-solid interaction but with combustion. It is implemented in derived class SolidFluidBurnSPC for simple staggered scheme with P-C, and SolidFluidBurnEnergySPC with simple staggered scheme with burn energy.

E SimIO User's Guide

***SimIO* Users Guide**
Version 0.1.0 IllinoisRocstar LLC October 25, 2016

License

The software package sources and executables referenced within are developed and supported by Illinois Rocstar LLC, located in Champaign, Illinois.The software and this document are licensed by the University of Illinois/NCSA Open Source License (see `opensource.org/licenses/NCSA`). The license is included below.

For more information regarding the software, its documentation, or support aggreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

E.1 Functionality

SimIN creates a series of *COM* windows by reading in a list of files in either HDF or CGNS format. The files are self-contained in that they contain not only field data but also metadata (such as sizes, data types, locations, and units), which are needed to create *COM* windows. *SimIN* maps the "blocks" in the HDF files or "zones" in CGNS files into panes in *COM* windows. *SimIN* can be called collectively on multiple processes, or by a single process in a sequential program, in which MPI does not need to be initialized.

E.2 API

SimIN provides a simple API while maintaining the necessary flexibility and efficiency. It provides two sets of functions: the first set reads in metadata from the files (and can optionally read in array data as well), and the second set passes the array data from the files (or from *COM*'s memory space if data are already loaded) to users memory space. The API are typically called through COM_call_function (see *COM* Users Guide) and hence all arguments are passed by pointers (references).

E.2.1 Read Window

o read metadata from files, a user can use one of the following two functions to read a single window or a series of windows, respectively.

- void read_window(const char *filename_patterns,
  const char *window_name,
  const MPI_Comm *comm=NULL,
  const RulesPtr *is_local=NULL,
  char *time_level=NULL,
  const char *str_maxlen=NULL);
- void read_windows(const char *filename_patterns,
  const char *window_prefix,
  const char *material_names=NULL,
  const MPI_Comm *comm=NULL,
  const RulesPtr *is_local=NULL,
  char *time_level=NULL,
  const char *str_maxlen=NULL);

The argument filename_patterns specifies a list of zero, one or more patterns (regular expressions) of the files to be read. Multiple patterns should be separated with empty spaces. If there is no file matching the patterns in filename_patterns on all processes (where filename_patterns can be empty or not), then a warning message will be printed, and an empty window (or windows) will be created. If there are matching files on any process, then a window (or windows) will be created by mapping from the matching files. It is guaranteed that each window defines all the DataItems existing in any of the blocks (or zones) of its corresponding material in the files read by the processes within the given MPI communicator. If a pane has no data for a particular DataItem in its corresponding block (or zone), then the array associated with the DataItem will not be allocated in the pane.

The argument window_prefix specifies the window name (in the case of read_window()) or a prefix of the name(s) of the window(s) (for read_windows()) to be created. For read_windows, the argument material_names specifies a list of (space-separated) materials to be read from the files, and these strings are appended to window_prefix to obtain the complete window names. If the material_name is NULL or the empty string, then it is assumed that the files contain only one type of material, and the window name will be window_prefix. Note that the windows created by these functions must be deleted by the user by calling COM_delete_window after usage.

Among the remaining more advanced arguments, comm specifies an MPI communicator. If comm is not present or is NULL, then the default communicator is MPI_COMM_SELF. The arguments is_local is a function pointer of type

```
void (*)(const int &pid, const int &comm_rank, const int &comm_size, int *local),
```

which determines whether a pane of given block ID (HDF) or zone ID (CGNS) will be read by the current process. The first three arguments are input-only and the final argument is output-only. The last two arguments are for setting and obtaining the time level of the dataset. If time_level is a nonempty string, it will be used as an input, and the functions read only those datasets that have the matching time stamp; otherwise, all datasets are assumed to have the same time stamp, and that of the first dataset read from the files will be returned by copying up to *str_maxlen characters (including a null terminator) into time_level, if both time_level and str_maxlen are present and not NULL pointers.

E.2.2 Read by Control File

o allow more flexible user control, *SimIN* can also obtain HDF/CGNS file names and pane IDs from a user-provided control file.

- void read_by_control_file(const char *control_file_name,
  const char *window_name,
  const MPI_Comm *comm=NULL,
  char *time_level=NULL,
  const char *str_maxlen=NULL);

The control file contains a number of control blocks, each of which has up to four fields (a process rank, a list of file names, a list of pane IDs, and optionally, a material name), as described shortly, and each filed in general should be on one line. If present, the material name must be the same in all control blocks. At runtime, a process obtains the file names and pane IDs from the first control block that has a matching process rank. If comm is NULL, the default communicator is MPI_COMM_SELF, and the rank for all processes will be 0; if comm is present and is not MPI_COMM_NULL, then the process rank in the given MPI communicator will be used; if comm is MPI_COMM_NULL, then the rank of the current MPI process is replaced by a wild card, and the panes in all the files listed in all the control blocks will be read. In other words, when comm is MPI_COMM_NULL, all the panes in all the files listed in the control file will be considered local. This wild-card feature is useful for a serial application to read in all the panes, which would have been distributed by the control file onto different processes in a parallel run.

@Proc: arks the beginning of a process block, followed by a process's rank. A process reads in all the blocks that match the current process rank. A wild card '*' (without quotes) can be used after @Procs: to match any process. If the rank of the current process is a wild card (i.e., *comm==MPI_COMM_NULL), then all blocks will match the current process.

@Files: list of zero, one, or more file name patterns separated by empty spaces. A file name can contain the following place holders:

1. %*d*p for process rank, where *d* is an integer indicating the number of digits in the rank. If the number *d* is absent, then the default value is 4. If the current process's rank is a wild card (i.e., *comm==MPI_COMM_NULL), then any *d* digits in a file name will match.
2. %*d*i for pane ID, where *d* is an integer indicating the number of digits in the pane ID. It maps a file with a pane ID *n* onto a process, if *n* will be mapped onto the current process by the pane mapping. The default value of *d* is 4.
3. %*d*b for block ID, where *d* is an integer indicating the number of digits in the block ID. This option can only be used in conjunction with the @Block or @BlockCyclic mapping in the next subsection, which maps a file with a block ID *n* onto a process, if n*base will be mapped onto the current process by the mapping. The default value of *d* is 4.
4. %t for time stamp, which will be replaced by the time_level input argument.

For example, a file name "fluid*_%t_%4p.*" with a time level "00.000000" will be replaced by "fluid*_00.000000_0000.*" on process 0 and by "fluid*_00.000000_0001.*" on process 1. In general, a file name may use at most one of %*d*p, %*d*i or %*d*b. If the listed file names contain no directory path, then the files are assumed to be in the same directory that contains the control file. If a file name contains a relative path, then the path is considered to be relative to the current working directory at runtime.

@Panes: pecifies a list of zero, one, or more pane IDs to be read by the process. For convenience, the user can also specify one of the following mapping rules:

- **@All** (or equivalently a wild card '*' without quotes)

All panes are mapped onto the process.

- **@Cyclic [<offset>]**

A pane is mapped onto a process if

$$\text{mod}(\text{paneID} - \text{offset}, \text{nprocs}) = \text{rank}.$$

- **@BlockCyclic <base> [<offset>]**

A pane is mapped onto a process if

$$\text{mod}((\text{paneID-offset})/\text{base}, \text{nprocs}) = \text{rank}.$$

The default value of offset is 0. For example, for four processes, "@BlockCyclic 100 100" for 14 panes results in the following mapping:

Process 0: 100 500 900 1300
Process 1: 200 600 1000 1400
Process 2: 300 700 1100
Process 3: 400 800 1200

- **@Block <nblocks> <base> [<offset>]**

A pane is mapped onto a process if

$$\begin{cases} (\text{paneID} - \text{offset})/(\text{quot} * \text{base} + \text{base}) = \text{rank}, & \text{if rank} < \text{rem}, \\ (\text{paneID} - \text{offset} - \text{rem})/(\text{quot} * \text{base}) = \text{rank}, & \text{otherwise}, \end{cases}$$

where nblocks=quot*nprocs+rem. The default value of offset is 0. For example, for four processes, "@Block 14 100 100" results in the following mapping:

Process 0: 100 200 300 400
Process 1: 500 600 700 800
Process 2: 900 1000 1100
Process 3: 1200 1300 1400

Note that the @Panes field may be left out if the @Files field is empty. When *comm==MPI_COMM_NULL, then the @Panes field is immaterial.

@Material: To be implemented.] The keyword is followed by a character string to indicate the name of the material to be read. This field is optional and typically need not to be present when there is only one type of material in the files (i.e., when all the data in the files belong to the same window).

Sample Control Files he following is a generic control file specifying each process to read in a rank-dependent file for a given time stamp, with block cyclic mapping for panes.

```
@Proc: *
@Files: fluid*_%t_%4p.hdf
@Panes: @BlockCyclic 100 1
```

The following is a specific control file for two processes.

```
@Proc: 0
@Files: fluid*_00.00_0000.hdf
@Panes: 1 3 5 7 9
@Material: fluid
@Proc: 1
@Files: fluid*_00.00_0001.hdf
@Panes: 2 4 6 8 10
@Material: fluid
```

Read Parameter File o read parameters from a file into a window, the following function should be used:

- void read_parameter_file(const char *file_name,
  const char *window_name,
  const MPI_Comm *comm=NULL);

The function reads parameters from the given file and stores them as window DataItems in the given parameter window. If the window already exists, then only the DataItems that already exist in the window are read from the file. Otherwise a new window is created and all of the parameters are read in. Process 0 of the communicator should read the parameters, and then broadcast to all the processes. If comm is not specified, then the communicator of the window is used. If an option is listed more than once in the parameter file, the last value for that option will overwrite the others.

E.2.3 Obtain DataItem

o obtain array data from files, the following function should be used:

- void obtain_dataitem(const DataItem *DataItem_in,
  DataItem *DataItem_user,
  int *pane_id=NULL);

This function fills the second (destination) DataItem from the files using the data corresponding to the first (source) DataItem. The destination and source DataItems can be the same. The DataItems could be a user-defined DataItem, or an aggregate DataItem, such as "window.conn", "window.mesh", "window.pmesh", "window.atts", and "window.all", which indicate obtaining connectivity tables only, mesh only (nodal coordinates and connectivity tables), mesh with pane connectivity, DataItems (everything except for pmesh), and everything (including pmesh and DataItems), respectively. If the third argument is present and is nonzero, then only the pane with the given ID will be copied.

E.2.4 Initialization and Finalization

SimIN provides the following routines for initialization and finalization.

- extern "C" void *SimIN*_load_module(const char *module_name);

Usually this procedure is invoked by COM_load_module("*SimIN*", module_name). It creates a window with name <module_name> in *COM* and register its functions into the window.

- extern "C" void *SimIN*_unload_module(const char *module_name);

This procedure is typically invoked by COM_unload_module("*SimIN*", module_name). It unloads the module from *COM* by deleting the window created by *SimIN*_load_module.

E.3 Implementation Notes

n read_window, in general, only metadata are read into memory to create windows. The data buffers of the windows may or may not be allocated yet. In obtain_dataitem, *SimIN* obtains data from the files to fill in user buffers. However, for certain file formats, an implementation of *SimIN* may read in physical data during read_window as well. The downside of the latter approach is higher memory requirements.

The function obtain_dataitem can permute memory layout of an DataItem. In general, an DataItem in *SimIN* can have either staggered or contiguous layout with a stride 1, but the user DataItem can have either staggered or contiguous layout and can also have a stride other than 1. The function obtain_dataitem support all these layouts.

The functions in the API can be implemented as C++ static member functions of *SimIN*, or regular member functions. In the former case, the functions are registered with *COM* using COM_set_function; in the latter case, they are registered using COM_set_member_function. *SimIN* works even if MPI_Init was not called. *SimIN* must be Charm-safe in the sense that there is no global (or static) variable [Current implementation is not yet Charm-safe].

E.4 *SimOUT*

E.4.1 Functionality

SimOUT writes a given DataItem in a *COM* window into a file in one of the supported formats (HDF and CGNS), which can be read by application codes through *SimIN*, and by *Rocketeer* (and CGNS-compliant tools, such as Tecplot, for CGNS format) for visualization. *SimOUT* can support background output by creating an I/O thread to allow overlap computation with I/O.

E.4.2 API

Similar to *SimIN*, *SimOUT* API typically should be called through COM_call_function.

E.4.3 Output

- void write_dataitem(const char *filename_pre,
  const COM::DataItem *attr,
  const char *material,
  const char *timelevel,
  const char *mfile pre = NULL,
  const MPI_Comm *comm=NULL,
  const int *pand_id=NULL);

This function writes an DataItem of local panes or of the pane with the given Pane ID (*pane_id, if present) into a file, where the file name is <fname_pre><process_rank>.<suffix>, where <process_rank> is the rank of the given MPI process, whose number of digits can be controlled by set_option() (see below). This function will either overwrite the file if the output mode (set by set_option()) is “w” or append to the file if the mode is “a”.

If mfile_pre is not null and nonempty, then the output file will make a reference to the file <mesh_pre><process_rank>.<suffix for the pmesh data with the same material name, and write only non-pmesh data into the current file. When appending data DataItems into a file that already contains the pmesh, then mfile_pre should be the same as filename_pre.

When calling write_dataitem multiple times to write several datasets into the same set of HDF files, it is important that the write operations for different panes must not interleave (i.e., the data for the same pane must be written out consecutively). In general, different windows can be written into the same set of files, but these windows must have different material names.

Parameters:

1. fname_pre: the prefix of the file name, which can contain the directory part of the file.
2. attr: a reference to the DataItem to be written. The given DataItem can be either a user defined DataItem, or one of the following aggregate DataItems: "window.mesh" (coordinates and connectivity), "window.pmesh" (mesh with pane connectivity), "window.atts" (all the data in the pane except for pmesh), or "window.all" (all the data).
3. material: the material name to distinguish different windows. It is recommended that different windows use different material names, and is required if more than one window is written into the same HDF/CGNS file.
4. timelevel: a time stamp of the dataset.
5. mfile_pre: the prefix of the name of the file that contains the pmesh data of the given DataItem. If not present or is empty, then the pmesh will be written along with the given data DataItems. If mfile_pre does not start with "/" (i.e., does not have an absolute path), then the path of the mesh file must be either relative to the directory for fname_pre (with higher precedence) or relative to the current working directory (with lower precedence).
6. comm: the MPI communicator in which the process rank should be obtained. If comm is NULL, then the default value is the communicator of the owner window of the DataItem (note that the default value is different from that with *SimIN*::read_window).
7. pane_id specifies the pane (or panes) to be written. If pane_id=NULL or *pane_id=0, then all panes will be written. If *pane_id>0, then only that specific pane will be written. It is an error if *pane_id<0.

Instead of using set_option to control the output mode, a user can also use one of the following two functions, which correspond to overwrite and append, respectively. [To be implemented.]

- void put_dataitem(const char *filename_pre,
  const COM::DataItem *attr,
  const char *material,
  const char *timelevel,
  const char *mfile pre = NULL,
  const MPI_Comm *comm=NULL,
  const int *pand_id=NULL);
- void add_dataitem(const char *filename_pre,
  const COM::DataItem *attr,
  const char *material,
  const char *timelevel,
  const char *mfile pre = NULL,
  const MPI_Comm *comm=NULL,
  const int *pand_id=NULL);

These functions take the same arguments as write_dataitem.

E.4.4 Metadata Output

- void write_rocin_control_file(const char *window_name,
  const char *file_prefixes,
  const char *control_file_name);

This function generates a control file for *SimIN* for the given window and datafile prefixes. This control file can be used with *SimIN*'s read_by_control_file member function.

- void write_parameter_file(const char *file_name,
  const char *window_name,
  const MPI_Comm *comm=NULL);

This function writes out the parameters defined in the given window to a parameter file. Only process 0 of the MPI communicator writes the file. If comm is NULL, then the communicator of the window associated with window_name is used.

E.4.5 Synchronization

- void sync();

Wait for the completion of an asynchronous write operation. It is needed only if the ”async” mode is set to “on” by set_option, described as follows.

E.4.6 Control

- void set_option(const char *option_name,
  const char *option_val);

Set an option for *SimOUT*, such as controlling the output format. The currently supported option_name and their potential values are:

“format”: with values “HDF” and “CGNS” (default is “HDF”).
“async”: with values “on” and “off” for enabling/disabling background out, respectively (default is off).
“mode”: with values “w” and “a” (corresponding to overwrite the file and append to the file), which control the output mode of write_dataitem (default is “w”’).
“localdir”: a directory path to prepend to the filename prefixes given to write_dataitem, put_dataitem and add_dataitem (default is “”).
“rankwidth”: the width of the process-rank to be appended to the filename_pre and mesh_pre (default is “4”). If zero, then do not append process rank.
“pnidwidth”: the width of the pane ID to be appended to the filename_pre and mesh_pre after appending process rank. Default value is 0, for which the pane ID is not appended.
“separator”: the character to use to separate the rank and pane ids in generated filenames (default is “_”). A separator is only used if both “rankwidth” and “pnidwidth” are non-zero.
“errorhandle”: with values “abort”, “ignore”, or “warn”.
“ghosthandle”: with values “write” and “ignore”.

Option names and values are case-sensitive.

- void read_control_file(const char *filename);

This function allows the user to set *SimOUT* options by means of a control file. The given file should be a list of option name/values pairs, separated by an equals sign. For example:

```
format = CGNS
localdir = /turing/projects/csar/MyDataDir
errorhandle = abort
```

Any option name supported by set_option may be used.

E.4.7 Initialization and Finalization

As *SimIN*, *SimOUT* provides the following routines for initialization and finalization.

- extern "C" void *SimOUT*_load_module(const char *module_name);

Usually this procedure is invoked by COM_load_module("*SimOUT*", module_name). It creates a window with name <module_name> in *COM* and register its functions into the window.

- extern "C" void *SimOUT*_unload_module(const char *module_name);

This procedure is typically invoked by COM_unload_module("*SimOUT*", module_name). It unloads the module from *COM* by deleting the window created by *SimOUT*_load_module.

E.4.8 Implementation Notes

The functions in the API can be implemented as C++ static member functions or regular member functions of *SimOUT*. In the former case, the functions are registered with *COM* using COM_set_function; in the latter case, they are registered using COM_set_member_function. *SimOUT* works even if MPI_Init was not called, in which case the rank is assumed to be 0. *SimOUT* must be Charm-safe in the sense that there is no global (or static) variable.

E.4.9 Sample Code

Samples codes of *SimIN* and *SimOUT* can be found under SimIO/IN/test and SimIO/OUT/test, respectively. These are built automatically with *MPACT*.

F Simpal User's Guide

***Simpal* Users Guide**
Version 0.1.0 IllinoisRocstar LLC October 25, 2016

License

The software package sources and executables referenced within are developed and supported by Illinois Rocstar LLC, located in Champaign, Illinois.The software and this document are licensed by the University of Illinois/NCSA Open Source License (see `opensource.org/licenses/NCSA`). The license is included below.

For more information regarding the software, its documentation, or support aggreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

F.1 Overview

Simpal is an *IMPACT* service module which provides some basic algebraic operations on DataItems registered with *COM*. Written in C++ using *COM*'s developers interface, *Simpal*'s interface functions take pointers to COM::DataItem objects or scalar numbers as arguments, and should in general be invoked through *COM* using COM_call_function. In the implementation, each process computes on the datasets of its local panes. Most operations implemented in *Simpal* are embarrassingly parallel requiring no inter-process or inter-pane communications.

F.2 Requirements and Conventions

All operations of *Simpal* share similar requirements and conventions on the operands, listed as follows.

- All operands must have the same base data type.
- All nodal or elemental DataItems of an operation must be associated with the same set of nodes or elements.
- At least one of the input operands must be a nodal or elemental DataItem.
- For functions taking two inputs and without the **_scalar** suffix (including **add**(), **sub**(), **mul**(), and **div**()), all input operands must be DataItem objects, and one is allowed to be a panel or windowed DataItem. In the corresponding functions with the **_scalar** suffix, the second input argument must be a scalar number.
- Each DataItem operand has two versions: a scalar version (with only one entry per entity) and a vector version (with only one entries per entity). An DataItem operand can be either one, except that for swap and copy, the two DataItem operands must have the same number of components.
- DataItem operands with more than one data entry per entity (where the entity is a window, pane, a node, or an element for windowed, panel, nodal or elemental, respectively) must have the same number of entries per entity.
- It is legal to use a DataItem with only one entry per entity in place of a DataItem with more than one entry per entity. *Simpal* will use the value of that single entry to compute against with all entries of another operand.
- In most cases, it is legal to mix contiguous and staggered layouts of nodal and elemental DataItems.
- Output arguments are usually the last arguments, except for functions with optional input arguments.

Most of these requirements are checked at runtime by *Simpal* if possible. A violation will cause a run to abort. A user, however, can disable error checking at compile time by passing `-DNDEBUG` to the C++ compiler.

F.3 *Simpal* Interface

F.4 Supported Operations

The following table lists the operations supported by *Simpal*, in which we use S, W, P, N, and E to abbreviate different types of operands.

S = scalar pointer
W = windowed DataItem
P = panel DataItem
N = nodal DataItem
E = elemental DataItem

$+, -, \times, \div$	$N = N \diamond W, N = N \diamond P, N = N \diamond N;$ $E = E \diamond W, E = E \diamond P, E = E \diamond E$
scalar $+, -, \times, \div$	$N = N \diamond S, E = E \diamond S$
dot	$W/P/N = \langle N, N\rangle, W/P/E = \langle E, E\rangle$
dot-scalar	$S = \langle N, N\rangle, S = \langle E, E\rangle$
2-norm	$W/P/N = \|2\|_2 N, W/P/E = \|2\|_2 E$
2-norm-scalar	$S = \|2\|_2 N, S = \|2\|_2 E$
swap	$N \leftrightarrow N, E \leftrightarrow E$
neg	$N = -N, E = -E$
copy	$N = W, N = P, N = N, E = W, E = P, E = E$
copy-scalar	$N = S, E = S$
axpy	$N = WN + N, N = PN + N, N = NN + N$ $E = WE + E, E = PE + E, E = EE + E$
axpy-scalar	$N = SN + N, E = SE + E$

F.5 *Simpal* API

- void **Simpal_load_mudule**(const char name)
  void **Simpal_unload_mudule**(const char *name)

  Loads/unloads *Simpal* to/from *COM* by creating a service CI window of the given name and register its functions.

- void **add**(const DataItem *x, const DataItem *y, DataItem *z)
  void **sub**(const DataItem *x, const DataItem *y, DataItem *z)
  void **mul**(const DataItem *x, const DataItem *y, DataItem *z)
  void **div**(const DataItem *x, const DataItem *y, DataItem *z)

  Performs the operation $\mathrm{z} = \mathrm{x}\, op\, \mathrm{y}$, where op is $+, -, \times$, or $\div$. The output argument z must be nodal or elemental; one of x and y must have the same type as z, and the other can have the same type or be a windowed or panel DataItem. If all operands are nodal or elemental, then the operation is performed node-wise or element-wise, respectively. If one of the operand is windowed, then its value is used in the operation of every node/element. If one of the operand is panel, then its value in a pane will be uses in the operations on the nodes/elements of this pane. It is legal to have the same DataItem to appear multiple times in the operands.

- void **add_scalar**(const DataItem *x, const void *y, DataItem *z, const char *swap = NULL)
  void **sub_scalar**(const DataItem *x, const void *y, DataItem *z, const char *swap = NULL)
  void **mul_scalar**(const DataItem *x, const void *y, DataItem *z, const char *swap = NULL)
  void **div_scalar**(const DataItem *x, const void *y, DataItem *z, const char *swap = NULL)
  Performs the operation $\mathrm{z} = \mathrm{x}\, op\, \mathrm{y}$ or $\mathrm{z} = \mathrm{y}\, op\, \mathrm{x}$, if swap is null or not, respectively, where op is $+, -, \times$, or $\div$. Here, the scalar behaves similar to a window DataItem in their corresponding functions

without **_scalar**. It is a user's responsibility to ensure that the data type of the scalar is the same as the base data types of the DataItems. Again, it is legal to have the same DataItem to appear multiple times in the operands.

- void **maxof_scalar**(const DataItem *x, const void *y, DataItem *z)
  Performs the operation z = max(x, y). It is a user's responsibility to ensure that the data type of the scalar is the same as the base data types of the DataItems. Again, it is legal to have the same DataItem to appear multiple times in the operands.

- void **dot**(const DataItem *x, const DataItem *y, DataItem *z, const DataItem *mults = NULL)

  Performs the operation $z = \langle x, y \rangle$. The inputs x and y must be nodal or elemental. The output z can be windowed, panel, nodal, or elemental. If z is windowed, then the result is the dot product for x and y over the whole window. If z is panel, the result is over the each pane. If z is nodal or elemental, then the results are over each node or element (i.e., the value associated with a node is treated as vectors).

  The optional argument `mults` specifies the multiplicity of the nodes or elements, i.e., the number of times a node or pane appears in the window. It is useful only when z is a windowed DataItem. The product of of the values associated with a node or element will be divided by its multiplicity before being summed. When no value is passed for `mults`, then a multiplicity of 1 is assumed for each node or element.

  Note: An MPI all-reduce is needed for this operation if the solution is a scalar.

- void **nrm2**(const DataItem *x, const DataItem *y, const DataItem *mults = NULL)

  Performs the operation $y = \| x \|_2$. This function works the same as **dot**(), except that it takes only 1 required input argument instead of 2.

- void **swap**(DataItem *x, DataItem *y)

  Swaps the contents of x and y. x and y must be nodal or elemental and must have the same number of entries per entity.

- void **neg**(DataItem *x)

  Negate the signs of the values of x, where x must be nodal or elemental and must have the same number of entries per entity.

- void **copy**(const DataItem *x, DataItem *y)

  Assigns the value of x to y. The argument y must be nodal or elemental. x can be windowed, panel, nodal, or elemental. If x is windowed, then its value is assigned to every node or element in y. If x is panel, then each node or element of y receives the value of x associated with its pane. If x is nodal or elemental, then each node or element of y receives the value of its corresponding node or element of x.

- void **axpy**(const DataItem *a, const DataItem *x, const DataItem *y, DataItem *z)

Performs the saxpy operation `z` = `ax` + `y`. `x`, `y`, and `z` must be nodal or elemental. The DataItem `a` can be windowed, panel, nodal, or elemental.

- void **dot_scalar**(const DataItem *x, const DataItem *y, void *z, const DataItem *mults = NULL)
  void **nrm2_scalar**(const DataItem *x, void *z, const DataItem *mults = NULL)
  void **copy_scalar**(const void *x, DataItem *y)
  void **axpy_scalar**(const void *a, const DataItem *x, const DataItem *y, DataItem *z)

  Has the same semantics as their corresponding version without **_scalar**, except that the scalar argument acts in place of a window DataItem.

F.6 Building and Testing *Simpal*

Simpal is integrated and built as a part of *IMPACT*. *Simpal* comes with a test program named blastest.C in the test subdirectory, and the test is built automatically with *IMPACT*. The test program takes no command-line arguments. Instead, it will prompt for a user to choose interactively the types and layout of operands and the operations to be tested.

G SurfX User's Guide

SurfX
Version 0.1.0 IllinoisRocstar LLC October 25, 2016

License

For more information regarding the software, its documentation, or support agreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

G.1 Overview

SurfX is a service utility for transferring data between surface meshes in a fashion that is numerically accurate and physically conservative. *SurfX* transfers data in two steps. First, it constructs a common refinement of two surface meshes using a sophisticated algorithm described in Jiao's thesis. This step is done sequentially and is typically done off-line. Second, it transfers data using a least squares formulation that minimizes errors in the L_2 norm. The user also has the option of using a simple interpolation scheme for the second step, but it will not be conservative.

SurfX supports both node-centered and face-centered data, and can transfer between these two types of data in any combination. It supports both multi-block structured meshes and unstructured surface meshes with mixed elements. For unstructured meshes, both first-order (3-node triangles and 4-node quadrilaterals) and second-order (6-node triangles and 8- or 9-node quadrilaterals) elements are supported.

SurfX was implemented in C++ using *COM*'s API. Interprocessor communication is done through MPI.

G.2 *SurfX* API

SurfX should typically be called through *COM* by an orchestrator (such as *Rocman* in *Rocstar*). An application does not interact directly with *SurfX*, except that they must register CI windows with *COM*. Note that *SurfX* can only take nodal and elemental data attributes with contiguous layouts (i.e., no staggered layout). See *COM* Users Guide for how to create CI windows. In this section, we will describe only the function interface to be called from an orchestration module.

SurfX provides subroutines *SurfX*_load_module and *SurfX*_unload_module. Each takes a window name as an input. Typically, the name is “RFC”. These two routines are the only ones that are not called through COM_call_function. All the interface functions of *SurfX* are registered with *COM* as a member function of a *SurfX* object, which encapsulates all the internal context of *SurfX*. In the following, we omit the *SurfX* object in the argument list.

G.2.1 Overlaying Meshes

Construction of Overlay Because the overlay algorithm is sequential, the construction of the overlay is typically done off-line using the surfdiver utility described in Section G.4. In a sequential run, a user may want to construct the overlay on-line using the following interface.

```
overlay( const COM::Attribute *mesh1, const COM::Attribute *mesh2, const MPI_Comm
    *comm=NULL, const char *path=NULL, const double *tol1=NULL, const double *tol2=NULL)
```

The first two arguments are two *COM* objects referring to the mesh data of two windows. Only these two arguments are mandatory and are sufficient for most cases. The third argument comm is an MPI communicator, which should be the same as the communicator on which the two input windows are distributed. The argument path is the path where the output files should go. The remaining two arguments control the tolerancing schemes used in the overlay algorithm, and they should be left blank.

A user can clean up the memory allocated for an overlay using the following interface.

```
clean_overlay( const char *win1, const char *win2)
```

It takes the window names of the two meshes as input.

I/O of Overlay After constructing the overlay, a user can write it out into binary files using the interface write_overlay_sdv.

write_overlay_sdv(const COM::Attribute *mesh1, const COM::Attribute *mesh2, const char *prefix1=NULL, const char *prefix2=NULL)

The first two arguments are two *COM* objects referring to the mesh data of two windows. Only these two arguments are mandatory and are sufficient for most cases. The third and fourth arguments are the prefixes that should be used for the output files. The default are the window names.

The overlay files can be read in using the following interface.

read_overlay_sdv(const COM::Attribute *mesh1, const COM::Attribute *mesh2, const MPI_Comm *comm, const char *prefix1=NULL, const char *prefix2=NULL)

It takes an additional argument comm before the prefixes. This routine need to be called in parallel simulations, and comm should be the communicator on which the CI windows are defined.

G.2.2 Data Transfer

After constructing an overlay or loading it in from files, the user can invoke data transfer for two attributes using the interface least_squares_transfer.

least_squares_transfer(const COM::Attribute *att1, COM::Attribute *att2, const int *ord=NULL, double *tol=NULL, int *iter=NULL, const int *v=NULL)

Again, the first two arguments correspond to the source and target attributes, respectively, and only these two are mandatory. The remaining arguments typically should not be used. The third argument specifies what order of accuracy *COM* should use for quadrature rules, and the default is 2. The tol and iter arguments specify the convergence criteria of the linear solver that *SurfX* uses. The last argument specifies the verbose level, and the default is zero.

If a user would like to use interpolation for speed, the following interface can be used instead, whose argument list is a subset of that of least_squares_transfer.

interpolate(const COM::Attribute *att1, COM::Attribute *att2, const int *v=NULL)

G.3 Compiling *SurfX*

SurfX can be compiled using only the C++ compilers that reasonably conform to the ISO/IEC C++ standard with supports for namespace, template, and STL. *SurfX* is known to compile with g++-2.91 or later on various platforms (including Linux, Sun, SGI Origin 2000, IBM SP/2), SGI CC-7.30 and KAI CC-4.0 on SGI Origin 2000, and the native C++ compiler (xlC) on IBM SP.

SurfX depends on *COM*, *SurfMap*, *SurfUtil*, *SimIO*, and *Simpal*, and builds under the *CMake* build system. The recommended procedure is to do an “out-of-source” build by creating a build directory and then invoking cmake on the source directory from the build directory. It is highly recommended to set CXX=mpicxx before building *SurfX*.

G.4 *Surfdiver*

Surfdiver is a preprocessor of *SurfX* for generating a common refinement of two surface meshes. It takes two ASCII interface meshes files as input and generates binary overlay files for *SurfX*. These binary outputs are compatible across all platforms that use either big- or small-endian. Therefore, you can run surfdiver on any of your favorite platform and then take the output to the any machine.

G.5 Advanced Tuning for Feature Detection

SurfX automatically detects the corners and ridges of a surface mesh. For most models, the default parameters set by *SurfX* would work. For some complex models, one can control the parameters of feature detection by providing a control file <window name>.fc. This file should have five lines:

1. The first line contains four parameters for face angle: cosine of the upper bound, cosine of the lower bound, the signal-to-noise ratio, and cosine of an open-end of a 1-feature.
2. The second line contains three parameters for angle defect: upper bound, lower bound, and signal-to-noise ratio.
3. The third line contains three parameters for the turning angle: cosine of the upper bound, cosine of the lower bound, and the signal-to-noise ratio.
4. The fourth line contains four parameters controling the filteration rules: the minimum edge length for open-ended 1-features, whether to apply the long-falseness filteration rule, whether to apply the strong-end filtration rule, and whether to snap 1-features of input meshes on top of each other.
5. The fifth line controls the verbosity level. The default value is one. Setting vebosity level to greater than one will instruct *SurfX* to write out HDF files "*_fea.hdf", which contain the feature information and are very helpful for fine tuning feature detection.

The following is a sample control file containing the default values.

```
0.76604444 0.98480775  3 0.98480775 # Feature angles
1.3962634  0.314159265 3            # Angle defects
0.17364818 0.96592583  3            # Turning angles
6 1 0 0                             # Filteration rules
2                                   # Verbosity level
```

If the control file is missing, then the default values (as shown above) will be used. These default values should work for most cases. For some other cases, it typically suffices to adjust the signal-to-noise ratios (the third parameters of the first three lines) and the minimum edge length (first parameter of line 4) should suffice.

G.5.1 Fine-Tuning Parameters

If it ever becomes necessary to fine-tune the feature-detection parameters, adjusting only the parameters of feature angles (i.e., the first line of the .fc files) typically suffices. The following procedure is useful in determining the proper values of feature angles:

1. Find the ifluid_fea*.hdf and isolid_fea*.hdf files in the output directory. These files are generated by surfdiver if the ".fc" files are present and the verbosity level in these files are greater than 1.
2. Convert these HDF files into Tecplot files using the utility hdf2plt. Typically, the commands look like (note that the quotation marks are important):
   (a) hdf2plt "ifluid_fea*.hdf" ifluid.plt
   (b) hdf2plt "isolid_fea*.hdf" isolid.plt
3. Load the ifluid.plt and isolid.plt into two separate Tecplot sessions. Look at the contour of "frank" (stands for feature rank) to eyeball the discrepancies of the detected features in the input meshes.
4. Use the "probe" tool of Tecplot to look at the values of "fangle" (stands for feature angles), and adjust the feature-angle thresholds based on the values "fangle" of false features. Typically, if some edges are marked as features in correctly, then one should increase the feature-angle thresholds; if some feature edges are missing, then one should decrease the feature-angle thresholds.

G.6 Test Problems

SurfX includes a series of test problems in the directory test. These problems are meant to test the robustness of the mesh overlay algorithm.

H SurfMap User's Guide

SurfMap User’s and Developer's Guide

Illinois Rocstar LLC
1800 S. Oak, STE 108
Champaign, IL 61820

1.0 Introduction

1.1 *SurfMap* Overview

SurfMap provides a small set of commonly used functions for communication among mesh panes in the *COM* infrastructure. It is implemented in C++ and uses the *COM* infrastructure. Typically, *SurfMap*'s static member functions are called from within other *COM* modules. Nevertheless, an interface is provided for using *SurfMap* as a *COM* module. It builds on all platforms supported by *Rocstar* 3.

1.2 Related Documents

The information in this guide is supplemented by the following documents:

- "*COM* User's Guide".
- "*SimIO* User's Guide".

2.0 Purpose and Methods

SurfMap aids in the development of *COM* modules dealing with partitioned meshes by providing commonly required inter-pane communication functions. An API provides access to a few of these functions when the *SurfMap* module is loaded into the *COM* infrastructure. Other functionality is available at a lower level through C++ classes. See 3.3 discusses *SurfMap* as a *COM* module. The important *SurfMap* classes are discussed in this section.

2.1 Dual_connectivity

Element connectivity tables are a fundamental piece of a *COM* mesh. These tables list the nodes which constitute each particular element. The dual connectivity table is a similar structure which lists elements contained in a particular node. *COM* does not require the registration of a dual connectivity table, but this class is available for compiling and accessing that information.

2.1.1 Constructors

```
explicit Pane_dual_connectivity( const COM::Pane *p, bool with_ghost = true)
```

Construct a Pane_dual_connectivity for pane **p**. If **with_ghost** is *true*, then include ghost entities in the construction.

2.1.2 Important Member Functions

`void incident_elements(int node_id, std::vector<int>& elists)`

Fills **elists** with the list of elements containing the node with local id **node_id**.

2.2 Pane_boundary

Determines which nodes are on pane boundaries and which nodes are isolated. An Isolated node is one which does not belong to any element. This can happen when a surface mesh is extracted from a volume mesh. If an element of the volume mesh touches the surface with only a single node, then that node will be isolated in the surface mesh.

2.2.1 Algorithms

Isolated nodes are determined in a straight forward fashion. The isolated node vector is initialized to false for every node. Then, a pass is made over the element connectivity table(s) during which every node seen is marked as not isolated.

Determining which nodes (and facets) are on the border is slightly more complicated. Conceptually, this algorithm is simply facet matching. Facets which are not on the border are sandwiched between two elements. If we consider facets as unordered sets of nodes, then these facets are duplicated on the adjacent elements. Therefore, finding unique facets is equivalent to finding the border.

Again, a pass is made over the element connectivity table(s). An inner loop iterates through every facet of the current element. A four-tuple is created from the ids of the facet's constituent nodes. If the facet is triangular, then *-1* is used as the fourth node id. We compare the four-tuple to the set of all those already seen. If we find a duplicate, then both four-tuples are discarded. After the entire element connectivity is processed, we are left with the set of border faces from which computing border nodes is trivial.

2.2.2 Constructors

`Pane_boundary( const COM::Pane *p)`

Build a Pane_boundary for pane **p.**

`Pane_boundary( const Simple_manifold_2 *pm)`

Build a Pane_boundary for the pane associated with the given manifold.

2.2.3 Other member functions

```
determine_border_nodes(
        std::vector<bool> &is_border,
        std::vector<bool> &is_isolated,
        std::vector<Facet_ID > *b=NULL,
        int ghost_level=0) throw(int)
```

Sets **is_border**[node_id -1] to *true* if the node assocaited with the id is on the border, and to *false* otherwise. Stores similar information for isolated nodes in **is_isolated**. If **b** is not NULL, then it is filled with the set of faces which are on the pane border. The interpretation of **ghost_level** is differenrt for structured and unstructured meshes. If the target pane has a structured mesh, it determines how many ghost layers to include in the calculation. If **ghost_level** > 0 and the mesh is unstructured, the all registered ghost entities are considered.

```
static void determine_borders(
        const COM::DataItem *mesh,
        COM::DataItem* is_border,
        int ghost_level = 0)
```

Determines border nodes for the entire window associated with **mesh** and stores this information in **is_border**. The **ghost_level** parameter is the same as in `determine_border_nodes`. Note that the DataItem stored is still *pane* border, and not the border of the mesh as a whole.

2.3 Pane_ghost_connectivity

Adds a single layer of ghost elements to each pane by constructing and registering the complete 5-block pconn for the mesh as defined in the *COM* User's Guide. This is a single layer of ghost elements in the sense that all non local elements incident on a local real node are ghosted as are any nodes which are contained in these elements but not local to the pane. In Rocflu's terminology, this is a second-order mesh.

2.3.1 Algorithms

The basic idea of this algorithm is that each pane determines which local elements should be ghosted on other panes, and sends the connectivity information for those local elements to the appropriate panes. Shared node information is built using the Pane_connectivity class, then each local element is examined. If the local element contains a shared node, then the element is added to a set of elements which needs to be sent to the pane(s) with which that node is shared. After all elements have been examined, we can calculate how much information is to be sent to each adjacent pane. This size information is communicated, followed by the actual connectivity data. After all connectivity information is received, the pconn is constructed and registered.

COM itself does not use a global order for nodes or elements, however, we need that information because we will sometimes receive elements from multiple panes which share the same non local node, and we have to recognize that this is a single ghosted node rather than two separate nodes.

2.3.2 Constructors

`Pane_ghost_connectivity( COM::Window *window)`

Construct a Pane_ghost_connectivity for window **window.**

2.3.3 Important Member Functions

`void build_pconn()`

This is the highest level function, it builds and registers the pconn.

`void init()`

The initialization routine builds and registers the shared-node section of the pconn. It also determines border nodes, obtains a list of communicating panes, and creates data structures for the total node ordering.

`void get_node_total_order()`

Obtains a total ordering of nodes in the form of an ordered pair <P,N> where P is the "owner pane" and N is the node's local id on the owner pane. The "owner pane" is the pane with highest id which contains a real copy of the node. For each node, N is initialized to 0 and P is initialized to the local pane id. Then, the shared node section of the pconn is examined. P is updated each time a node is found being shared with a pane with a higher pane id than the currently stored value. At this point, all the owner panes have been determined. Each node owned by the local pane is now assigned its local node id as the N value. An MPI maximum reduction is performed on the N values to finalize the total ordering.

```
void get_ents_to_send(
    vector<vector<vector<int> > > &gelem_lists,
    vector<vector<map<pair<int,int>,int> > > &nodes_to_send,
    vector<vector<deque<int> > > &elems_to_send,
    vector<vector<int> > &comm_size)
```

This function determines the local elements and nodes to be remotely ghosted on communicating panes. A dual connectivity data structure is built which allows for querying which elements are adjacent to each node. The list of nodes shared with each communicating pane is examined, and all elements containing any of these nodes are added to a set to be sent to the communicating pane. Simultaneously, the eventual size of the element connectivity information to send is calculated and stored. Each elements type and node list (as <P,N> pairs) will be sent. The actual information to be send is written to **gelem_lists**, while the lists of the local nodes and elements to be remotely ghosted are stored in **nodes_to_send** and **elems_to_send**.

COM requires that the elements listed in one pane's real-elements-to-send list correspond to the element listed in the same index of the communicating pane's ghost-elements-to-receive list, and similarly for the ghost node communication sections. We fulfill this requirement as follows.

Owner panes list real-elements-to-receive in the same order that the elements are placed in the **gelem_lists** data structure, and the pane which ghosts the elements maintains this ordering. Similarly, as element connectivities are written into the **gelem_lists** structure, the owner pane checks to see if the node is already shared with the communicating pane. If not, then that node will need to be ghosted on the remote pane. These nodes are listed in the real-node-to-share list in the same order that they are first observed. When the remote pane processes the element connectivity lists, the same method is used.

```
void process_received_data(
      vector<vector<vector<int> > > &recv_info,
      vector<vector<int> > &elem_renumbering,
      vector<vector<map<pair<int,int>,int> > > &nodes_to_recv)
```

Determines the number of ghost nodes to receive, assigns them local ghost node ids, and maps <P,N> to those ids. Also determine the number of ghost elements of each type to receive. Ghost element ids can not be determine as elements arrive if we want to have a single connectivity table per element type because *COM* requires that elements in a single connectivity table be numbered consecutively.

```
void finalize_pconn(
      vector<vector<map<pair<int,int>,int> > > &nodes_to_send,
      vector<vector<map<pair<int,int>,int> > > &nodes_to_recv,
      vector<vector<deque<int> > > &elems_to_send,
      vector<vector<int> > &elem_renumbering,
      vector<vector<vector<int> > > &recv_info)
```

Takes the data we've collected, and builds and registers the pconn. The shared node information is already present, so there are four blocks to build: real-nodes-to-send, ghost-nodes-to-receive, real-elements-to-send and ghost-elements-to-receive. Data for the first three of these blocks is available in the correct order from the first three input parameters to the function. Creating the ghost-elements-to-receive block of the pconn requires combining information in the latter two data structures.

First, the input data structures are examined to determine the size of each ghost block of the pconn. Connectivity tables are resized to accommodate the new ghost entries, and the pconn is resized to accommodate the four new blocks. Next we actually fill in the new sections of the pconn. We also fill in the ghost element's connectivity into the correct connectivity table by taking into account both the element's type and the order in which it was received. Finally, we extend the nodal coordinate DataItem to accommodate the new ghost nodes, and update those coordinates from their real coordinates using the newly created pconn.

```
void get_cpanes()
```

Get the list of communicating panes for each local pane. A communicating pane is any pane with which this pane shares a node. This data is stored in **_cpanes**.

```
void send_gelem_lists(
      vector<vector<vector<int> > > &gelem_lists,
      vector<vector<vector<int> > > &recv_info,
      vector<vector<int> > &comm_sizes)
```

Send ghost element connectivity lists to the communicating panes.

```
void send_pane_info(
      vector<vector<vector<int> > > &send_info,
      vector<vector<vector<int> > > &recv_info,
      vector<vector<int> > &comm_sizes)
```

Sends an arbitrary amount of data to the communicating panes.

```
void determine_shared_border()
```

Determines whether or not each local node is shared.

```
void mark_elems_from_nodes(
       vector<vector<bool> > &marked_nodes,
       vector<vector<bool> > &marked_elems)
```

Takes the given Boolean nodal property, and extends that property to elements. An element is considered to have the property if any of its constituent nodes have the property.

3.0 Building and Running

SurfMap is written in C++ and uses the *CMake* infrastructure with *MPACT.* The source and CMakeLists.txt are found at /MPACT/SurfMap in the *MPACT* distribution. The *SurfMap* module is built automatically by the *MPACT* build system as libSurfMap.so

The former contains utility programs automatically built by the makefiles, while the latter contains all libraries including *COM*'s and *SurfMap*'s.

3.1 Library Dependencies and Building

SurfMap is integrated into *MPACT* and is built along with it. Because *SurfMap* links to the *COM* library, it may only be built directly from its own directory if the *COM* library already exists.

3.2 *SurfMap* Build Targets

The default *COM* build creates the *SurfMap* dynamic library as well as a *SurfMap* utility named “addpconn”. Several other test programs are included in *SurfMap*/test, and may be built individually:

Test Program	***Description***
bordertest_hex	Demonstrates determination of pane borders through *SurfMap* on an unstructured hex mesh. Builds an unstructured hex mesh and uses *SurfMap* to determine which nodes are on the border. Dumps the mesh into an .hdf file, "hexmesh", with node border information stored in the DataItem "borders".
bordertest_struc	Demonstrates determination of pane borders through *SurfMap* on a structured hex mesh. Creates a structured hex mesh, determines which nodes are on the pane border, and dumps the mesh into an .hdf file, "strucmesh"

3.3 *COM* Accessible Functions (*SurfMap* API)

This section describes the set of functions which is available through the *COM* infrastructure when *SurfMap* is registered as a module. All of these functions are *static void* member function of the *SurfMap* class.

compute_pconn(
const COM::DataItem *mesh,
COM::DataItem *pconn)

Computes the first block of the pconn DataItem described in the *COM* User's Guide. If pconn hasn't been initialized, then memory is allocated and the computed pconn block is saved. Otherwise, saves up to the capacity of the DataItem.

Parameter	***Description***
*mesh	The target mesh.
*pconn	The target mesh's pconn DataItem.

pane_border_nodes(
const COM::DataItem *mesh,
COM::DataItem *isborder,
int *ghost_level=NULL)

Determines which nodes are on the pane border, and saves this information to a *COM* DataItem.

Parameter	***Description***
*mesh	The target mesh.

Parameter	Description
*isborder	DataItem where border information will be saved.
*ghost_level	If > 0, include ghost nodes and elements as part of the pane.

reduce_average_on_shared_nodes(
COM::DataItem *att,
COM::DataItem *pconn = NULL)

Calculates an average DataItem value for each shared node across all sharing panes, and sets the DataItem value to that average on all sharing panes.

Parameter	Description
*att	Target DataItem
*pconn	Pconn of the mesh corresponding to the target DataItem

reduce_maxabs_on_shared_nodes(
COM::DataItem *att,
COM::DataItem *pconn=NULL);

Sets the value of each component of an DataItem to the value of that component of the DataItem with the largest magnitude. Note that this is done on a *component by component basis*, not in the sense of any norms.

Parameter	Description
*att	Target DataItem
*pconn	Pconn of the mesh corresponding to the target DataItem

update_ghosts(
COM::DataItem *att,
COM::DataItem *pconn = NULL)

Sets the value of an DataItem at ghost nodes or cells to the value on the corresponding real nodes or cells. In the case that a shared node has different values across its incident panes, it is undetermined which value each ghost node will receive.

Parameter	Description
*att	Target DataItem
*pconn	Pconn of the mesh corresponding to target DataItem

3.4 Other Functions (*SurfMap* API)

SurfMap()

SurfMap's constructor, does not perform any initialization.

static void load(const std::string &mname)
static void unload(const std::string &mname)

These functions are used for loading or unloading *SurfMap* from *COM* with the given module name.

4.0 Input and Output (User Interface)

SurfMap if typically used as a C++ object, though its functions other than the constructor are all static, so no instantiation is required. It may also be loaded as a *COM* module, and called through the standard *COM* interface.

4.1 *SurfMap* as a C++ Object

The following code fragment demonstrates the use of *SurfMap* as a C++ object:

```
// Assuming that "window" is a pointer to a COM window
// move shared nodes to their average position across all
// incident panes
SurfMap::reduce_average_on_shared_nodes(window->
     DataItem(COM::COM_NC));

// Update the ghost copies of the nodes with their new positions
SurfMap::update_ghosts(window->DataItem(COM::COM_NC),
     window->DataItem(COM::COM_PCONN));
```

4.2 *SurfMap* as a *COM* Module

SurfMap is typically loaded and invoked through the *COM* API as illustrated in the following C++ example:

```
// Load SurfMap into the COM infrastructure
COM_LOAD_MODULE_STATIC_DYNAMIC( SurfMap, "MAP");

// Get function handle for SurfMap::compute_pconn
int MAP_compute_pconn =
     COM_get_function_handle("MAP.compute_pconn");
```

```
// Get the handle for the fluids mesh
int mesh_hdl = COM_get_DataItem_handle("fluids.mesh");

// Get the handle for the pconn DataItem of the fluids mesh
int pconn_hdl = COM_get_DataItem_handle("fluids.pconn");

// Use SurfMap::compute_pconn to create the shared-node section
// of the pconn for the fluids mesh.
COM_call_function(MAP_compute_pconn, &mesh_hdl, &pconn_hdl);
```

5.0 Utilities and Test Programs

A default build of *SurfMap* creates “addpconn”, a utility for building the shared-node section of pconn for a given .hdf file. Source code for this utility is found in *SurfMap*/util/addpconn.C . Four other test programs are available in *SurfMap*/test/ . The source files for these programs also have the same name as the programs, but with “.C” appended. Section 3.2 explains how to build them.

5.1 addpconn

This utility reads in one or more .hdf files using *SimIN*. It makes a copy of the mesh from which it removes all mesh DataItems other than nodal coordinates and connectivity tables. *SurfMap* is then used to rebuild the pconn on the new mesh, which is written to file.

```
addpconn <input filename patters or Rocin control file>
         <output file prefix>
```

To run in parallel, a *SimIN* control file must be passed as a second argument. If the second argument ends in “.hdf”, then all panes are written out to a single pane. Otherwise, each pane is written to a separate .hdf file.

5.2 bordertest_hex

This test program runs without any command line input. It builds an unstructured hex mesh with 4 elements and 18 nodes. *SurfMap* is used to determine which nodes are on the border, and store this information in an DataItem. The mesh is then written out to “hexmesh0000.hdf”.

5.3 bordertest_struc

This program is similar to bordertest_hex, except that a much larger mesh is created, and it is a structured hex mesh. The output file is “strucmesh0000.hdf”.

I Physics of FSI Coupling in *ElmerFoamFSI*

Physics of Coupling
Version 0.1.0 IllinoisRocstar LLC
October 25, 2016

License

I.1 Fluid-Solid Interaction (FSI)

Fluid-solid interaction (FSI) is a multi-physical phenomenon in mechanics that involves non-linear interactions between a fluid and solid. Many physical problems involve FSI phenomenon, example can be taken from different time/space scales and from a variety of systems for example deformations of the blood vessels during blood circulation and the fluctuation of airplane wings during flight both involve FSI. In every FSI problem, a moving fluid (compressible like air, or incompressible like water) interacts with a deformable solid. These interactions usually involve energy and momentum transfer between the phases.

Physical balance laws can be resorted to explain an FSI problem, including conservation of mass, energy and momentum. In some simple configurations these laws can be used to directly develop an analytical solution for an FSI problem. For more complex problems (most of engineering problems in 2D/3D) an analytical solution, if not impossible, is very hard to obtain. Laboratory experiments provide a limited scope of the problem as well and the understanding of the fundamental physics involved in complex interactions between the phases can only be obtained by numerical simulations.

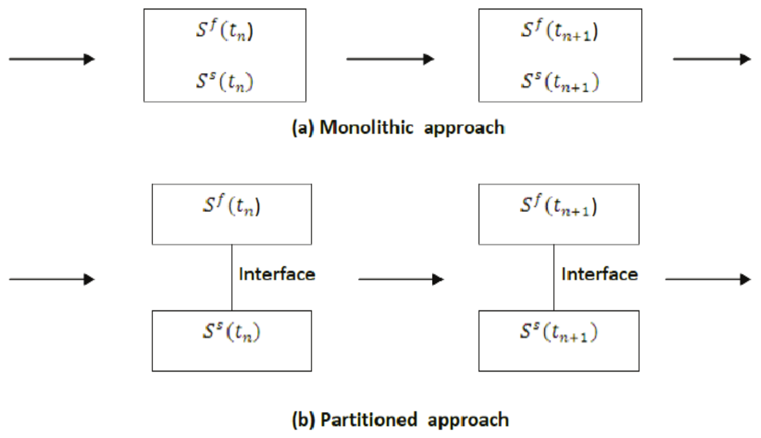
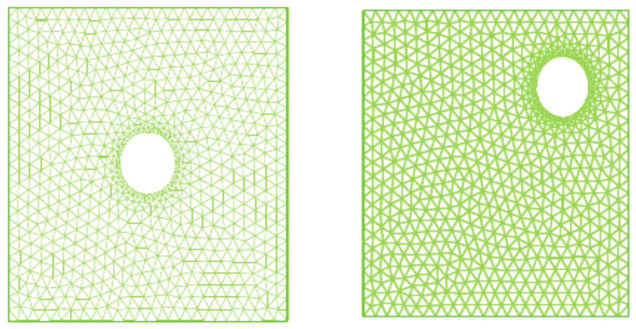
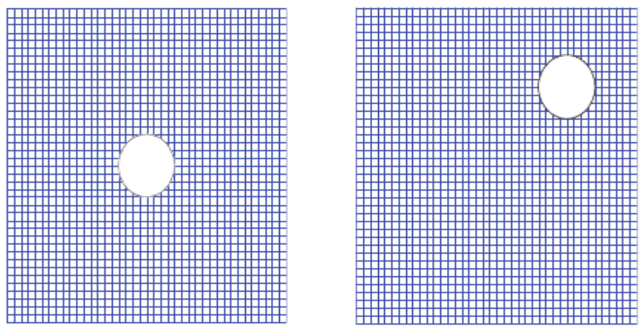


Figure I.59: Different types of numerical solution strategies for FSI problems (Figure taken from Hou et al. (2012)).

Based on Hou et al. (2012), numerical treatment of FSI problems can be broadly classified into two major approaches: the *monolithic* approach and *partitioned* approach, shown in Figure I.59. In the monolithic approach the interaction between solid and fluid at their interface is treated simultaneously, using a unified mathematical model which leads to a single system of equations following a unified discretization strategy. This approach may result a better accuracy for strong fluid-solid interactions, but it is computationally more expensive. In the partitioned approach, two separate mathematical models are used to describe fluid and solid domains. These two models are used to solve for fluid and solid separately, usually with two different discretization strategies, and they communicate through the common interface. The advantage of this approach is the possibility of using two separate disciplinary solution strategies, possibly two different legacy software to reduce the code development effort substantially. The challenge, however, for this approach is to integrate the two solvers and coordinate them to obtain an accurate and efficient solution with minimal code modification for each side. A better comparison between the monolithic and partitioned approach is provided by Michler et al. (2004).



(a) Conforming mesh. Left: $t = t_1$; Right: $t = t_2$.



(a) Non-conforming mesh. Left: $t = t_1$; Right: $t = t_2$.

Figure I.60: Conformal and non-conformal meshing strategies used in numerical solution of FSI problems (Figure taken from Hou et al. (2012)).

The position of interface between fluid and solid phases is a part of solution in an FSI problem. With regards to moving fluid-solid interface, two different strategies can be used: *conforming* mesh and *non-conforming* mesh, shown in Figure I.60. In the more traditional conformal mesh approach, the position and motion of the solid-fluid interface is captured sharply. Because of the movements of common interface, this method requires continuous re-meshing (or mesh updating) during the solution which can be computational expensive and error-prone. In the non-conformal meshes, also known as immersed-methods, the position of the interface is captured by applying some constraints to each solver to avoid the need for a sharp disjoint mesh and the mesh updates. This method is currently constitutes major research efforts in the computational FSI community.

I.2 Partitioned Approach: Strong vs. Weak Coupling

Following partitioned approach strategy, in order to solve for the coupled unknown structural and fluid quantities, two different algorithms can be applied. Figure I.61-a illustrates the more traditional weak coupling (also known as one-way coupling) algorithm. In this algorithm for each time step, the solution for the tractions obtained by the fluid solver is passed to the solid solver to find the new deformations. The output of the solid solver then will be used as an input to the fluid solver for the next time step. This approach is more efficient, but its application is limitted to FSI problems with weak fluid-solid interactions. For problems in which strong fluid-solid interactions are present, strong (two-way) coupling algorithm should be used. Figure I.61-b illustrates the outline of this algorithm with more details. The major difference for the strong-coupling algorithm is the application of an inner-loop within the external time stepping loop that helps to capture a converged fluid-solid interface solution in each time step. This method is computationally more expensive, but it guarantees a more accurate solution for problems in which strong fluid-solid interactions is expected.

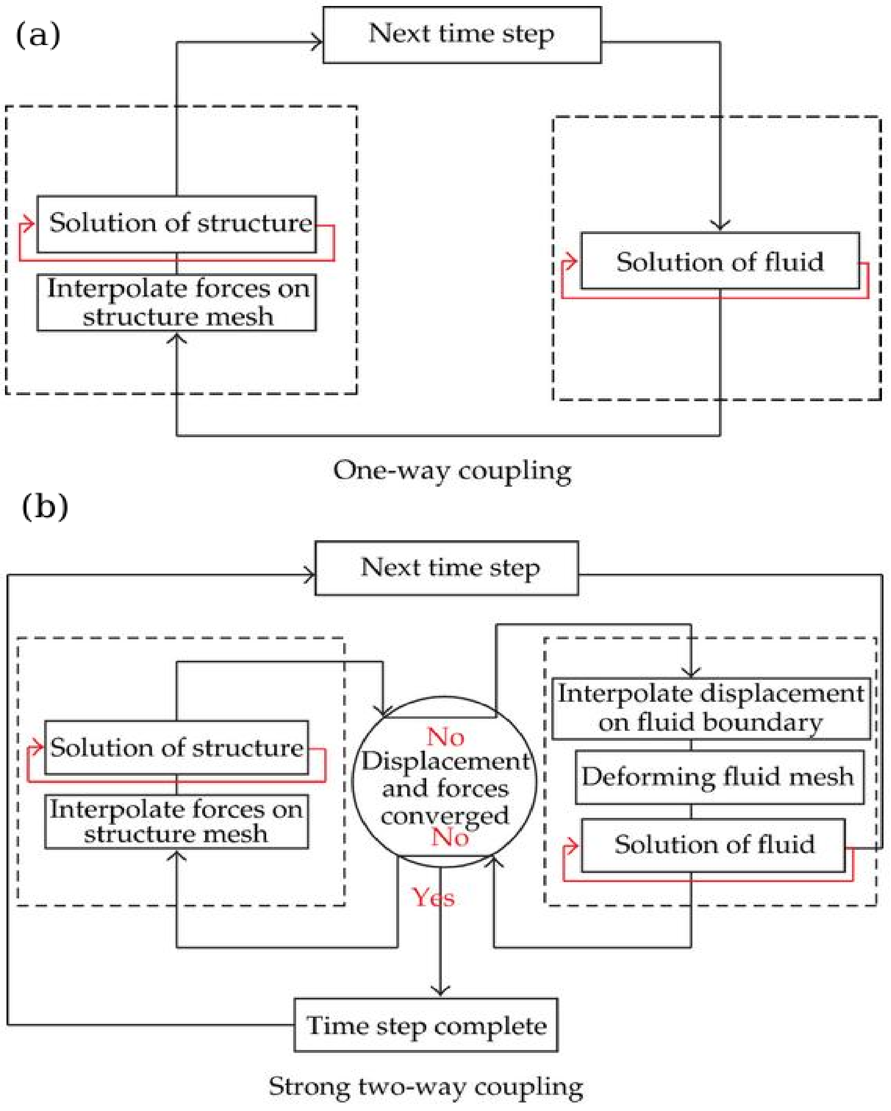


Figure I.61: Strong and weak coupling strategies used in numerical solution of FSI problems (Figure taken from Benra et al. (2011)).

I.3 ElmerFoamFSI: A Partitioned Approach

I.4 Overview

ElmerFoamFSI uses a partitioned approach to solve FSI problems. In *ElmerFoamFSI*, *OpenFOAM* (a finite-volume solver [*]) is used as the fluid solver and *Elmer* (a finite-element solver [†]) is used as the structural solver. In *ElmerFoamFSI* both weak and strong coupling approaches can be used to solve FSI problems. Currently a weak-coupling algorithm is implemented into *ElmerFoamFSI*. In the rest of this document, we briefly explain how to setup and use *ElmerFoamFSI* to solve a FSI problem.

I.5 *Elmer* Input

Elmer is an extensive open-source finite element multiphysics code. This software can be used to find deformation of a solid object (U) subject to different types of loads. *ElmerFoamFSI* uses *Elmer* as its structural solver. At least two different structural solvers (linear and non-linear elasticity) are provided by *Elmer*. To capture large deformations of the solid, *ElmerFoamFSI* uses the non-linear solver. The description of a finite element problem usually starts with input files. The full description of the problem, including: geometry, materials, boundaries conditions, type of equations to solve, etc., is usually provided in the input file(s).

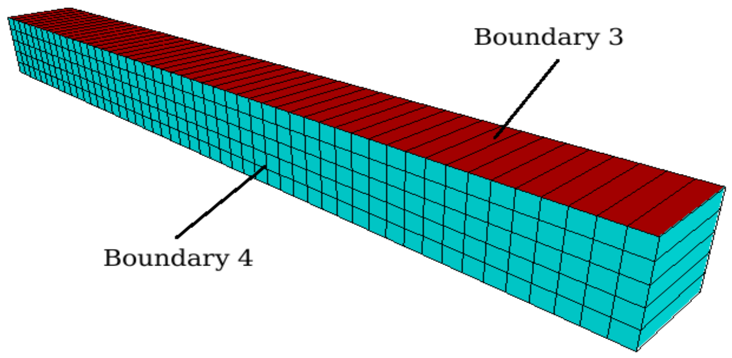


Figure I.62: Elmer mesh and boundary numbers.

Elmer uses a multiple-part input definition strategy. The description of the geometry of the problem in *Elmer* is provided in a GRD file (*".grd"* file), and the rest of the definitions is described in a SIF file (*".sif"* file). If the *ELMERSOLVER_STARTINFO* is used, upon execution *Elmer* scans this file to figure out which input file should be read (the name of the input file). Elmer uses a separate program (*ElmerGrid*) to read the GRD file and compile geometry description to a format useful for its internal usage. A proper description of problem geometry (also called problem mesh) requires a good level of understanding of finite element method, therefore it is beyond the purpose of the current document [‡]. The SIF file uses a *Keyword*-pair format to describe the rest of the problem. Each keyword can be thought of as a command to the *Elmer* solver which describes certain feature of the problem. Many of these commands require some parameters to be passed by user. These parameters are defined in front of the command after the "=" symbol. The first

[*] `http://www.extend-project.de/`
[†] `https://www.csc.fi/web/elmer/`
[‡] `http://www.nic.funet.fi/pub/sci/physics/elmer/doc/ElmerGridManual.pdf`

section of SIF file defines general settings of the problem as well as the location for geometry file, where to save simulation results, the type of simulation, time step used etc. This definition of this section of the SIF file is very straight forward as shown in the following :

```
Header
  CHECK KEYWORDS Warn
  Mesh DB "." "."
  Include Path ""
  Results Directory ""
End
Simulation
  Max Output Level = 5
  Coordinate System = Cartesian
  Coordinate Mapping(3) = 1 2 3
  Simulation Type = Transient
  Timestep intervals = 100
  Timestep Size = 1e-3
  Output Intervals = 1
  Timestepping Method = BDF
  BDF Order = 2
  Solver Input File = case.sif
  Post File = case.vtu
End
Constants
  Gravity(4) = 0 -1 0 9.82
  Stefan Boltzmann = 5.67e-08
  Permittivity of Vacuum = 8.8542e-12
  Boltzmann Constant = 1.3807e-23
  Unit Charge = 1.602e-19
End
```

In the next section, the types of equations to be solved and material properties for each constituent is described. Depending on what type of solver needed, a series of solver-related parameters should be passed to the *Elmer*. [*]. The following example defines a non-linear elasticity solver for *Elmer* along with the type of equation solver and non-linear solution strategies that should be used for the problem.

```
! Which equation to solve and which materials
Body 1
  Target Bodies(1) = 1
  Name = "Body 1"
  Equation = 1
  Material = 1
End
! Solver to use for equation 1
Equation 1
  Name = "Elasticity"
```

[*] http://www.nic.funet.fi/pub/sci/physics/elmer/doc/ElmerSolverManual.pdf

```
  Calculate Stresses = True
  Active Solvers(1) = 1
End
! Solver 1 definition and the solution strategies
Solver 1
  Exec Solver = Always
  Equation = Nonlinear elasticity
  Variable = Displacement
  Variable Dofs = 3
  Procedure = "ElasticSolve" "ElasticSolver"
  Nonlinear System Convergence Tolerance = 1.0e-4
  Nonlinear System Max Iterations = 1
  Nonlinear System Newton After Iterations = 3
  Nonlinear System Newton After Tolerance = 1.0e-2
  Nonlinear System Relaxation Factor = 1.0
  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Max Iterations = 1000
  Linear System Convergence Tolerance = 1.0e-6
  Linear System Abort Not Converged = True
  Linear System Residual Output = 1
  Steady State Convergence Tolerance = 1.0e-5
  Linear System Preconditioning = Diagonal
  Time Derivative Order = 2
End
! Definition of the material 1
Material 1
  Name = "blahBlah"
  Youngs modulus = 1.4e6
  Density = 10
  Poisson ratio = 0.4
End
```

The rest of SIF file defines the boundary conditions for the problem. For a structural problem different types of boundary conditions can be used, including displacement and force (traction). The boundary condition definition starts with a *Target Boundaries(x) = y* statement that specifies how many boundaries are using this description (x: an integer) and which boundary in the geometry file should be used (y: separated by space if more than one, look at Figure I.62). Finally we specify which direction our loads/displacement should be applied (here $1, 2, 3$ correspond to X, Y and Z directions in a Cartesian grid, respectively). Some of the boundary conditions require special treatments that are defined by some extra keywords. For example look at the boundary condition 4 that describes a FSI boundary. In this case, we tell *Elmer* to apply a variable force to the sections of the geometry that reside on this boundary. We also specify: 1) which direction loads should be applied (*direction 2* is *Force 2*), 2) they are variable in time, and 3) *Elmer* should resort to function *LoadYDirection* in *LoadFunctionLibrary* whenever it is needed. Care most be practiced with the proper formating. Further definition of the keywords can be found in the *Elmer* manuals.

```
Boundary Condition 1
  Target Boundaries(1) = 6
  Name = "Wall"
  Displacement 3 = 0
  Displacement 2 = 0
  Displacement 1 = 0
End
Boundary Condition 2
  Target Boundaries(1) = 2
  Name = "stabilizer1"
  Displacement 1 = 0
End
Boundary Condition 3
  Target Boundaries(1) = 4
  Name = "stabilizer2"
  Displacement 1 = 0
End
Boundary Condition 4
  Fsi BC = True
  Target Boundaries(1) = 3
  Name = "FSI"
  Force 2 = Variable Time
    Real Procedure "LoadFunctionLibrary" "LoadYDirection"
End
```

I.6 *OpenFOAM* Input

ElmerFoamFSI uses *OpenFOAM* as its fluid solver module to solve for the fluid pressure (p) and velocity (U). To capture the deformations of the solid/fluid interface, *OpenFOAM* has to also perform mesh update and re-meshing procedures in order to maintain the quality of the computational grid during the simulation (in a fluid problem, gird is equivalent to mesh for solid problems). *OpenFOAM* uses a finite-volume discretization scheme to obtain pressures and velocities and a Laplace solver to perform the re-meshing tasks. User has to specifies proper parameters for these solvers in the input files.

The definition of a problem in *OpenFOAM* is performed based on a multi-part input file strategy. For FSI problems, *OpenFOAM* requires both the definition of the solid and the fluid sections of the grid. In *ElmerFoamFSI*, however, the solid definition for *OpenFOAM* is disregarded and will be replaced with that of *Elmer*. Similarly, all other computations for the solid section will be performed solely by *Elmer*. To setup a fluid problem in *OpenFOAM*, we have to follow a strict folder/file hierarchy as *OpenFOAM* seeks the definition of a problem in multiple files residing in a set of subfolders. The input files in *OpenFOAM* are also called dictionary files. In each dictionary file, a set of keyword-value pairs specify proper settings for some part of the problem and solution procedure. There should be at-least three major sub-folders in the main fluid problem definition folder *./fluid/*. These sub-folders are *1)./fluid/0/, 2)./fluid/constant/ and 3)./fluid/system/*. In the rest we briefly describe each folder and its contents. Readers are strongly encouraged to resort to *OpenFOAM* documentation for more details.[*]

In *OpenFOAM*, *./fluid/0/* contains initial and boundary conditions for the problem. In this folder, we specify

[*]valuable information can also be found in `http://www.cfd-online.com/`

initial and boundary conditions for the pressures, velocities and grid deformations in three separate files with the proper names. Similarly, *./fluid/system/* folder contains a series of input files to specify which solvers should be used and what solution strategies should be used for them. Finally, *.fluid/constant/* folder contains the description of the grid and material properties. In this folder, a sub-folder exists which contains all information about the problem geometry, specified in *blockMeshDict*. These geometric information will be read by *blockMesh*, an accessory of *OpenFOAM* that generates simple block-shaped grids from the geometric data defined in the dictionary. The overall structure of an example blockMesh file is as following:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     3.0                                |
|   \\  /    A nd           | Web:         http://www.extend-project.de       |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
convertToMeters 1;
vertices
(
    (0.1 0.1 0.0)     // 0
    (0.1 0.1 1.0)     // 1
    (0.1 1.0 1.0)     // 2
    (0.1 1.0 0.0)     // 3

    (0.0 0.1 0.0)     // 4
    (0.0 0.1 1.0)     // 5
    (0.0 1.0 1.0)     // 6
    (0.0 1.0 0.0)     // 7
);
blocks
(
    hex (0 1 2 3 4 5 6 7) (50 50 1) simpleGrading (1 1 1)
);
edges
(
);
patches
(
    wall left
    (
        (0 3 7 4)
    )
    wall right
    (
        (1 5 6 2)
    )
    wall fsiFace
    (
        (0 4 5 1)
    )
    wall top
    (
        (3 2 6 7)
    )
    empty front
    (
        (4 7 6 5)
    )
    empty back
    (
        (0 1 2 3)
    )
);
mergePatchPairs
(
);
// ************************************************************************* //
```

The structure and the commands used in this file is fairly easily to understand.* After the *OpenFOAM* banner, at the top section of the file *blockMesh* accessory to be used and the format of the file is specified. In the *vertices* section, the coordinates for the vertices of the grid are specified (starting from 0, each line specifies a vertex coordinate). In the *block* section, the connectivity for the blocks of the mesh are specified. In the *patches* section, we specify boundaries of the domain and their connectivity with respect to the vertex indices.

I.7 *ElmerFoamFSI* Input

Currently, *ElmerFoamFSI* uses a very basic input file (named *"test.config"*). This file should reside in the *./fluid/* folder where the *ElmerFoamFSI* executable is called, and it should be passed to the executable as a command-line parameter. Following example shows the structure of this keyword-value pair file. In this example the name of fluid and structure solver modules that should be loaded are specified in the first two lines. The third line specifies the solution transfer module (in this case *SurfX* which a part of *IMPACT*). In the rest of the file the total analysis time and the time step are specified.

```
FluidSolver=OpenFoamFSI
SolidSolver=ElmerCSC
```

*for more details resort to `http://cfd.direct/openfoam/user-guide/`

```
TransferService=SurfX
FinalTime=0.05
TimeStep=2.0e-3
```

I.8 *ElmerFoamFSI* Call Procedure

To execute *ElmerFoamFSI* simulation, *Elmer* input files and *ElmerFoamFSI* input file should be copied to the *./fluid/* folder of the *OpenFOAM* input hierarchy. Following directions specify steps to take to run a problem:

- Create a folder and copy *Allclean*, *Allrun* (scripts), */fluid/* and */solid/* folders into this folder, make sure to copy all sub-folders as well.
- Copy elmer SIF, GRD and ELMERSOLVER_STARTINFO files to */fluid/*.
- run ./Allclean and ./Allrun to clean-up and prepare input files for the *OpenFOAM*.
- Go to *./fluid/* and type *ElmerGrid 1 2 name* where "name" should be replaced with the name of the GRD file for *Elmer*.
- Run *ElmerFoamFSI* the FSI simulation by "*elmerformfsi name*" where *name* should be replaced with the name of *ElmerFoamFSI* input file.

References

Allan, B., Armstrong, R., Wolfe, A., Ray, J., and Bernholdt, D. (2002). The *CCA* core specification in a distributed memory spmd framework. *Concurr. Comput. Practices Exp.*, 5:323–345.

Allen, G., Dramlitsch, T., Foster, I., Karonis, N., Ripeanu, M., Seidel, E., and Toonen, B. (2001). Supporting efficient execution in heterogeneous distributed computing environments with *Cactus* and *Globus*. In *Proc. Supercomput.*

Arienti, M., Hung, P., Morano, E., and Shepherd, J. E. (2003). A level set approach to eulerian-lagrangian coupling. *J. Comput. Phys.*, 185:213–251.

Bartlett, R. A., Heroux, M. A., and Willenbring, J. M. (2012). Tribits lifecycle model. Technical report, Sandia National Laboratory.

Bassetti, F., Brown, D., Davis, K., Henshaw, W., and Quinlan, D. (1998). *Overture*: an object-oriented infrastructure for high performance scientific computing. In *Proc. Supercomput.*

Bellavia, S., Bertaccini, D., and Morini, B. (2011). Nonsymmetric preconditioner updates in Newton–Krylov methods of nonlinear systems. *J. Sci. Comput.*, 33:2595–2619.

Benra, F.-K., Dohmen, H. J., Pei, J., Schuster, S., and Wan, B. (2011). A comparison of one-way and two-way coupling methods for numerical analysis of fluid-structure interactions. *Journal of Applied Mathematics*, 2011:16.

Budge, K. G. and Peery, J. S. (1998). Experiences developing *ALEGRA*: a C++ coupled physics framework. In *Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*.

Chisholm, T. T. and Zingg, D. W. (2009). A Jacobian-free Newton–Krylov algorithm for compressible turbulent fluid flows. *J. Comput. Phys.*, 228:3490–3507.

Council on Competitiveness (2011). Council on Competitiveness, 2010–2011 Annual Report. Technical report.

Dick, W. A., Fiedler, R. A., and Heath, M. T. (2006). Building *Rocstar*: Simulation science for solid propellant rocket motors. In *Proc. AIAA*, volume 4590, pages 1–10.

Elmer (2016). *Elmer*. https://www.csc.fi/web/elmer.

Gerstenberger, A. and Wall, W. A. (2008). An eXtended Finite Element Method/Lagrange multiplier based approach for fluid–structure interaction. *Comput. Methods Appl. Mech. Engrg.*, 197:1699–1714.

Geubelle, P. H., Jiao, X., Haselbacher, A. C., and Campbell, M. T. (2004). Numerical Coupling Interface in *Rocstar – Rocman* (GEN 3) Design Document. PDF.

Heroux, M. A., 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., Williams, A., and Stanley, K. (2005). An overview of the *Trilinos* projects. *ACM Trans. Math. Software*, 31:397–423.

Hou, G., Wang, J., and Layton, A. (2012). Numerical methods for fluid-structure interaction – a review. *Communications in Computational Physics*, 12:337–377.

Jaiman, R. K., Jiao, X., Geubelle, P. H., and Loth, E. (2004). Assessment of conservative load transfer for fluid-solid interface with nonmatching meshes. *International Journal for Numerical Methods in Engineering*, pages 1–40.

Jaiman, R. K., Jiao, X., Geubelle, P. H., and Loth, E. (2005). Assessment of conservative load transfer for fluid-solid interface with nonmatching meshes. *Intl. J. Numer. Methods Eng.*, pages 1–40.

Jiao, X. (2007). Face offsetting: A unified approach for explicit moving interfaces. *Journal of Computational Physics*, 220:612–625.

Jiao, X. and Heath, M. T. (2005). Common-refinement-based data transfer between non-matching meshes in multiphysics simulations. *International Journal for Numerical Methods in Engineering*, 61:2402–2427.

Jiao, X., Zheng, G., Alexander, P. J., Campbell, M. T., Lawlor, O. S., Norris, J., Hasselbacher, A., and Heath, M. T. (2006). A system integration infrastructure for coupled multiphysics simulations. *Engineering with Computers*, 22:293–309.

Kamakoti, R. and Shyy, W. (2004). Fluid-structure interaction for aeroelastic applications. *Prog. Aerosp. Sci.*, 40:535–558.

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., Rivière, B., Rüde, U., Scheibe, T., Shadid, J., Sheehan, B., Shephard, M., Siegel, A., Smith, B., Tang, X., Wilson, C., and Wohlmuth, B. (2013). Multiphysics simulations: Challenges and opportunities. *Intl. J. High Perform. Comput. Appl.*, 27:5.

Knoll, D. A. and Keyes, D. E. (2004). Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *J. Comput. Phys.*, 193:357–397.

M. A. Heroux et al. (2012). *Trilinos*. http://trilinos.sandia.gov.

Michler, C., Hulshoff, S., van Brummelen, E., and de Borst, R. (2004). A monolithic approach to fluid-structure interaction. *Computers and Fluids*, 33(5-6):839–848. Applied Mathematics for Industrial Flow Problems.

OpenFOAM Extend Project (2016). *OpenFOAM Extend Project*. http://www.extend-project.de.

Pawlowski, R., Bartlett, R., Belcourt, N., Hooper, R., and Schmidt, R. (2011). A theory manual for multi-physics code coupling in LIME. Technical report, Sandia National Laboratory.

Poppendieck, M. and Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley.

Reynders, J. V. W., Hinker, P. J., Cummings, J. C., Atlas, S. R., Banerjee, S., Humphrey, W. F., Karmesin, S. R., Keahey, K., Srikant, M., and Tholburn, M. (1996). *POOMA*: A infrastructure for scientific simulations on parallel architectures. In Wilson, G. V. and Lu, P., editors, *Parallel programming using C++*, pages 547–588. Massachusetts Institute of Technology.

Schafer, M. and Turek, S. (1996). Flow simulation with high-performance computers ii. dfg priority research program results 1993-1995. In Hirschel, E., editor, *Notes in Numerical Fluid Mechanics*, number 52, pages 547–566. Vieweg Weisbaden.

Schieffer, G., Ray, S., Bramkamp, F. D., Behr, M., and Ballmann, J. (2010). An adaptive implicit finite volume scheme for compressible turbulent flows about elastic configurations. In Schröder, W., editor, *Flow Modulation & Fluid-Structure Interaction*, volume 109, pages 25–51. Springer-Verlag.

Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D. (2013). Cfd vision 2030 study: A path to revolutionary computational aerosciences. Technical report, NASA Langley Research Center, Hampton, Virginia 23681-2199.

Stewart, J. R. and Edwards, H. C. (2004). A infrastructure approach for developing parallel adaptive multiphysics applications. *Finite Elem. Anal. Des.*, 40:1599–1617.

Turek, S. and Hron, J. (2006). *Fluid-Structure Interaction: Modelling, Simulation, Optimisation*, chapter Proposal for Numerical Benchmarking of Fluid-Structure Interaction between an Elastic Object and Laminar Incompressible Flow, pages 371–385. Springer Berlin Heidelberg, Berlin, Heidelberg.

J Mathematics of FSI Coupling

Mathematics of Coupling
Version 0.1.0 IllinoisRocstar LLC
October 25, 2016

License

J.1 Interaction of Fluids and Structures – An Introduction by Keyes

Numerical simulations that model the interactions between incompressible laminar flows and elastic structures require coupling a description of the fluids with a description of the structures. The unknowns of the involved equations (velocities and pressure for the fluid, displacements for the structure) are associated with different locations in the overall computational domain, resulting in a surface-coupled problem.

FSI can be simulated in at least two ways. One approach is to solve a large system of equations for all fluid and structure unknowns as a single system – typically ill-conditioned. Alternatively, in a partitioned approach, one has separate solvers for the fluid and the structure, together with a suitable coupling method. In the latter case, boundary conditions for both single-physics problems at the coupling surface have to be defined. The respective interface values are passed from one solver to the other. This approach requires mapping methods for physical variables between (in general) non-matching solver grids at coupling surfaces. Important features desired of such mappings are accuracy, consistency, and conservation of energy and momentum. Two main classes of mapping methods can be identified: interpolation methods, based on geometric relations between the involved grid points; and mortar methods, in which boundary conditions are formulated in weak form by using Lagrange multipliers.

Quantities of Interest

- Fluid velocities and pressures (loads)
- Structural displacements

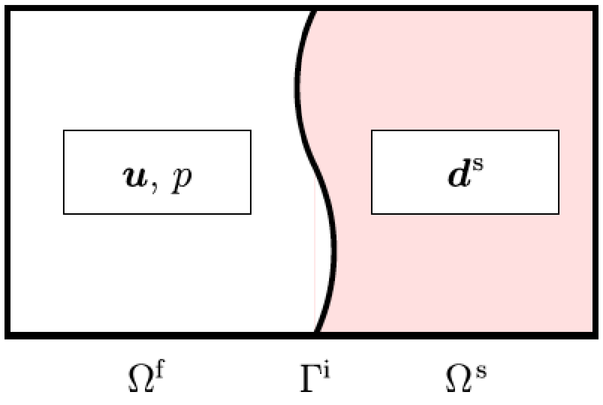


Figure J.63: Problem setup: Fluid field Ω_{f}, structural field Ω_{s}, and the conjoined interface Γ_{i}. Here $\vec{u}$ is the fluid velocity, p is the pressure, and $\vec{d}^{\mathrm{s}}$ are the structure displacements. Figure taken from [Gerstenberger and Wall (2008)].

J.1.1 Comments on a Few Common Coupling Approaches

The coupling itself can be done with different methods, leading to more loosely or more tightly coupled timestepping methods. The loosest coupling is a one-way coupling, where the flow solver computes a force exerted on the structure using a rigid-structure geometry, and structural movements are computed in a postprocessing-like manner based on these forces. This strategy is obviously applicable only for small and static structure deformations. The most widely used class of iteration schemes is Gauss–Seidel-like coupling iterations, with variants ranging from a single iteration loop per timestep to repeated iteration-to-convergence within each timestep, with or without (Aitken) underrelaxation, to interface quasi-Newton methods that efficiently compute approximate Newton iterations based on sensitivities resulting from Gauss–Seidel iterations. To account for what are usually moderately different timescales in the fluid and the structure, a subcycling in the flow solver can be used. In the absence of turbulence, spatial scales are essentially the same throughout fluid and structure domains.

The choice of coupling implementation can lead to development of numerical instabilities in multiphysics codes. For incompressible fluids, the so-called “added-mass” effect induces instabilities in loosely coupled simulations and in Gauss–Seidel-like iterations. The force exerted by the fluid on a moving structure can be interpreted as a virtual added mass of the structure. For an incompressible flow, each acceleration or deceleration of the structure causes an immediate change in this added mass (whereas the added-mass change for a compressible flow increases continuously over time). If this change is too large, both loosely and tightly coupled Gauss–Seidel-like coupling schemes become unconditionally unstable; in other words, a reduction of the timestep does not cure the instability. In the case of a massless structure, a reduction of the timestep even worsens instabilities. Typically, only a few low-frequency Fourier modes of interface displacements or velocities are unstable. Interface quasi-Newton methods rapidly capture the unstable Fourier modes and thus lead to stable coupling iterations. However, standard multigrid techniques, with Gauss–Seidel-like iterations as a smoother, do not work for incompressible flows. Although they smooth high frequencies, they are unstable for low frequencies. In addition, the convergence rate for low-frequency modes does not improve on spatially coarser grids.

Since Gauss–Seidel-like coupling iterations, including the interface quasi-Newton method, are inherently sequential in execution order of flow and structure solver, and since the structure solver is usually much less costly than is the flow solver, alternative Jacobi-like methods are desirable on massively parallel systems. These avoid processor idle time during the execution of the structural solver and the coupling numerics.

Many of these problems feature multiple spatial and temporal scales. In all of them, the fluid domain can be occupied by one or multiple interacting fluids, the structure can be linear or nonlinear with self-contact, and the FSI occurs at physical interfaces.

J.1.2 Example Multiphysics FSI Software Package – *AERO*

As a practical application of FSI, we consider examples in high performance, high fidelity analysis, which finds application in modeling the flow of airstreams over aircraft and vehicles as well as fluid flow over underwater structures. Airflow problems necessitate the solution of nonlinear compressible FSI problems. Specific examples include the parametric identification of an F-16 Block-40 fighter aircraft in clean wing configuration and subsonic, transonic, and supersonic airstreams; the aeroelastic analysis of an F/A-18 5.6 Hz limit-cycle oscillation configuration; the flutter clearance of the laminar flow wing of a supersonic business jet concept; and the aeroelastic tailoring of a Formula 1 car. Examples of incompressible FSI problems include the study of intense implosive collapses of gas-filled underwater structures and their effects on nearby structures.

The *AERO* code is an example of a multiphysics FSI software package that considers these issues. The functional modules of the *AERO* code include some of the most important considerations for producing accurate models:

1. The structural analyzer AERO-S

2. The compressible turbulent flow solver AERO-F
3. The auxiliary module MATCHER that enables the discretization of fluid–structure transmission conditions at nonmatching, discrete fluid–structure interfaces
4. The auxiliary module, SOWER that manages all parallel input/output (I/O) associated with this software

AERO-F can operate on unstructured body-fitted meshes as well as fixed meshes that embed discrete representations of surfaces of obstacles, around and/or within which the flow is to be computed. The body-fitted meshes and the embedded discrete surfaces can be fixed, move, and/or deform in a prescribed manner, or can be driven by interaction with the structural analyzer AERO-S.

In the case of body-fitted meshes, the governing equations of fluid motion are formulated in the arbitrary Lagrangian–Eulerian (ALE) framework. In this case, large mesh motions are handled by a corotational approach, which separates the rigid and deformational components of the motion of the surface of the obstacle, and robust mesh motion algorithms that are based on structural analogies. In the case of embedded surfaces that can have complex shapes and arbitrary thicknesses, the governing equations of fluid motion are formulated in the Eulerian framework, and the wall boundary or transmission conditions are treated by an embedded boundary method.

Both AERO-F and AERO-S feature explicit and implicit time integrators with adaptive timestepping. Both modules are coupled by a family of partitioned analysis procedures that are loosely coupled but exhibit excellent numerical stability properties and are provably second-order time-accurate. AERO-F and AERO-S communicate via runtime software channels, for example, using MPI. They exchange aerodynamic (pressure) and elastodynamic (displacement and velocity) data across non-matching, discrete fluid and structure mesh interfaces using a conservative method for the discretization of transmission conditions and the data structures generated for this purpose by MATCHER.

[Keyes et al. (2013)]

J.2 Computational Aeroelastic Models

Computational aeroelasticity (CAE) can be classified broadly under three major categories: fully coupled, closely coupled, and loosely coupled analyses. Before looking at the various CAE models in detail, it is useful to look at the generalized equations of motion to better explain CAE methodologies.

$$[M]\{\ddot{q}(t)\}+[C]\{\dot{q}(t)\}+[K]\{q(t)\}=\{F(t)\}, \tag{J.7}$$

$$\{w(x,y,z,t)\}=\sum_{i=1}^{N} q_i(t)\{\phi_i(x,y,z)\}. \tag{J.8}$$

Here, $\{w(x,y,z,t)\}$ is the structural displacement at any time instance and position, and $\{q(t)\}$ is the generalized displacement vector. The matrices $[M]$, $[C]$, $[K]$ are the generalized mass, damping, and stiffness matrices, respectively, and ϕ_i are the normal modes of the structure, with N being the total number of modes of the structure.

The term on the rhs of Eq J.7, $\{F(t)\}$, is the generalized force vector, which is responsible for linking the unsteady aerodynamics and inertial loads with the structural dynamics. Eq J.7 shows that there are distinct terms representing the structures, aerodynamics, and dynamics disciplines, which gives us the flexibility in choosing different methods for any particular system.

In *fully coupled models*, the governing equations are reformulated by combining fluid and structural equations of motion, which are then solved and integrated in time simultaneously. While using a fully coupled procedure, one must deal with fluid equations in an Eulerian reference system, and structural equations in a Lagrangian system. This leads to the matrices being orders of magnitude stiffer for structure systems as compared to fluid systems, thereby making it virtually impossible to solve the equations using a monolithic computational scheme for large scale problems.

In *loosely coupled models*, unlike the fully coupled analysis, the structural and fluid equations are solved using two separate solvers. This can result in two different computational grids (structured or unstructured), which are not likely to coincide at the boundary. This calls for an interfacing technique to be developed, to exchange information back and forth between the two modules. The loosely coupled approach has only external interaction between the fluid and structure modules; or the information is exchanged after partial or complete convergence. This approach is like a multidisciplinary computing environment, where one effectively controls the interaction between two commercial codes for each of the modules by means of interfacing techniques. This gives us the flexibility of choosing different solvers for each of the modules, but the coupling procedure leads to a loss in accuracy as the modules are updated only after partial or complete convergence. A typical block diagram comprising of different solvers for fluid and structure models as well as the interfacing methodologies is shown in Figure J.64. This kind of a loosely coupled approach is limited to small perturbations and problems with moderate nonlinearity.

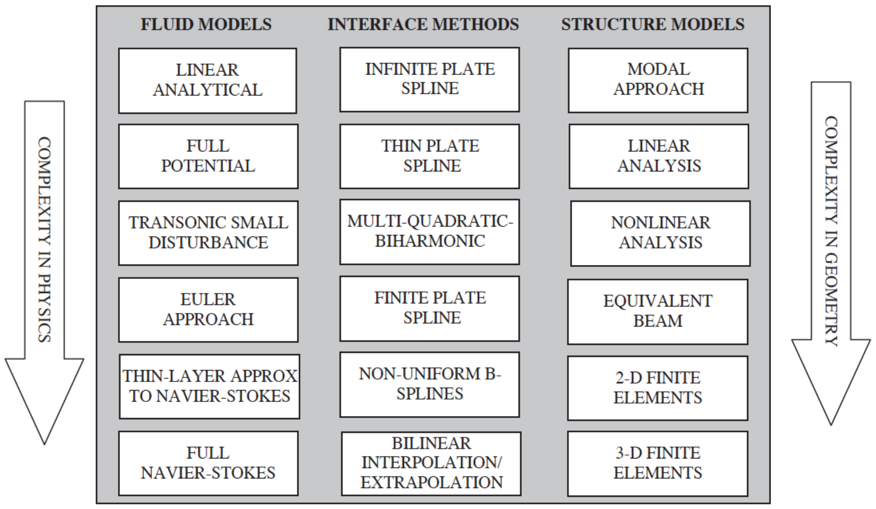


Figure J.64: Sample fluid and structure solvers along with select interfacing methodologies for aeroelastic simulation. Figure taken from [Kamakoti and Shyy (2004)].

Closely coupled models are some of the most widely used methods in the field of CAE as it not only paves the way for the use of different solvers for fluid and structure models but also couples the solvers in a tight fashion thereby making it an efficient method for complex nonlinear problems. In this approach, the fluid and structure equations are solved separately using different solvers but are coupled into one single module with exchange of information taking place at the interface or the boundary via an interface module thereby making the entire CAE model tightly coupled. The information exchanged here are the surface loads, which are mapped from CFD surface grid onto the structure dynamics grid; and the displacement field, which are mapped from structure dynamics grid onto CFD surface grid. The transfer of surface displacement back to the CFD module implies deformation of the CFD surface mesh, and this calls for a moving boundary technique to enable remeshing of the entire CFD domain as we march in time. This can cause potential problems for multiblock grids with complex geometries.

For descriptions of fluids and structures solvers, including the governing equations and overview of different solution algorithms, see [Kamakoti and Shyy (2004)].

J.3 Keyes' Prototype Algebraic Forms and Nomenclature

The two simplest systems that exhibit the crux of a multiphysics problem are the coupled equilibrium problem,

$$\begin{aligned} F_1(u_1,u_2) &= 0 \\ F_2(u_1,u_2) &= 0 \end{aligned} \tag{J.9}$$

and the coupled evolution problem,

$$\begin{aligned} \partial_t u_1 &= f_1(u_1,u_2) \\ \partial_t u_2 &= f_2(u_1,u_2). \end{aligned} \tag{J.10}$$

where $\partial_t u_i$ represents the time derivative of u_i, i.e., $\partial u_i/\partial t$. Here u refers generically to a multiphysics solution, which has multiple components indicated by subscripts $u = (u_1,\ldots,u_{N_c})$; the simplest case of $N_c = 2$ components is indicated here.

Noting this, we will generically use the notation in eq (J.9) to refer to either a coupled equilibrium problem or a single timestep of a coupled evolution problem.

When eq (J.10) is semi-discretized in time, the evolution problem leads to a set of problems that take the form of eq (J.9) and are solved sequentially to obtain values of the solution $u(t_n)$ at a set of discrete times.

We assume initially, for convenience, that the Jacobian,

$$J = \frac{\partial(F_1,F_2)}{\partial(u_1,u_2)}, \tag{J.11}$$

is diagonally dominant in some sense and that the partial derivatives, $\partial F_1/\partial u_1$ and $\partial F_2/\partial u_2$, are nonsingular. These assumptions are natural in the case where the system arises from the coupling of two individually well-posed systems with legacies of being solved separately.

In the equilibrium problem, eq (J.9), we refer to F_1 and F_2 as the component *residuals*; in the evolution problem, eq (J.10), we refer to f_1 and f_2 as the component *tendencies*.

The choice of solution approach for these coupled systems relies on a number of considerations. From a practical standpoint, existing codes for component solutions often motivate operator splitting as an expeditious route to a first multiphysics simulation capability making use of the separate components. This approach, however, may ignore strong coupling between components and give a false sense of completion. Solution approaches ensuring a tight coupling between components require smoothness, or continuity, of the nonlinear, problem-defining functions, F_i, and their derivatives. Any potential discontinuities must be identified and addressed before putting confidence in these approaches.

Classic multiphysics algorithms preserve the integrity of the two uniphysics problems, namely, solving the first equation for the first unknown, given the second unknown, and the second equation for the second unknown, given the first. This represents the reductionist approach of science, and software generally exists to do this. Multiphysics coupling is taken into account by iteration over the pair of problems, typically in a Gauss–Seidel manner (see Algorithm 4), linear or nonlinear, according to context. Here we employ superscripts to denote iterates.

Procedure 4 Gauss–Seidel multiphysics coupling.

```
Given: Initial iterate $\{u_1^0, u_2^0\}$
for $k = 1, 2, \ldots, (until\ convergence)$ do
    Solve for $v$ in $F_1(v, u_2^{k-1})$ ; set $u_1^k = v$   Solve for $w$ in $F_2(u_1^k, w)$ ; set $u_2^k = w$
end
```

When this iteration converges, the accuracy with which the discrete equations are solved can be improved by continuing the iterations. The largest implicit aggregate is the largest of the uniphysics problems; we refer to this iteration as "loosely coupled." A Jacobi-like iteration can be similarly defined. This further decoupling exposes more parallelism, albeit possibly at the cost of a slower convergence rate.

The simplest approach to the evolutionary problem likewise employs a field-by-field approach in a way that leaves a first-order-in-time splitting error in the solution. Algorithm 5 gives a high-level description of this process, which produces solution values at time nodes $t_0 < t_1 < \ldots < t_N$. Here, we use notation $u(t_0), \ldots, u(t_N)$ to denote discrete timesteps. An alternative that staggers solution values in time is also possible.

Procedure 5 Multiphysics operator splitting.

```
Given: Initial values $\{u_1(t_0), u_2(t_0)\}$
for $n = 1, 2, \ldots, N$ do
    Evolve one timestep in $\partial_t u_1 + f_1(u_1, u_2(t_{n-1})) = 0$ to obtain $u_1(t_n)$   Evolve one timestep in $\partial_t u_2 + f_2(u_1(t_n), u_2) = 0$ to obtain $u_2(t_n)$
end
```

The individual component evolutions in Algorithm 5 can be implicit or explicit and performed with or without subcycles. However, there is no point in their being of high order unless a higher-order coupling scheme than this is used, such as Strang splitting or temporal Richardson extrapolations for higher order.

An inner loop may be placed inside each timestep in which the coupling variables are updated to satisfy implicit consistency downstream while still preserving the legacy software for each component.

For example, this loosely coupled evolution is depicted in the context of aeroelasticity, where the first component consists of fluid velocity and pressure and the second component of structural displacements. This is an interface transmission form of coupling in which structural displacements provide boundary conditions for the fluid, and fluid pressures provide boundary conditions for the structure in the context of a dynamic mesh.

If the residuals or tendencies and their derivatives are sufficiently smooth and if one is willing to write a small amount of solver code that goes beyond the legacy component codes, a good algorithm for both the equilibrium problem and the implicitly time-discretized evolution problem is Jacobian-free Newton–Krylov (JFNK). Here, the problem is formulated in terms of a single residual that includes all components in the problems,

$$F(u) \equiv \begin{pmatrix} F_1(u_1, u_2) \\ F_2(u_1, u_2) \end{pmatrix} = 0 \tag{J.12}$$

where $u = (u_1, u_2)$. The basic form of Newton's method to solve eq (J.12), for either equilibrium or transient problems, is given by Algorithm 6. Because of the inclusion of the off-diagonal blocks in the Jacobian, for example,

$$J = \begin{bmatrix} \frac{\partial F_1}{\partial u_1} & \frac{\partial F_1}{\partial u_2} \\ \frac{\partial F_2}{\partial u_1} & \frac{\partial F_2}{\partial u_2} \end{bmatrix}, \tag{J.13}$$

Newton's method is regarded as being "tightly coupled."

Procedure 6 Newton's method.

```
Given: Initial iterate $u^0$
for $k = 1, 2, \ldots, (until\ convergence)$ do
    Solve $J(u^{k-1})\delta u = -F(u^{k-1})$   Update $u^k = u^{k-1} + \delta u$
end
```

The operator and algebraic framework described here is relevant to many divide-and-conquer strategies in that it does not "care" (except in the critical matter of devising preconditioners and nonlinear component solvers for good convergence) whether the coupled subproblems are from different equations defined over a common domain, the same equations over different subdomains, or different equations over different subdomains. The general approach involves iterative corrections within subspaces of the global problem. All the methods have in common an amenability to exploiting a "black-box" solver philosophy that amortizes existing software for individual physics components. The differences are primarily in the nesting and ordering of loops and the introduction of certain low-cost auxiliary operations that transcend the subspaces.

Not all multiphysics problems can easily or reliably be cast into these equilibrium or evolution frameworks, which are primarily useful for deterministic problems with smooth operators for linearization. In formulating multiphysics problems, modelers first apply asymptotics to triangularize or even to diagonalize the underlying Jacobian as much as possible, pruning provably insignificant dependencies, but bearing in mind the conservative rule of coupling: "guilty until proven innocent." One then applies multiscale analyses to simplify further, eliminating stiffness from irrelevant mechanisms. Because of significant timescale and resolution requirement differences, these activities often reveal justifications for splitting some physics or models from others. In these cases, an operator-splitting approach should be applied. However, caution is warranted to ensure that temporal or spatial scales do not overlap between split physics as the simulation progresses. If this happens, a more coupled approach is generally required [Keyes et al. (2013)].

J.4 Comments on Transfer Functions

There now exists an extra set of requirements for transferring the parameter information between codes. These transfer functions can be simple postprocessing that uses solution and parameter values from the other applications, or it can be something as complex as a complete nonlinear solve.

Strategies on how to aggregate the evaluation of the transfer operators are usually dictated by software design constraints, data transfer requirements for the multiple code couplings supported, and efficiency considerations.

In terms of software implementation, there are many ways to handle transfers of information between codes. The transfer functions are only a mathematical concept that explicitly demonstrates the transfers. Software implementations could implement this directly.

An alternative and equally valid way to transfer information is to directly write the transfer functions into the residual operator F. We typically do this when the code already does the postprocessing during the solution step.

Yet another way to implement transfers is to treat the transfer operator as an entirely new single physics application by implementing an additional F. In this case, simple transfer mappings are used to copy the data between residual functions.

In our experience, we usually reserve the transfer operators for simple mappings, and for anything that requires a linear or nonlinear solver, we implement the transfer as an additional physics application F.

In practice, the sets for a particular transfer function are very sparse. They usually take the solution and parameters from one physics application and generate values for another physics application.

The union of all transfer operators defines an implicit dependency graph between the coupled physics applications. Explicitly exposing this information can help determine nonzero sensitivity blocks for implicit solution techniques [Pawlowski et al. (2011)].

J.5 Discussion of Some FSI Coupling Strategies

In interfacial coupling we have two or more physical domains each containing different physical processes, but which share a common interfacial surface. The physical processes are independent in each domain apart from the interaction at the interface.

One example is the modeling of a re-entry vehicle as it travels through the atmosphere. The flight of the vehicle through the atmosphere creates a pressure load on the shell of the vehicle which in turn affects the structural dynamics of the interior of the vehicle. Here the two domains are the fluid exterior to the vehicle (compressible, turbulent fluid flow) and the interior of the vehicle (structural dynamics) coupled through the shell of the vehicle (interface).

In systems of this type the physics in each domain is typically modeled as a set of partial differential equations that are coupled through boundary conditions. Although the applications share a common interfacial surface, the interface could have different spatial discretizations for each. Here the transfer functions represent the interfaces between fluid and structure systems and typically map their argument to a much lower dimensional space

This section describes the basic algorithms used to solve a general multiphysics model, assuming that it is composed of nonlinear equations. The goal is not to present an exhaustive list of solution strategies, but to describe the typical requirements that a physics code must support for a particular solution method. The information in this section should be used to help determine the best coupling strategy for a particular set of applications or to select the most practical approach given current legacy software constraints.

The equations that compose the general model can be generated from any number of sources. For example, it could be a set of ordinary differential equations (ODEs), a discretized set of partial differential equations (PDEs), a set of algebraic equations, or a set of differential algebraic equations (DAEs).

J.5.1 Picard Iteration

The first solution method we mention is a stationary iterative method whose variants are known by multiple names: Picard iteration, nonlinear Richardson iteration, successive substitution or fixed-point iteration. Here we do not discuss differences among these variants and will refer to them collectively as Picard iteration methods.

Picard iteration, also known as successive substitution, is a simple numerical method for approximating the solution to a fully coupled system. It works by solving each component in the coupled system for its solution variables, treating the other variables as fixed quantities. This is repeated in a round-robin fashion until some measure of convergence is achieved (typically when the size of the change is small for the solution variables from iteration to iteration, and/or the size of the residual of each component evaluated at the most current solution values). For example, an algorithm for applying this technique to the two-component interfacially coupled system is displayed in Algorithm 7, where $r_{i,j}$ is the transfer function between components i and j.

Procedure 7 Picard iteration for the two-component interfacially coupled system.

```
Require: Initial guesses $u_1^0$ and $u_2^0$ for $u_1$ and $u_2$
$k = 0$  while not converged do
    $k = k + 1$   Solve $F_1(u_1^k, r_{1,1}(u_2^{k-1})) = 0$ for $u_1^k$   Solve $F_2(u_2^k, r_{2,1}(u_1^k)) = 0$ for $u_2^k$
end
```

The problem with this approach is that the convergence rate is slow (linear) and needs additional requirements on F_1 and F_2 to converge. Picard methods are attractive when the different components are only weakly coupled or when there is a very good initial guess and/or only an approximate solution is needed. For example, a Picard method would tend to work well to solve for the implicit timestep update in a transient predictor/corrector method (where the explicit predictor gives a good initial guess and the implicit corrector system only needs to be converged a little to guarantee stability).

J.5.2 Newton Methods (Review)

The primary alternative to the Picard iteration class of methods are various methods based on Newton's method. The basic algorithm for a Newton method is described by Algorithm 6, where $J(u^k)$ is the Jacobian matrix, $J = \partial F / \partial u$, evaluated using the state variables u^k at iterate k.

The advantages of Newton methods include a q-quadratic convergence rate in the error norm and the stability of the algorithm. Although robustness issues can arise, these can be addressed by leveraging globalization techniques such as linesearch and trust region methods. The primary drawback to Newton's method is the cost and difficulty of computing the full Jacobian matrix. The diagonal blocks are typically non-singular matrices (but may not always be), but the non-diagonal blocks are typically rectangular matrices. Note that even if each individual application were to implement a Newton-based solve, this would only supply the Jacobian diagonal blocks. The off-diagonal sensitivities provided by the transfer functions would still be missing.

One way to address the off-diagonal blocks is to leverage an approximate Newton-based method called the Newton–Krylov approach. When using Newton–Krylov solvers, we avoid the cost of constructing an explicit Jacobian. Newton–Krylov solvers build up an approximation to the solution of the Newton system by applying Jacobian-vector products to construct a Krylov subspace. By requiring only the Jacobian-vector products to solve the Newton system, the Jacobian need not be explicitly formed. While an explicit Jacobian matrix could be used, the Jacobian-vector product can be computed to machine precision using automatic differentiation or approximated by directional differences using only residual evaluations. This method eliminates the burdensome and error-prone procedure of hand coding an analytic Jacobian and reduces the runtime memory footprint, since the Jacobian is not explicitly stored.

J.5.3 Nonlinear Elimination

The disadvantage of the full Newton approach is that it requires forming and solving the fully-coupled Newton systems, making it very difficult to use nonlinear solvers, linear solvers, and preconditioners that are specialized to each system component. An alternative approach that maintains the quadratic convergence of Newton's method, but allows for greater flexibility in the choice of solver for each system component is nonlinear elimination [Pawlowski et al. (2011)].

J.6 Simple Coupling Strategy for Aeroelastic Problems Using *Rocstar*

The purpose of this section is to describe, in detail, the coupling algorithm and information transfer between the various components (fluids solver, structures solver, and orchestrator) of the *Rocstar* integrated code.

J.6.1 Variables Definitions and Nomenclature

ξ	general (discrete) coordinates in geometric space (e.g., x, y, z or r, θ, ϕ)
t	time
Δt	system timestep
n	timestep index
α	local time index, $(t - t^n)/\Delta t^n \in [0...1]$

s	structure
f	fluid
i	interface
o	orchestrator

$\vec{u}$	displacement field
$\vec{v}$	velocity field
$\vec{a}$	acceleration field
$\vec{t}$	traction field

p	pressure
ρ	density
$\dot{m}$	mass flux (set to zero in absence of burning)
$\dot{r}_b$	burning rate (set to zero in absence of burning)

$\vec{x}$	undeformed location
$\vec{y}$	deformed location
$\vec{n}$	deformed surface normals (measured positive into structure)
$\vec{N}$	undeformed surface normals (measured positive into structure)

$(\cdot)_{\text{sn}}$	value computed at *nodes* of the structure interface
$(\cdot)_{\text{sf}}$	value computed at *faces* of the structure interface
$(\cdot)_{\text{fn}}$	value computed at *nodes* of the fluid interface
$(\cdot)_{\text{ff}}$	value computed at *faces* of the fluid interface

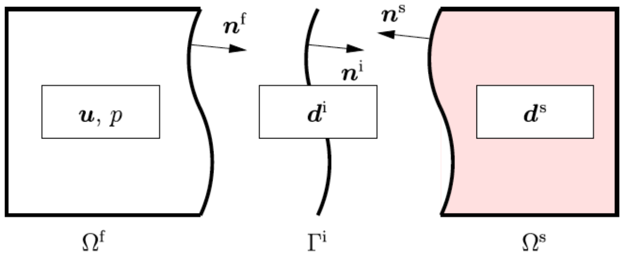
$(\hat{\cdot})$	value associated with particles
$(\bar{\cdot})$	value associated with mesh
$(\cdot)^{-}$	solutions of previous system timestep (for interpolation or extrapolation)

No-slip boundary condition: $\vec{v}_{\text{f}} \cdot \hat{\vec{t}} = \vec{v}_{\text{s}} \cdot \hat{\vec{t}}$, where $\hat{\vec{t}}$ denotes surface tangents.

Conservation of velocity: $\vec{v}_{\text{f}} = \vec{v}_{\text{s}}$.

Conservation of momentum: $\vec{t}_{\text{s}} = \vec{t}_{\text{f}}$.

Continuity of displacement: $\vec{u}_{\text{f}} = \vec{u}_{\text{s}}$.



$\boldsymbol{n}^{\text{f}}$
$\boldsymbol{n}^{\text{s}}$
$\boldsymbol{n}^{\text{i}}$
$\boldsymbol{u}, p$
$\boldsymbol{d}^{\text{i}}$
$\boldsymbol{d}^{\text{s}}$
Ω^{f}
Γ^{i}
Ω^{s}

Figure J.65: Three-field setup: fluid field Ω_{f}, interface Γ_{i}, and structural field Ω_{s} along with respective domain normals and variables. It is important to note the position of Γ_{i} is obviously varying with time and is only defined through the interaction of both fields. Figure taken from [Gerstenberger and Wall (2008)].

J.6.2 Subcycling Scheme

The individual component codes, fluid (f) and structure (s), proceed at their respective timesteps (Δt_{f} and Δt_{s}, respectively), i.e., they are allowed to subcycle based on their respective stability and precision criteria. However, information is transferred between these component codes only at every "system timestep" (Δt), which is greater than or equal to the largest of the "component timesteps."

Interpolation and extrapolation During subcycling, a physical component can request boundary conditions from *Rocman* at any time $t \in [t^n, t^{n+1}]$ using a local time index, α:

$$\alpha = \frac{(t - t^n)}{\Delta t^n} \in [0...1] .$$

Rocman interpolates or extrapolates values using the following schemes to obtain values at time $n + \alpha$.

First, if it has values at times n and $n+1$, the interpolation takes the form of

$$v^{n+\alpha} = v^n + \alpha(v^{n+1} - v^n).$$

Second, if it has values at times $n-1$ and n, and the extrapolation takes the form of

$$v^{n+\alpha} = v^n + \alpha \Delta t^n \frac{v^n - v^{n-1}}{\Delta t^{n-1}}.$$

Note that we may also simplify them and have $v^{n+\alpha} = v^n$. Using interpolated/extrapolated values might deliver higher order accuracy and allow use of larger system timesteps (i.e., more solver subcycles).

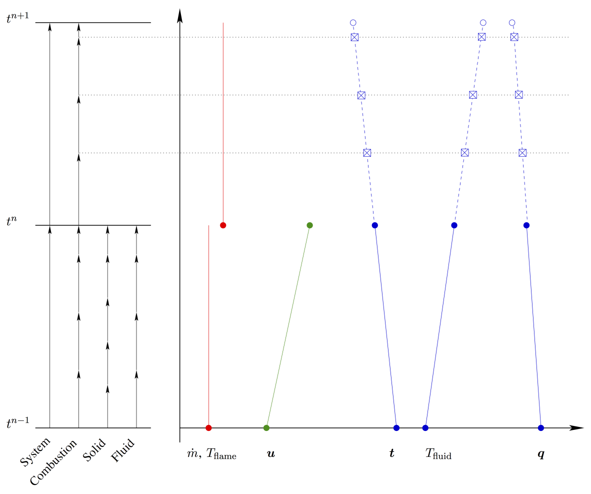
Subcycling without and predictor-corrector iterations Figure J.66a shows a schematic overview of the coupling among the physical modules as they march in time from t^n to t^{n+1} without predictor-corrector iterations. The lhs of the figure indicates how three physical modules march at their own timesteps to meet the user-specified system timestep. The rhs of the figure depicts variations of some of the interface quantities in time. Interface quantities associated with the combustion code are shown in red, a solid interface quantity is shown in green, and some fluid interface variables are indicated in blue.

At the beginning of a system timestep, the combustion solver takes as many steps as required to meet the specified system timestep. Since the fluid solver has not yet marched to its solution from t^n to t^{n+1}, the combustion solver receives time-extrapolated values for the fluid temperature and heat flux at the interface from the orchestrator. The extrapolated variations are indicated by dashed blue lines, and the values passed to the combustion solver are marked by the cross symbols.

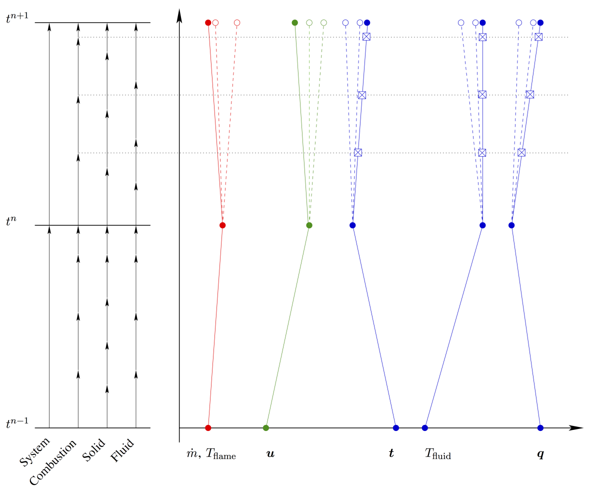
The extraction of the combustion solver from the fluid solver and the use of extrapolated values in the coupling algorithm leads to two important benefits. First, it increases the modularity of the coupled code and hence the ease with which plug-and-play may be performed. Second, and perhaps more importantly, the variation in time of each interface quantity is now uniquely defined by the orchestrator, which means that it should be possible to rigorously enforce mass conservation even when a simulation employs subcycling.
A schematic overview of the coupling among the physical modules as they march in time from t^n to t^{n+1} with predictor-corrector steps is shown in Figre J.66b. In contrast to Figure J.66a, dashed lines indicate variations of interface quantities during previous predictor-corrector steps. Thus the orchestrator can provide, for example, the combustion code with a fluid temperature and heat flux at the interface based on the variation of these quantities from the last predictor-corrector step. Only during the first predictor-corrector step is it necessary to use extrapolation.

J.6.3 Communication Mechanism

All components communicate with each other through the orchestrator. Each component owns some buffers for storing incoming or outgoing messages, which are registered to the orchestrator. The orchestrator copies data from outgoing buffers to respective incoming buffers, and performs algebraic manipulations for the buffered data as necessary. In this parlance, "send" indicates putting data into *outgoing* buffers. The orchestrator also provides subroutines to manipulate interface data. The prerequisite of these subroutines is that the data in the outgoing buffers are up-to-date. The output of these subroutines are stored in incoming buffers of the physical components.



(a) Coupling *without* predictor-corrector steps.



(b) Coupling *with* predictor-corrector steps.

Figure J.66: Schemata of the relationship between solutions without (J.66a) and with (J.66b) predictor-corrector steps used during system timestepping from t^n to t^{n+1}. In both cases, interface quantities associated with the combustion solver are shown in red, structure solver in green, and fluid solver in blue. Dashed lines indicate data extrapolation in (J.66a) and variations in interface quantities during previous predictor-corrector steps in (J.66b). (Only during the first predictor-corrector step is it necessary to use extrapolation.) Values passed to the different solvers are marked by cross symbols.

Interface quantities to be stored

particle displacement	$\hat{\vec{u}}(\xi,t) = \vec{u}(\vec{x}(\xi,t),t)$
parameterized velocity	$\hat{\vec{v}}(\xi,t) = \frac{\partial \hat{\vec{u}}(\xi,t)}{\partial t}$
parameterized acceleration	$\hat{\vec{a}}(\xi,t) = \frac{\partial^2 \hat{\vec{u}}(\xi,t)}{\partial t^2}$
mesh displacement	$\bar{\vec{u}}(\xi,t) = \vec{x}(\xi,t) - \vec{x}(\xi,0)$
mesh velocity	$\bar{\vec{v}}(\xi,t) = \frac{\partial \vec{x}(\xi,t)}{\partial t}$
mesh acceleration	$\bar{\vec{a}}(\xi,t) = \frac{\partial^2 \vec{x}(\xi,t)}{\partial t^2}$
particle velocity	$\vec{v}(\vec{x},t) = \frac{\partial \vec{u}(\vec{x},t)}{\partial t} = \hat{\vec{v}} - \nabla_x \vec{u} \cdot \bar{\vec{v}}$
deformed location	$\vec{y}(\xi,t) = \vec{x}(\xi,t) + \hat{\vec{u}}(\xi,t)$

Data transfer parameters:

- mode of load transfer (pressure or tractions)
- order of interpolation in time
- ρ_s (solid density for fluid-only simulations)

Structure domain

- Outgoing data to orchestrator (fluid)
  - Defined on interacting surface patches:
    * $(\rho_\mathrm{s})^{n+1}_\mathrm{sf}$
    * $(\vec{v}_\mathrm{s})^{n+1}_\mathrm{sn}$
    * $\left(\hat{\vec{u}}\right)^{n+1}_\mathrm{sn} = (\vec{y} - \vec{x})^{n+1}_\mathrm{sn}$
    * $(\Delta\vec{u})^{n+1}_\mathrm{sn}$
  - Defined on *both* interacting *and* noninteracting patches: $(\vec{x})^{n+1}_\mathrm{sn}$
- Incoming data from orchestrator (fluid)
  - Defined on interacting surface patches: $(\vec{t}_\mathrm{s})^{n+\alpha}_\mathrm{sf}$
  - Defined on *both* interacting *and* noninteracting surface patches: $(\bar{\vec{v}})^{n+\alpha}_\mathrm{sn}$

Fluid domain

- Outgoing data to orchestrator (structure)
  - Defined on interacting surface patches:
    * $(p_\mathrm{f})^{n+1}_\mathrm{ff}$
    * $(\vec{n}_\mathrm{f})^{n+\alpha}_\mathrm{ff}$ (outwards normal)
    * $(\rho_\mathrm{f})^{n+\alpha}_\mathrm{ff}$
    * $(\vec{t}_\mathrm{f})^{n+1}_\mathrm{ff}$
  - Defined on *both* interacting *and* noninteracting surface patches: $(\vec{y})^{n+1}_\mathrm{fn}$
- Incoming data from orchestrator (structure)
  - Nodal displacement vector of grid points due to boundary motion: $(\Delta\vec{u})^{n+\alpha}_\mathrm{fn}$
  - Defined on interacting surface patches: $(\rho_\mathrm{f}\vec{v}_\mathrm{f})^{n+\alpha}_\mathrm{ff}$

Orchestrator

- Intermediate data for structure solver
  - For surface propagation/deformation, defined on *both* interacting *and* noninteracting surface patches:
    * $(\bar{\vec{v}}_\mathrm{s})^{n+1}_\mathrm{sn}$
    * $(\bar{\vec{v}}_\mathrm{s})^{-}_\mathrm{sn}$
    * $\left(\vec{N}\right)^{n+1}_\mathrm{sf}$
  - Defined on interacting surface patches:
    * $(\vec{t}_\mathrm{s})^{n+1}_\mathrm{sf}$

    * $(\vec{t}_\mathrm{s})^{-}_\mathrm{sf}$
    * $(\vec{u})^{n+1}_\mathrm{sn}$
    * $(\vec{v}_\mathrm{s})^{n+1}_\mathrm{sf}$
    * $(\vec{y})^{n}_\mathrm{sn}$

- Intermediate data for fluid solver
  - $(\vec{t}_\mathrm{s})^{n+1}_\mathrm{ff}$, $(\vec{v}_\mathrm{s})^{n+1}_\mathrm{ff}$, $(\vec{v}_\mathrm{s})^{n+\alpha}_\mathrm{ff}$, and $(\vec{v}_\mathrm{s})^{-}_\mathrm{ff}$
  - Helpers for computing $(\Delta\vec{u})_\mathrm{fn}$ or transferring from structure to fluid (see Step (4) in Section J.6.4)
    * $(\Delta\vec{u})^{n+1}_\mathrm{fn}$
    * $(\vec{v}_\mathrm{o})^{n+1}_\mathrm{fn}$
    * $(\vec{v}_\mathrm{o})^{n+\alpha}_\mathrm{fn}$
    * $(\vec{v}_\mathrm{o})^{-}_\mathrm{fn}$
    * $(\vec{y})^{0}_\mathrm{fn}$
  - For convergence check if including predictor-corrector iterations:
    * $(\vec{t}_s)^\mathsf{pre}_\mathrm{ff}$
    * $(\vec{v}_s)^\mathsf{pre}_\mathrm{ff}$
    * $(\Delta\vec{u})^\mathsf{pre}_\mathrm{ff}$

J.6.4 Controlling Program Flow – FSI Coupling Scheme

Starting point: Assume the solution is known everywhere in the fluid and structure domains at time t^n.

(0) **Initialization**: To be called only at time 0.

(0.1) Fluid solver sends the following information to the orchestrator:
- $(p_\mathrm{f})^0_\mathrm{ff}$
- $(\rho_\mathrm{f})^0_\mathrm{ff}$
- $(\vec{t}_\mathrm{f})^0_\mathrm{ff}$
- $(\vec{y})^0_\mathrm{fn}$
- $(\vec{n}_\mathrm{f})^0_\mathrm{ff}$

(0.2) Structure solver sends the following information to the orchestrator:
- $(\rho_\mathrm{s})^0_\mathrm{sf}$
- $\left(\hat{\vec{u}}\right)^0_\mathrm{sn}$
- $(\vec{x})^0_\mathrm{sn}$

(0.3) Orchestrator computes $(\vec{y})^0_\mathrm{sn} = (\vec{x})^0_\mathrm{sn} + \left(\hat{\vec{u}}\right)^0_\mathrm{sn}$

(1) **Orchestrator updates data to structure solver:** Transfers $\left(\vec{t}_\mathrm{s}\right)_\mathrm{ff}$ to $\left(\vec{t}_\mathrm{s}\right)_\mathrm{sf}$. The time index is n if using PC iterations and $n+1$ otherwise.

(2) **Interface code invoked by orchestrator**: Propagate undeformed boundary (on solid interface mesh) using Huygens' construction due to only burning. In this first implementation, the connectivity of the surface elements is assumed to remain unchanged (i.e., no interface nodes or elements are added or removed during the burning process). The reasoning is that Huygens' construction is only valid for burning, not for structural displacements, and that $\dot{r}_b$ is defined on the deformed configuration, whereas the solid code applies mesh motion to the undeformed configuration.

(2.1) Assigns propagated interface to be undeformed (structure or fluid) interface

(2.2) Computes normals $\left(\vec{N}\right)^{n+1}_{\text{face}}$ of propagated interface using $(\vec{x})^{n}_{\text{node}}$ [see Step (0.2) and Step (3.4)]

(2.3) Propagates burning faces using mesh motion velocities $(\vec{v})^{n+1}_{\text{face}} = 0$, where $\vec{v}$ is $\bar{\vec{v}}$ for structure and $\vec{v}_{\text{o}}$ for fluid [see Step (3.4)]

(2.4) Transfers mesh motion from faces to nodes by geometric construction

(2.5) Interpolates mesh motion to nonburning nodes

(2.6) Sends $(\vec{v})^{n+1}_{\text{node}}$ to owner of propagated interface to be used as boundary condition for mesh motion in volume in Step (3.3) or Step (5.4)

(3) **Structure displacements:** Solver subcycles by repeating steps

(3.1) Obtains $\left(\vec{t}_{\text{s}}\right)^{n+\alpha}_{\text{sf}}$ by extrapolation and $\left(\bar{\vec{v}}\right)^{n+\alpha}_{\text{sn}}$ by interpolation [see Step (2.6)] from orchestrator

(3.2) Solves for mesh motion, i.e., computes $\bar{\vec{v}}^{n+\alpha}$ and then $\bar{\vec{u}}^{n+\alpha}$ over the volume

(3.3) Solves for structural motion, i.e., computes $\hat{\vec{v}}$, $\hat{\vec{u}}$, and $\hat{\vec{a}}$ using $\left(\vec{t}_{\text{s}}\right)^{n+\alpha}_{\text{sf}}$

(3.4) At the end of subcycling, computes $(\vec{v}_{\text{s}})^{n+1}_{\text{sn}}$ and sends $(\rho_{\text{s}})^{n+1}_{\text{sf}}$, $\left(\hat{\vec{u}}\right)^{n+1}_{\text{sn}}$, $(\vec{u})^{n+1}_{\text{sn}}$ (total displacement), $(\vec{v}_{\text{s}})^{n+1}_{\text{sn}}$, and $(\vec{x})^{n+1}_{\text{sn}}$to orchestrator

(4) **Orchestrator updates data to fluid solver:** Computes and transfers quantities of interest

(4.1) Computes $(\vec{y})^{n+1}_{\text{sn}} = (\vec{x})^{n+1}_{\text{sn}} + \left(\hat{\vec{u}}\right)^{n+1}_{\text{sn}}$

(4.2) Transfers $(\vec{u})^{n+1}_{\text{sn}}$ to $(\vec{u})^{n+1}_{\text{fn}}$ and $(\vec{v}_{\text{s}})^{n+1}_{\text{sn}}$ to $(\vec{v}_{\text{s}})^{n+1}_{\text{ff}}$

(4.3) Computes $(\Delta\vec{u})^{n+1}_{\text{fn}} = \left((\vec{y})^{0}_{\text{fn}} + (\vec{u})^{n+1}_{\text{fn}}\right) - (\vec{y})^{n}_{\text{fn}}$ [see Step (2.6)]

(5) **Fluid displacements:** Solver subcycles by repeating the steps

(5.1) Computes $(\rho_{\text{f}})^{n+\alpha}_{\text{ff}}$ and sends $(\rho_{\text{f}})^{n+\alpha}_{\text{ff}}$ and $(\vec{n}_{\text{f}})^{n+\alpha}_{\text{ff}}$ to orchestrator

(5.2) Obtains injection velocity at fluid faces from orchestrator, which were computed as $(\rho_{\text{f}}\vec{v}_{\text{f}})^{n+\alpha}_{\text{ff}} = (\rho_{\text{f}})^{n+\alpha}_{\text{ff}}\,(\vec{v}_{\text{s}})^{n+\alpha}_{\text{ff}}$

(5.3) Obtains $(\Delta\vec{u})^{n+\alpha}_{\text{fn}}$ from orchestrator [see Step (4.3)]

(5.4) Solves for mesh motion at fluid nodes and field variables at face centers

(5.5) At the end of subcycling, sends $(p_{\text{f}})^{n+1}_{\text{ff}}$, $\left(\vec{t}_{\text{f}}\right)^{n+1}_{\text{ff}}$, and $(\vec{y})^{n+1}_{\text{ff}}$ to orchestrator

(6) **Orchestrator updated by fluid solver:** Data transferred and convergence checked

(6.1) Conserves linear momentum by enforcing $\left(\vec{t}_{\text{s}}\right)_{\text{ff}} = \left(\vec{t}_{\text{f}}\right)_{\text{ff}}$ [see Step (0.1) and Step (5.5)]

(6.2) With PC iterations, checks convergence of $\vec{t}_\mathrm{s}$, $\Delta\vec{u}$, and $\vec{v}_\mathrm{s}$ on fluid faces

(6.3) If converged ...

- Backs up $\left(\vec{t}_\mathrm{s}\right)^n_\mathrm{sf}$, $\left(\vec{\bar{v}}\right)^n_\mathrm{sn}$, $(\Delta\vec{u})^n_\mathrm{fn}$, and $(\vec{v}_\mathrm{s})^n_\mathrm{ff}$
- Advances system timestep (Δt)

Remark: With PC iterations, the orchestrator always uses interpolation for obtaining values at $n+\alpha$ if iPredCorr > 1 (i.e., after the first PC iteration). At time t^0, the orchestrator sets values at $n+\alpha$ to the most recent values (i.e., uses constant extrapolation), which avoids requiring an initial guess for all quantities at time t^0.

[Geubelle et al. (2004)]

J.7 JFNK Coupling Strategy for Aeroelastic Problems

Jacobian-free Newton–Krylov (JFNK) methods are becoming increasingly popular in many branches of computational physics. They cite numerous examples in fluid dynamics, plasma physics, reactive flows, flows with phase change, radiation diffusion, radiation hydrodynamics, and geophysical flows. It is interesting to note, however, that JFNK methods have not become the approach of choice in the numerical solution of the compressible Navier–Stokes equations in computational fluid dynamics and aerodynamics. The major NASA flow solvers typify the algorithms that are most popular in the computational aerodynamics community. *OVERFLOW* and *CFL3D* are both based on implicit approximate factorization methods, *TLNS3D* and *CART3D* utilize multistage explicit schemes with multigrid, and *FUN3D*, although it includes a Newton–Krylov option, is usually run using either a point implicit procedure or an implicit line relaxation scheme. All of these approaches are more mature than JFNK methods.

The lack of broad acceptance of JFNK methods in the computational fluid dynamics community stems from several factors. Although the Jacobian matrix is not required, some sort of matrix is typically formed to precondition the linear system. Together with the Krylov subspace, this can lead to higher memory use than some of the more popular methods. In addition, formation of this matrix can require programming effort, e.g., hand linearization of the discrete residual equations. Completely matrix-free algorithms avoid these issues by using a solver as preconditioner but are typically slower and often inherit the shortcomings of solver used. Moreover, a JFNK method often involves a number of parameters, and there some effort involved in ensuring that suitable values are chosen for specific classes. Poorly chosen parameter values can lead to inefficient and unreliable algorithms. Other algorithms require some parameter selection as well, but often somewhat fewer, and optimal values for specific problem classes are better established than for the relatively newer JFNK algorithms. Furthermore, efficient globalization is sometimes difficult to achieve; the highly nonlinear behavior of some field-equation turbulence models can be particularly problematic. Finally, there are some subtle aspects of Newton–Krylov methods that are not often reported and can also lead to inefficient and unreliable performance if handled improperly.

The above issues notwithstanding, JFNK methods also offer many compelling potential advantages in the solution of compressible flows. They can be the most efficient option for extremely stiff problems, where stiffness can be introduced by multiple scales associated with complex physics, such as chemical reactions, or stiff source terms, such as those introduced by some turbulence models. In addition, Newton-like convergence properties of JFNK methods are ideal when deep convergence is needed. Consequently, they are very effective in the context of aerodynamic shape optimization. Moreover, their convergence tends to be insensitive to the properties of the mesh. For example, JFNK methods converge well on meshes with high aspect ratios. Also, high order methods can lead to reduced stability bounds for explicit iterative methods;

hence JFNK methods can be appropriate in this context. Furthermore, JFNK methods can be a very efficient means of solving the nonlinear problem that arises at each time step of a time-accurate implicit algorithm. Finally, the parameters involved in JFNK methods can be used to advantage to tune the algorithm to be very efficient for specific problem classes or to adjust the algorithm for particularly difficult problems.

[Chisholm and Zingg (2009)]

J.7.1 Introduction to JFNK Method

It is our observation that solution strategies for nonlinearly implicit PDEs have evolved along somewhat different trajectories in the applied mathematics community and the computational physics community. In discussing solution strategies for boundary value problems (BVPs), the applied mathematics community has emphasized Newton-based methods. Outside of finite element practitioners, the computational fluid dynamics (CFD) community has emphasized Picard-type linearizations and splitting by equation or splitting by coordinate direction. The difference in predominating approach (Newton versus Picard) seems stronger for implicit initial value problems (IVPs).

Again, the applied mathematics community has focused on Newton-based methods and on converging the nonlinear residual within a timestep. In the computational physics community, operator splitting (time splitting, fractional step methods) has been the “bread and butter” approach, with little attention to monitoring or converging the nonlinear residual within a timestep, often allowing a splitting error to remain in time. In both IVP and BVP contexts, the concept of splitting (a form of divide-and-conquer at the operator level) has been motivated by the desire to numerically integrate complicated problems with limited computer resources. This tension does not vanish with terascale hardware, since the hardware is justified by the need to do ever more refined simulations of more complex physics. One can argue that the stakes for effective methods become higher, not lower, with the availability of advanced hardware.

Recent emphasis on achieving predictive simulations has caused computational scientists to take a deeper look at operator splitting methods for IVPs and the resulting errors. As a result, the computational physics community is now increasingly driven towards nonlinear multigrid methods and Jacobian-free Newton–Krylov methods. These nonlinear iterative methods have grown out of advances in linear iterative methods, multigrid methods, and preconditioned Krylov methods

An advantage of JFNK is that the code development curve is not steep, given a subroutine that evaluates the discrete residual on the desired (output) grid. Furthermore, inexpensive linearized solvers can be used as preconditioners. Developing effective preconditioners may be a challenge, and the storage required for the preconditioner and Krylov vectors may be a limitation. An important feature of JFNK is that the overall nonlinear convergence of the method is not directly affected by the approximations made in the preconditioning. The overall framework, making use of multiple discrete approximations of the Jacobian operator, has a polymorphic object-oriented flavor that lends itself well to modern trends in software design and software integration. In many cases, JFNK has been used to retrofit existing BVP and IVP codes while retaining the most important investments (in the physics routines) of the original code.

[Knoll and Keyes (2004)]

J.7.2 Fundamentals JFNK Methods

The Jacobian-free Newton–Krylov method is a nested iteration method consisting of at least two, and usually four, levels. The primary levels, which give the method its name, are the loop over the Newton corrections and the loop building up the Krylov subspace out of which each Newton correction is drawn. Interior to the Krylov loop, a preconditioner is usually required, which can itself be direct or iterative. Outside of the Newton loop, a globalization method is often required. This can be implicit timestepping, with timesteps chosen to preserve a physically accurate transient or otherwise, or this can be some other form of parameter continuation such as mesh sequencing.

Review of Newton methods The Newton iteration for $\vec{F}(\vec{u}) = 0$ derives from a multivariate Taylor expansion about a current point $\vec{u}^k$:

$$\vec{F}(\vec{u}^{k+1}) = \vec{F}(\vec{u}^k) + \vec{F}'(\vec{u}^k)(\vec{u}^{k+1} - \vec{u}^k) + \text{HOTs}. \tag{J.14}$$

Setting the rhs to zero and neglecting the terms of higher-order curvature yields a strict Newton method, iteration over a sequence of linear systems

$$\vec{J}(\vec{u}^k)\delta\vec{u}^k = -\vec{F}(\vec{u}^k), \quad \vec{u}^{k+1} = \vec{u}^k + \delta\vec{u}^k, \quad k = 0, 1, \ldots \tag{J.15}$$

given $\vec{u}^0$. Here, $\vec{F}(\vec{u})$ is the vector-valued function of nonlinear residuals, $\vec{J} \equiv \vec{F}'$ is its associated Jacobian matrix, $\vec{u}$ is the state vector to be found, and k is the nonlinear iteration index. The Newton iteration is terminated based on a required drop in the norm of the nonlinear residual

$$\frac{\left\|\vec{F}(\vec{u}^k)\right\|}{\left\|\vec{F}(\vec{u}^0)\right\|} < \text{tol}_{\text{res}}, \tag{J.16}$$

and/or a sufficiently small Newton update

$$\frac{\|\delta\vec{u}^k\|}{\|\vec{u}^k\|} < \text{tol}_{\text{update}}. \tag{J.17}$$

For a scalar problem, discretized into n equations and n unknowns, we have

$$\vec{F}(\vec{u}) = \{F_1, F_2, \ldots, F_i, \ldots, F_n\} \tag{J.18}$$

and

$$\vec{u} = \{u_1, u_2, \ldots, u_i, \ldots, u_n\}, \tag{J.19}$$

where i is the component index. In vector notation, the (i, j)th element (ith row, jth column) of the Jacobian matrix is

$$J_{ij} = \frac{\partial F_i(\vec{u})}{\partial u_j}. \tag{J.20}$$

In this scalar example there is a one-to-one mapping between grid points and rows in the Jacobian. Forming each element of $\vec{J}$ requires taking analytic or discrete derivatives of the system of equations with respect to $\vec{u}$. This can be both error-prone and time consuming for many problems in computational physics. Nevertheless, there are numerous examples of forming $\vec{J}$ numerically and solving eq (J.15) with a preconditioned Krylov method. $\vec{J}$ can also be formed using automatic differentiation.

Krylov methods Krylov subspace methods are approaches for solving large linear systems introduced as direct methods in the 1950s, whose popularity took off after Reid reintroduced them as iterative methods in 1971. They are projection (Galerkin) or generalized projection (Petrov–Galerkin) methods for solving $\vec{A}\vec{x} = \vec{b}$ using the Krylov subspace, $\vec{K}_j$,

$$\vec{K}_j = \text{span}(\vec{r}_0, \vec{A}\vec{r}_0, (\vec{A})^2\vec{r}_0, \ldots, (\vec{A})^{j-1}\vec{r}_0),$$

where $\vec{r}_0 = \vec{b} - \vec{A}\vec{x}_0$. These methods require only matrix-vector products to carry out the iteration (not the individual elements of $\vec{A}$), and this is key to their use with Newton's method.

The widely used Generalized Minimal RESidual method (GMRES) is an Arnoldi-based method. In GMRES the Arnoldi basis vectors form the trial subspace out of which the solution is constructed. One matrix–vector product is required per iteration to create each new trial vector, and the iterations are terminated based on a by-product estimate of the residual that does not require explicit construction of intermediate residual vectors or solutions – a major beneficial feature of the algorithm. GMRES has a residual minimization property in the Euclidean norm (easily adaptable to any inner-product norm) but requires the storage of all previous Arnoldi basis vectors. Full restarts, seeded restarts, and moving fixed-sized windows of Arnoldi basis vectors are all options for fixed-storage versions.

As a result of numerous studies, we tend to use GMRES (and its variants) almost exclusively with JFNK. The resulting pressure on memory has put an increased emphasis on quality preconditioning. We believe that it is only through effective preconditioning that JFNK is feasible on large scale problems. It is in the preconditioner that one achieves algorithmic scaling and also in the preconditioner that one may stand to lose the natural excellent parallel scaling enjoyed by all other components of the JFNK algorithm as applied to PDEs.

Jacobian-free Newton–Krylov methods The origins of the Jacobian-free Newton—Krylov method can be traced back to publications motivated by the solution of ODEs and publications motivated by the solution of PDEs. The primary motivation in all cases appears to be the ability to perform a Newton iteration without forming the Jacobian. Within the ODE community these methods helped to promote the use of higher order implicit integration. The studies on PDE problems focused on the use of nonlinear preconditioning, preconditioners constructed from linear parts of the PDEs, and the addition of globalization methods.

In the JFNK approach, a Krylov method is used to solve the linear system of equations given by eq (J.15). An initial linear residual, $\vec{r}_0$, is defined, given an initial guess, $\delta\vec{u}_0$, for the Newton correction,

$$\vec{r}_0 = -\vec{F}(\vec{u}) - \vec{J}\delta\vec{u}_0. \tag{J.21}$$

Note that the nonlinear iteration index, k, has been dropped. This is because the Krylov iteration is performed at a fixed k. Let j be the Krylov iteration index. Since the Krylov solution is a Newton correction, and since a locally optimal move was just made in the direction of the previous Newton correction, the initial iterate for the Krylov iteration for $\delta\vec{u}_0$ is typically zero. This is asymptotically a reasonable guess in the Newton context, as the converged value for $\delta\vec{u}$ should approach zero in late Newton iterations. The jth GMRES iteration minimizes $\|\vec{J}\delta\vec{u}_j + \vec{F}(\vec{u})\|_2$ within a subspace of small dimension, relative to n (the number of unknowns), in a least-squares sense. $\delta\vec{u}_j$ is drawn from the subspace spanned by the Krylov vectors, $\{\vec{r}_0, \vec{J}\vec{r}_0, (\vec{J})^2\vec{r}_0, \ldots, (\vec{J})^{j-1}\vec{r}_0\}$, and can be written as

$$\delta\vec{u}_j = \delta\vec{u}_0 + \sum_{i=0}^{j-1} \beta_i (\vec{J})^i \vec{r}_0, \tag{J.22}$$

where the scalars β_i minimize the residual. (In practice, $\delta\vec{u}_j$ is determined as a linear combination of the orthonormal Arnoldi vectors produced by GMRES.)

Upon examining eq (J.22) we see that GMRES requires the action of the Jacobian only in the form of matrix–vector products, which may be approximated by

$$\vec{J}\vec{v} \approx \left[\vec{F}(\vec{u}+\varepsilon\vec{v}) - \vec{F}(\vec{u})\right]/\varepsilon, \tag{J.23}$$

where ε is a small perturbation.

Eq (J.23) is simply a first-order Taylor series expansion approximation to the Jacobian, $\vec{J}$, times a (Krylov subspace) vector, $\vec{v}$. For illustration, consider the two coupled nonlinear equations, $F_1(u_1,u_2) = 0$ and $F_2(u_1,u_2) = 0$. The Jacobian matrix is

$$\vec{J} = \begin{bmatrix} \frac{\partial F_1}{\partial u_1} & \frac{\partial F_1}{\partial u_2} \\ \frac{\partial F_2}{\partial u_1} & \frac{\partial F_2}{\partial u_2} \end{bmatrix}$$

JFNK does not require the formation of this matrix; we instead form a result vector that approximates this matrix multiplied by a vector. Working backwards from eq (J.23), we have

$$\frac{\vec{F}(\vec{u}+\varepsilon\vec{v}) - \vec{F}(\vec{u})}{\varepsilon} = \begin{pmatrix} \frac{F_1(u_1+\varepsilon v_1, u_2+\varepsilon v_2) - F_1(u_1,u_2)}{\varepsilon} \\ \frac{F_2(u_1+\varepsilon v_1, u_2+\varepsilon v_2) - F_2(u_1,u_2)}{\varepsilon} \end{pmatrix}.$$

Approximating $\vec{F}(\vec{u}+\varepsilon\vec{v})$ with a first-order Taylor series expansion about $\vec{u}$, we have

$$\frac{\vec{F}(\vec{u}+\varepsilon\vec{v}) - \vec{F}(\vec{u})}{\varepsilon} \approx \begin{pmatrix} \frac{F_1(u_1,u_2) + \varepsilon v_1 \frac{\partial F_1}{\partial u_1} + \varepsilon v_2 \frac{\partial F_1}{\partial u_2} - F_1(u_1,u_2)}{\varepsilon} \\ \frac{F_2(u_1,u_2) + \varepsilon v_1 \frac{\partial F_2}{\partial u_1} + \varepsilon v_2 \frac{\partial F_2}{\partial u_2} - F_2(u_1,u_2)}{\varepsilon} \end{pmatrix},$$

which simplifies to

$$\begin{pmatrix} v_1 \frac{\partial F_1}{\partial u_1} + v_2 \frac{\partial F_1}{\partial u_2} \\ v_1 \frac{\partial F_2}{\partial u_1} + v_2 \frac{\partial F_2}{\partial u_2} \end{pmatrix} = \vec{J}\vec{v}.$$

The error in this approximation is proportional to ε. This matrix-free approach has many advantages. The most attractive is Newton-like nonlinear convergence without the costs of *forming* or *storing* the true Jacobian. In practice one forms a matrix (or set of matrices) for preconditioning purposes, so we eschew the common description of this family of methods as fully "matrix-free." However, the matrices employed in preconditioning can be simpler than the true Jacobian of the problem, so the algorithm is properly said to be "Jacobian-free."

A convergence theory has been developed for JFNK. Conditions are provided on the size of ε that guarantee local convergence. Issues regarding convergence are often raised with Newton-based methods, and Jacobian-free Newton–Krylov are no exception. Two specific situations known to cause convergence problems for JFNK are sharp nonlinear solution structure, such as a shock or a reaction front, and discontinuities in the nonlinear function, such as one might see in higher order monotone advection schemes. Issues of non-convergence tend to be seen more in boundary value problems and less in initial value problems.

For more information on JFNK, such as choosing the perturbation parameter ε and discussions on preconditioners, the reader is referred to [Knoll and Keyes (2004)]. [Bellavia et al. (2011)] also has some useful information on nonsymmetric preconditioner updates in NK methods.

J.7.3 Computational JFNK Algorithm for Aeroelastic Problems

Newton–Krylov methods require only the product of the Jacobian matrix with a given vector, rather than explicit access to the elements of the Jacobian. Krylov subspace methods do not require the system matrix $\hat{\vec{J}}(u)$ in an explicit form but only the product of $\hat{\vec{J}}(u)$ with a Krylov subspace vector $\vec{v}$ [Schieffer et al. (2010)].

References

Allan, B., Armstrong, R., Wolfe, A., Ray, J., and Bernholdt, D. (2002). The *CCA* core specification in a distributed memory spmd framework. *Concurr. Comput. Practices Exp.*, 5:323–345.

Allen, G., Dramlitsch, T., Foster, I., Karonis, N., Ripeanu, M., Seidel, E., and Toonen, B. (2001). Supporting efficient execution in heterogeneous distributed computing environments with *Cactus* and *Globus*. In *Proc. Supercomput.*

Arienti, M., Hung, P., Morano, E., and Shepherd, J. E. (2003). A level set approach to eulerian-lagrangian coupling. *J. Comput. Phys.*, 185:213–251.

Bartlett, R. A., Heroux, M. A., and Willenbring, J. M. (2012). Tribits lifecycle model. Technical report, Sandia National Laboratory.

Bassetti, F., Brown, D., Davis, K., Henshaw, W., and Quinlan, D. (1998). *Overture*: an object-oriented infrastructure for high performance scientific computing. In *Proc. Supercomput.*

Bellavia, S., Bertaccini, D., and Morini, B. (2011). Nonsymmetric preconditioner updates in Newton–Krylov methods of nonlinear systems. *J. Sci. Comput.*, 33:2595–2619.

Benra, F.-K., Dohmen, H. J., Pei, J., Schuster, S., and Wan, B. (2011). A comparison of one-way and two-way coupling methods for numerical analysis of fluid-structure interactions. *Journal of Applied Mathematics*, 2011:16.

Budge, K. G. and Peery, J. S. (1998). Experiences developing *ALEGRA*: a C++ coupled physics framework. In *Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*.

Chisholm, T. T. and Zingg, D. W. (2009). A Jacobian-free Newton–Krylov algorithm for compressible turbulent fluid flows. *J. Comput. Phys.*, 228:3490–3507.

Council on Competitiveness (2011). Council on Competitiveness, 2010–2011 Annual Report. Technical report.

Dick, W. A., Fiedler, R. A., and Heath, M. T. (2006). Building *Rocstar*: Simulation science for solid propellant rocket motors. In *Proc. AIAA*, volume 4590, pages 1–10.

Elmer (2016). *Elmer*. https://www.csc.fi/web/elmer.

Gerstenberger, A. and Wall, W. A. (2008). An eXtended Finite Element Method/Lagrange multiplier based approach for fluid–structure interaction. *Comput. Methods Appl. Mech. Engrg.*, 197:1699–1714.

Geubelle, P. H., Jiao, X., Haselbacher, A. C., and Campbell, M. T. (2004). Numerical Coupling Interface in *Rocstar – Rocman* (GEN 3) Design Document. PDF.

Heroux, M. A., 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., Williams, A., and Stanley, K. (2005). An overview of the *Trilinos* projects. *ACM Trans. Math. Software*, 31:397–423.

Hou, G., Wang, J., and Layton, A. (2012). Numerical methods for fluid-structure interaction – a review. *Communications in Computational Physics*, 12:337–377.

Jaiman, R. K., Jiao, X., Geubelle, P. H., and Loth, E. (2004). Assessment of conservative load transfer for fluid-solid interface with nonmatching meshes. *International Journal for Numerical Methods in Engineering*, pages 1–40.

Jaiman, R. K., Jiao, X., Geubelle, P. H., and Loth, E. (2005). Assessment of conservative load transfer for fluid-solid interface with nonmatching meshes. *Intl. J. Numer. Methods Eng.*, pages 1–40.

Jiao, X. (2007). Face offsetting: A unified approach for explicit moving interfaces. *Journal of Computational Physics*, 220:612–625.

Jiao, X. and Heath, M. T. (2005). Common-refinement-based data transfer between non-matching meshes in multiphysics simulations. *International Journal for Numerical Methods in Engineering*, 61:2402–2427.

Jiao, X., Zheng, G., Alexander, P. J., Campbell, M. T., Lawlor, O. S., Norris, J., Hasselbacher, A., and Heath, M. T. (2006). A system integration infrastructure for coupled multiphysics simulations. *Engineering with Computers*, 22:293–309.

Kamakoti, R. and Shyy, W. (2004). Fluid-structure interaction for aeroelastic applications. *Prog. Aerosp. Sci.*, 40:535–558.

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., Rivière, B., Rüde, U., Scheibe, T., Shadid, J., Sheehan, B., Shephard, M., Siegel, A., Smith, B., Tang, X., Wilson, C., and Wohlmuth, B. (2013). Multiphysics simulations: Challenges and opportunities. *Intl. J. High Perform. Comput. Appl.*, 27:5.

Knoll, D. A. and Keyes, D. E. (2004). Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *J. Comput. Phys.*, 193:357–397.

M. A. Heroux et al. (2012). *Trilinos*. http://trilinos.sandia.gov.

Michler, C., Hulshoff, S., van Brummelen, E., and de Borst, R. (2004). A monolithic approach to fluid-structure interaction. *Computers and Fluids*, 33(5-6):839–848. Applied Mathematics for Industrial Flow Problems.

OpenFOAM Extend Project (2016). *OpenFOAM Extend Project*. http://www.extend-project.de.

Pawlowski, R., Bartlett, R., Belcourt, N., Hooper, R., and Schmidt, R. (2011). A theory manual for multi-physics code coupling in LIME. Technical report, Sandia National Laboratory.

Poppendieck, M. and Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley.

Reynders, J. V. W., Hinker, P. J., Cummings, J. C., Atlas, S. R., Banerjee, S., Humphrey, W. F., Karmesin, S. R., Keahey, K., Srikant, M., and Tholburn, M. (1996). *POOMA*: A infrastructure for scientific simulations on parallel architectures. In Wilson, G. V. and Lu, P., editors, *Parallel programming using C++*, pages 547–588. Massachusetts Institute of Technology.

Schafer, M. and Turek, S. (1996). Flow simulation with high-performance computers ii. dfg priority research program results 1993-1995. In Hirschel, E., editor, *Notes in Numerical Fluid Mechanics*, number 52, pages 547–566. Vieweg Weisbaden.

Schieffer, G., Ray, S., Bramkamp, F. D., Behr, M., and Ballmann, J. (2010). An adaptive implicit finite volume scheme for compressible turbulent flows about elastic configurations. In Schröder, W., editor, *Flow Modulation & Fluid-Structure Interaction*, volume 109, pages 25–51. Springer-Verlag.

Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D. (2013). Cfd vision 2030 study: A path to revolutionary computational aerosciences. Technical report, NASA Langley Research Center, Hampton, Virginia 23681-2199.

Stewart, J. R. and Edwards, H. C. (2004). A infrastructure approach for developing parallel adaptive multi-physics applications. *Finite Elem. Anal. Des.*, 40:1599–1617.

Turek, S. and Hron, J. (2006). *Fluid-Structure Interaction: Modelling, Simulation, Optimisation*, chapter Proposal for Numerical Benchmarking of Fluid-Structure Interaction between an Elastic Object and Laminar Incompressible Flow, pages 371–385. Springer Berlin Heidelberg, Berlin, Heidelberg.