

Rocom: An Object-Oriented, Data-Centric Software Integration Framework for Multiphysics Simulations*

Xiangmin Jiao

Michael T. Campbell

Michael T. Heath

Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{jiao,mtcampbe,heath}@uiuc.edu

ABSTRACT

We describe an object-oriented software integration framework, *Rocom*, abstracted from our five years of experience in developing a complex, integrated code for rocket simulation. *Rocom* provides a flexible mechanism for inter-module data exchange and function invocation in parallel multiphysics simulations. It is designed to minimize user effort and code changes for integration, facilitate interoperability between different programming languages (in particular, C++ and Fortran 90), and enable plug-and-play of different implementations of physics and computer science modules in an integrated system. Its unique abstraction of distributed objects allows cleaner inter-module interfaces and maximizes concurrency in development of different modules. Our framework also provides a set of reusable service utilities that allow rapid prototyping of various coupling algorithms without sacrificing performance.

Categories and Subject Descriptors

I.6.7 [Simulation and Modeling]: Simulation Support Systems—*Environments*; D.2.11 [Software Engineering]: Software Architectures—*domain-specific architectures, data abstraction*; D.1.5 [Software]: Object-Oriented Programming; D.2.12 [Software Engineering]: Interoperability

General Terms

Design, management

*Research supported by the U.S. Department of Energy through the University of California under subcontract B523819. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the U.S. Department of Energy, the National Nuclear Security Agency, or the University of California.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'03, June 23–26, 2003, San Francisco, California, USA.
Copyright 2003 ACM 1-58113-733-8/03/0006 ...\$5.00.

Keywords

Multiphysics simulation, system integration, object-oriented design, middleware, problem solving environments

1. INTRODUCTION

Large-scale numerical simulation of a complex system, such as a solid rocket motor, requires consideration of multiple physical components, such as fluid dynamics, solid mechanics, and combustion. Because of this multidisciplinary nature, development of such a simulation code typically involves collaboration among many research subgroups using a partitioned approach, in which the individual physics codes are developed more or less independently of the integration effort. The integration of such independently developed modules into a coherent, coupled system, particularly in a distributed parallel setting, is the central issue addressed in this paper.

In the integration of multiphysics codes, besides the complexities of physical modeling, numerical and geometrical methods, and parallel efficiency, there are also many challenges in software engineering. First, the data exchanged between modules must be *abstracted* appropriately so that inter-module interfaces can be as simple and clean as possible. Second, the software architecture must encourage good software practice, such as *encapsulation* and *code reuse*, and provide conveniences to code developers while being as *non-intrusive* as possible. Third, the software architecture must be *heterogeneous* to provide sufficient flexibility to application scientists and engineers in choosing appropriate discretization schemes, data structures, and programming languages according to their tastes and needs. Fourth, to support cutting-edge research, the software architecture must *maximize concurrency* in code development of different subgroups and support *rapid prototyping* of various coupling schemes through well-defined service components.

To address these issues, we have developed a software integration framework, *Rocom*, that provides a systematic, object-oriented, data-centric approach for inter-module interaction. Under this framework, a computation module constructs distributed objects called *windows* and registers its datasets into windows. With the authorization of their owner modules or the orchestration module, these datasets can later be retrieved from *Rocom* by other modules using handles provided by *Rocom*. Functions can be registered and invoked similarly through *Rocom*. 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. On top of *Rocom*, we have developed a set of service components to support implementation of various coupling schemes, including operations for manipulating jump conditions, storing and restoring internal states of modules for predictor-corrector iterations, and I/O for visualization and restart.

The remainder of the paper is organized as follows. Section 2 describes the rocket simulation application that motivates our work. Section 3 overviews the software challenges in the development of multiphysics simulation codes, which motivate the objectives of our framework. Section 4 explains the key concepts and object-oriented design of *Rocom*. Section 5 describes the architecture of the core of *Rocom*. Section 6 briefly discusses the utility services provided by our framework. Section 7 demonstrates the application of the framework to integrated rocket simulation. Section 8 compares the design of *Rocom* with some related scientific software frameworks. Section 9 concludes the paper.

2. MOTIVATING APPLICATION

The motivating multiphysics application for the integration framework described in this paper is an ongoing project at the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois (<http://www.csar.uiuc.edu>). The ultimate objective of CSAR is to develop an integrated software system for detailed whole-system simulation of solid rocket motors under normal and abnormal operating conditions. Figure 1 shows the overall structure of the current generation, GEN2.5, of our simulation code.

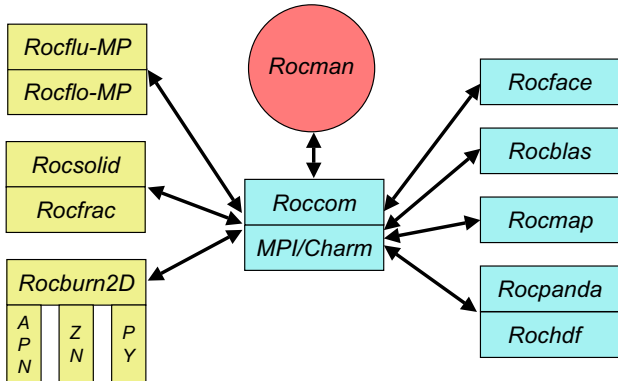


Figure 1: Software components of GEN2.5 rocket simulation code.

The boxes on the left in the figure show the physics modules. We have two 3-D explicit fluid dynamics solvers, *Rocflo-MP*, which uses block-structured meshes, and *Rocflu-MP*, which uses unstructured hybrid meshes. For solid mechanics, we have a 3-D implicit solver, *Rocsolid*, which uses unstructured multilevel hexahedral meshes, and an explicit solver, *Rocfrac*, which uses 4- and 10-node tetrahedral elements and can also utilize cohesive finite volume elements for crack propagation. The combustion module *Rocburn2D* uses various 1-D combustion models to determine the burning rate of the surface at the fluid-solid interface. On top

is the orchestration (or mediating) 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 our integration framework *Rocom*, as we discuss in later sections. 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 except *Rocflu-MP*, which uses the finite element framework of Charm++[4]. As the physical models and codes evolve, additional modules are expected to be added into the code.

Although it was motivated specifically by the needs of the rocket simulation application just described, the integration framework 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.

3. SOFTWARE CHALLENGES

In addition to the many challenges in physical modeling, computational algorithms, and parallel implementation, the development and coupling of a multicomponent simulation code such as the rocket simulation described in the previous section also entails numerous challenges in software engineering.

Data Management for Inter-Module Communication.

The physical components of our simulation code are developed by different research subgroups with differing expertise, independently of each other. These subgroups have their own preferences concerning spatial discretizations (i.e., meshes) and underlying data structures, and some of these modules are based on legacy codes developed prior to the current CSAR project. Therefore, unifying the data structures across these modules is impractical. However, a unified, language-independent, and abstract *view* of these data objects is necessary for data exchange between modules. This view must be flexible, self-descriptive, and both physics- and numerics-aware. It must encapsulate both data and metadata (such as dimensions and data types) of the objects, support irregularly distributed objects, and provide protection mechanisms for data integrity.

Concurrency in Component Development. Because the physical models and numerical algorithms, as well as codes for each module continue evolving, concurrent development of different modules is highly desirable. Constant evolution of the codes, however, would make it difficult for developers of service utilities, such as I/O modules, if they were exposed to the detailed data structures of physics modules. Developing a new capability based on a snapshot of an evolving physics module frequently leads to an unusable code and ultimately to loss of productivity.

Programming-Language Interoperability. The developers of different modules typically have a variety of preferences in programming languages. In the same spirit of allowing different methodologies, allowing coexistence of multiple programming languages is also highly desirable, especially since languages have their own strengths in dif-

ferent aspects. Currently, the most popular programming languages are Fortran 90 (F90) among engineers and C++ among computer scientists. Interoperability between these languages poses additional challenges. First, it is well-known that F90 and C++ have different naming and calling conventions. Furthermore, some data objects are incompatible across these languages: unlike C++, a character string in F90 is not null terminated, a pointer in F90 is not simply a memory address, and derived types in F90 and classes in C++ are generally incompatible. A common practice to interoperate between F90 and C++ is to provide primitive F77 and C interface wrappers, which sacrifices the object-orientedness of interfaces and is also tedious and inelegant.

Complexity of Coupling Schemes. Our simulation code takes a partitioned approach to system integration in which each physics module solves on its own domain, and any jump conditions between them are exchanged at the interfaces between domains. When an implicit solver is involved, a predictor-corrector style of implicit coupling scheme is necessary to iterate on the jump conditions. The implementation of jump conditions involves operations such as manipulating data on some interface mesh and transferring data between different meshes. These additional operations are independent of the physical modules and should be provided by service modules and invoked by a centralized orchestration module.

Plug-and-Play Capability. Intentionally, our simulation codes sometimes have multiple modules that provide the same functionality but use different methodologies. For example, we have two modules for structural mechanics, one explicit and one implicit, and two modules for gas dynamics, one structured and one unstructured. This redundancy is desirable for various reasons, including flexibility in selecting different methods based on particular situations, and verification by comparing results using different modules. In our experience, it is also particularly useful to allow substitution of some modules with dummy stubs, so that the remaining modules can run stand-alone within the coupled code, to allow progressive paths for integration and verification. Further, it is also desirable to allow plug-in of third-party modules. The design of the framework must support these capabilities while minimizing the associated complications of orchestration codes.

Debugging and Performance Tuning. Code debugging and testing take perhaps 40% of the time for general software development, and performance tuning requires additional effort for high-performance applications. These efforts are also substantial and challenging for system integration. Even if the individual components have been thoroughly tested, unforeseen situations may arise from interactions between subsystems, which are often more difficult to understand because code developers in general are familiar only with their own codes. It is unproductive, and often impractical, to gather all code developers together for debugging. For effective debugging, a framework must provide the ability to isolate bugs to certain modules quickly, so that the respective developers can investigate in more detail. In the same spirit, it must also be able to isolate performance bottlenecks to individual modules to facilitate performance tuning.

Checkpointing and Restarting. Production runs of our simulation codes frequently require days or even weeks of computation time. It is unwise to rely on continuous availability of a computer system for such a long period, because a dedicated debugging platform may not be stable enough and a production platform is shared among many users. It is therefore important to be able to checkpoint a run and then restart it, possibly on a different platform. For implicit coupling, the internal states of a physics module must be stored after the predictor-corrector iteration has converged and restored if not converged, and the data that must be stored are usually the same as those that must be written into restart files. It is desirable to have a consistent mechanism for checkpoint-restart and store-restore operations, and provide these capabilities opaquely to physical modules.

4. OBJECT-ORIENTED ABSTRACTION OF INTER-MODULE INTERFACES

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

4.1 Windows and Panes

Rocom organizes data and functions into distributed objects called *windows*. A window encapsulates a number of *data attributes* (such as the mesh and some associated field variables) and public *functions* of a module, any of which can be empty. A window can be partitioned into multiple *panes*, 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 window must have the same types of data members, although the sizes of data members may vary. A module constructs windows at runtime by creating attributes and registering the addresses of the attributes and functions. Typically, the attributes registered with *Rocom* are *persistent* (instead of temporary) datasets, in the sense that they live throughout the simulation (except that windows may need to be reinitialized at some events, such as remeshing). Different modules can communicate with each other only through windows, as illustrated in Figure 2.

A code module references windows, attributes, or functions using their names, which are of character-string type.

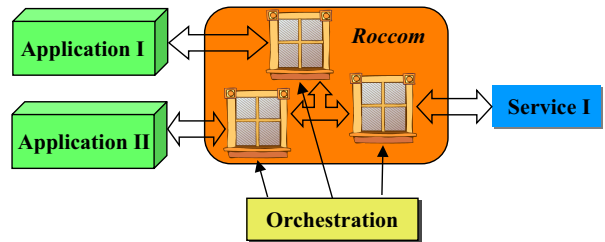


Figure 2: Schematic illustration of windows and panes.

Window names must be unique across all modules, and an attribute or function name must be unique within a window. A code module can obtain an integer *handle* of (i.e., a reference to) an attribute/function from *Rocom* with the combination of the window and attribute/function names. The handle of an attribute can be either *mutable* or *immutable*, where an immutable handle allows only read operations to its referenced attribute, 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.

4.2 Data Attributes

Data attributes of a window include mesh data, field variables, and window or pane attributes. The former two types of attributes are associated with nodes or elements. A nodal or elemental attribute 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. Its storage can have a *contiguous* or *staggered* layout: in the former, the dataset is stored in an array in which the data index within a node/element (vs. the index of nodes/elements) corresponds to the more rapidly changing dimension (i.e., the last dimension of C arrays and the first dimension of Fortran arrays); in the latter, the dataset is stored in an array in which the index within a node/element corresponds to the more slowly changing dimension, or stored in separate arrays, each of which contains one component of the dataset of all nodes/elements. Sometimes a dataset is not stored in consecutive memory space but is embedded in a larger array with a constant stride between neighboring entries of the dataset. To support this, *Rocom* allows users to specify a stride for nodal or elemental attributes.

Mesh Data. Mesh data include nodal coordinates and element connectivity, whose attribute names and data types are predefined by *Rocom*. The nodal coordinates are double-precision floating-point numbers, with three components per node. If the coordinates of a pane are stored contiguously, the storage can be registered using the name “nc”; otherwise, the x-, y-, and z- components must be registered separately using names “1-nc”, “2-nc”, and “3-nc”, respectively.

Rocom 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. Structured meshes have no connectivity tables, and the shape of a pane is registered using the name “st”. 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, registered using reserved keywords (such as “t3” or “t-3” for contiguous or staggered 3-node triangles, respectively). To facilitate parallel simulations, *Rocom* 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 attributes that have no designated names or data types. A

user must first define such an attribute in the window and then register the addresses of the attribute 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 user must register these arrays separately, similar to registering staggered nodal coordinates.

Window and Pane Attributes. A data member can also be associated with either the window or a pane. Examples of window attributes include a data structure that encapsulates the internal states of a module or some control parameters. An example of a pane attribute is an integer flag for the boundary condition type of a surface patch. Similar to field variables, these attributes do not have designated names or data types, and must be created within a window and then registered.

Aggregate Attributes. In *Rocom*, although attributes are registered as individual arrays, attributes 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 attributes in a window. For staggered attributes, one can use “*i*-attribute” ($i \geq 1$) to refer to the *i*th component of the attribute or use “attribute” to refer to all components collectively.

Aggregate attributes enable high-level inter-module interfaces. For example, one can pass the “all” attribute 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 *Rocom* allocate memory for data attributes and have application codes retrieve memory addresses from *Rocom*. *Rocom* provides a call for memory allocation, which takes a window-attribute name pair as input. A user can pass in “all” for the attribute name, which will have *Rocom* allocate memory for all the unregistered attributes.

4.3 Functions

A window can contain not only data members but also function members. A module can register a function into a window, to allow other modules to invoke the function through *Rocom*. 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 attributes into *Rocom*. Specifically, during registration a function can be specified as the member

function of a particular data attribute in a window. *Rocom* keeps track of the data attribute and passes it implicitly to the function during invocation, in a way similar to C++ member functions.¹ 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. *Rocom* 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. *Rocom* passes null pointers for those optional arguments whose corresponding actual parameters are missing during invocation.

4.4 Inheritance

In multiphysics simulations, inheritance of interface data is useful in many situations. First, the orchestration module 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 obtain a new window for extension or alteration, without altering the existing window. 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 orchestration module sometimes needs to split a user-defined window 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 3 depicts a scenario of inheritance among three windows.

To support these needs, *Rocom* allows inheriting the mesh from a parent window to a child window in either of two modes. First, inherit the mesh of the parent as a whole. Second, inherit only a subset of panes that satisfy a certain criterion, with the option to exclude the ghost nodes and cells of the parent from the child. 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 *multiple inheritance*). The child window obtains the data only in the panes it owns and ignores other panes. During inheritance, if an attribute already exists in a child window, *Rocom* overwrites the existing attribute with the new attribute.

Rocom 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 attribute 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.

4.5 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. We

¹A regular C++ member function cannot be registered with *Rocom*, due to the strong typing of C++. This limitation can be readily overcome by adapting regular member functions into static member functions whose first argument is a reference to the associated object.

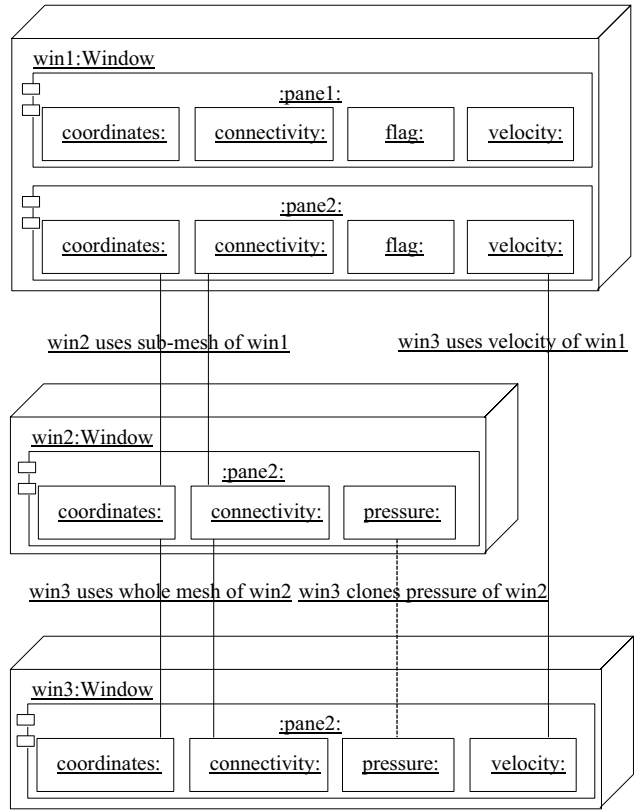


Figure 3: Scenario of inheritance of mesh and field attributes among three windows.

address these issues through the mechanisms of persistency and immutable references, respectively.

Persistency. *Rocom* maintains references to the datasets registered with its windows. To avoid dangling references associated with data registration, we impose the following persistency requirement: the datasets registered with a window must outlive the life of the window. This notion of persistency is simple and intuitive, and is widely used and considered the “preferred approach to implementing systems” in similar contexts such as object-oriented databases [8]. 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 system; instead, we consider it as a design pattern that application code developers should follow. Fortunately, typical physics codes allocate memory spaces during an initialization stage and deallocate memory during a finalization stage, which naturally adapts to this design pattern.

Immutable References. Another potential issue for data integrity is side effects due to inadvertent changes to datasets. To address this problem, some traditional object-oriented paradigms require that a client can change the state of a supplier object only through the supplier’s public interfaces. However, it has been noted that this integrity model is un-

necessarily restrictive for complex systems [6]. For the internal states of the modules, *Rocom* facilitates the traditional integrity model through member functions that we described earlier. For interface datasets, we adopt the concept of “deeply immutable references” [6] and enforce access control for immutable handles of data attributes. In *Rocom*, a service module can obtain accesses to another module’s data attributes only through its function arguments, and *Rocom* enforces at runtime that an immutable handle cannot be passed to mutable arguments. Furthermore, as we describe later, service modules of *Rocom* are implemented using a C++ interface that conforms to deeply immutable references at the language level, so *Rocom*’s application can be free of side effects with minimal runtime overhead.

5. ARCHITECTURE OF ROCCOM

The core of *Rocom* 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.

5.1 Rocom API

The *Rocom* API supplies a set of primitive function interfaces to physics and orchestration modules for system setup, window management, information retrieval, and function invocation. The subset of the API for window management serves essentially the same purpose as the Interface Definition Language (IDL) of other frameworks (such as CCA), except that *Rocom* parses the definitions of the windows at runtime. *Rocom* provides different bindings for C++ and F90, with similar semantics. In the following, we mention a few highlights of the API.

Data Management and Retrieval. The basic interface functions that all modules must use are the construction of windows and registration of data attributes. Figure 4 shows a sample F90 code fragment that creates a window with two panes. Typically, data registered in a window are accessed by service modules through C++ interfaces, which can enforce data integrity as discussed later. Some physics modules may also need to access a dataset through *Rocom*, for example, if an attribute was allocated by *Rocom* or was inherited from another window. To support this need, *Rocom* provides an API for retrieving information about panes and attributes, including the number of panes, the list of pane IDs, the numbers of nodes and elements in the panes, and the metadata of attributes. As an advanced feature, *Rocom* allows an F90 code to obtain the addresses of a dataset in *Rocom* through F90 pointers, which would then make the F90 code to assume ownership of the dataset. This feature enables the capability of managing memory spaces in C++ for F90 codes, which is convenient for developing some service utilities. Because ownership is transferred to the F90 code, data integrity is not compromised.

Function Registration and Invocation. A module registers a function with *Rocom* in a similar manner to registering window attributes. The arguments of a registered function can be pointers or references to primitive data types (such as integer, double, or char), or, more powerfully, pointers to Attribute objects, which we describe later. To reg-

```

INTEGER      :: nn1, nn2    ! sizes of nodes
INTEGER      :: ne1, ne2    ! sizes of elements
INTEGER      :: types(2)
INTEGER, POINTER :: conn1(3,ne1), conn2(4,ne2)
DOUBLE PRECISION, POINTER :: coors1(3,nn1), coor2(3,nn2)
DOUBLE PRECISION, POINTER :: disp1(3,nn1), disp2(3,nn2)
DOUBLE PRECISION, POINTER :: loc1(ne1,3), loc2(ne2,3)
EXTERNAL    fluid_update

CALL COM_CREATE_WINDOW("fluid")

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

! Create an element-centered double-precision dataset
CALL COM_NEW_ATTRIBUTE("fluid.loc", "e", COM_DOUBLE, -3, "m/s")

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

! Create a pane with ID 21 of a quadrilateral surface mesh
CALL COM_INIT_MESH("fluid.nc", 21, coors2, nn2)
CALL COM_INIT_MESH("fluid.q4", 21, conn2, ne2)

! Register addresses of data attributes for both panes
CALL COM_INIT_ATTRIBUTE("fluid.disp", 11, disp1)
CALL COM_INIT_ATTRIBUTE("fluid.loc", 11, loc1)
CALL COM_INIT_ATTRIBUTE("fluid.disp", 21, disp2)
CALL COM_INIT_ATTRIBUTE("fluid.loc", 21, loc2)

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

CALL COM_WINDOW_INIT_DONE("fluid")

```

Figure 4: Sample F90 code fragment for window registration.

ister a function, a module specifies a function pointer and the number, intentions (for input or output), and base data types of its arguments. For technical reasons, we impose a limit on the maximum number of the arguments that a registered function can take, but the limit can be adjusted, if desired, by a minor change to *Rocom*’s implementation.

Inter-module function invocation is done through *Rocom*, as demonstrated in Figure 5. *COM_call_function* takes the handle of the callee function, the number of arguments, and the actual arguments to be passed to the callee. If an argument of the callee is an Attribute object, the caller passes a reference to the handle of the attribute. This allows mixed-language interoperability. For data integrity, *Rocom* enforces that an immutable handle can be passed only to a read-only input argument. In a parallel setting, the invoked function will typically be executed on the same processor as the caller, supporting SPMD style parallelism. With a customized implementation of *COM_call_function*, *Rocom* can also allow the callee to run on different processors, enabling transparent remote procedure calls.

```

INTEGER      :: f_h        ! function handle
INTEGER      :: a1_h, a2_h, a3_h ! attribute handles

f_h = COM_GET_FUNCTION_HANDLE("Rocblas.add")
a1_h = COM_GET_ATTRIBUTE_HANDLE_CONST("fluid.nc")
a2_h = COM_GET_ATTRIBUTE_HANDLE_CONST("fluid.disp")
a3_h = COM_GET_ATTRIBUTE_HANDLE("fluid.loc")

! Compute loc=nc+disp
CALL COM_CALL_FUNCTION(f_h, 3, a1_h, a2_h, a3_h)

```

Figure 5: Sample F90 code fragment for function invocation.

Dynamic Loading of Modules. In the *Rocom* framework, each module can be built into a shared object, which is linked into the executable only at runtime. A dynamically loaded shared object facilitates plug-and-play of modules, and can also effectively avoid name-space pollution among modules, because such an object can have its own local name scope. *Rocom* accommodates dynamic loading by providing a `COM_load_module` function, which takes a module's name and a window name as arguments, and loads the shared object of the module using the dynamic linking loader `dlopen`. Each module provides an initialization routine `Module_load_module`, which constructs a window with a given name. *Rocom* tries to locate the routine using both the C and Fortran naming conventions and then invokes it following the corresponding calling convention. This technique further enhances transparency of C++/F90 interoperability.

5.2 C++ Class Interfaces

Rocom provides a unified view of the organization of distributed data objects for service modules through the abstractions of windows and panes. Internally, *Rocom* organizes windows, panes, attributes, functions, and connectivities into C++ objects, whose associations are illustrated in Figure 6, on a UML class diagram [14]. A Window object

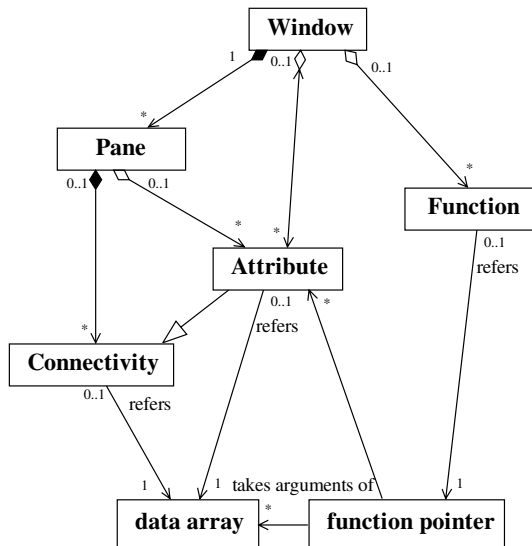


Figure 6: UML associations of *Rocom*'s classes.

maintains a list of its local panes, attributes, and functions; a Pane object contains a list of attributes and connectivities; an Attribute object contains a reference to its owner window. By taking references to attributes as arguments, a function can follow the links to access the data attributes 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.

5.3 Rocom Runtime System

The runtime system serves as the middleware between modules. It keeps track of the user-registered data and functions. During function invocation, it translates the function and attribute handles into their corresponding references with an efficient table lookup, enforces access protection of the attributes, 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, *Rocom* also provides tracing and profiling capabilities for inter-module calls to aid in debugging and performance tuning. It also exploits hardware counters through PAPI [9] 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. To further ease debugging, we have integrated the `malloc()` debugger Electric Fence (<http://perens.com/FreeSoftware>) into the framework. Electric Fence stops a program on the first instruction that causes a bounds violation, and requires no recompilation but only relinking of the executable. Together with the tracing capability, this tool makes it much easier to identify which module is responsible for a memory violation.

6. SERVICE COMPONENTS

To support rapid prototyping of various coupling schemes, we have identified a set of service utilities from our experience with the rocket simulation code.

Data Transfer. In multiphysics simulations with a partitioned approach, each physics module solves on its own mesh, and jump conditions are exchanged at surfaces between volume meshes. Therefore, it is necessary to transfer data between different surface meshes, and the criteria for data transfer include numerical accuracy and physical conservation. We have developed a software component, *Rocface*, which constructs a common refinement of nonmatching meshes [7], and supports various accurate and conservative data transfer schemes using this common refinement. *Rocface* supports multi-block structured and unstructured hybrid surface meshes, and can transfer node- or element-centered data. The main interface of *Rocface* for data transfer takes two key arguments, which are the Attribute objects of the source and target data. Through the framework, *Rocface* obtains the metadata of the data attributes and automatically invokes appropriate algorithms.

Numerical Operations. In multiphysics simulations, the jump conditions between physics modules frequently require mathematical manipulation. For example, in implicit coupling one must interpolate jump conditions in time, compute the norms for a convergence check, and store-restore of boundary conditions for predictor-corrector iterations. To support such operations, we have developed two numerical modules, *Rocblas* and *Rocmap*. The former provides some basic linear algebraic operations, such as addition, `saxpy`, and computation of norms. The latter is composed of a set of mesh-aware procedures, such as computation of face nor-

mals, nodal normals, cell volumes, and measures of element quality.

Parallel I/O. In numerical simulations, scientists writing a computation module simply want to write out a collection of data blocks to a file with a given name, and they see this as an atomic operation. *Rocomm* enables *Rocpanda* and *Rochdf* to encapsulate all lower-level I/O operations into high-level operations including `read_attribute` and `write_attribute` [10], operating on platform-independent files, which can be used for restart or visualized by our in-house visualization tool, *Rocketeer* (http://www.csar.uiuc.edu/F_software/rocketeer). Hidden under these one-step interfaces are file operations such as open, close, seek (used in reading), and data accesses. Because panes are distributed instances of a window, and all the panes must be read/written when I/O is performed, all the processes just need to invoke these high-level interfaces collectively on a window. They can use predefined *Rocomm* keywords “all” or “mesh” to read/write all the contents of the window, or mesh data only. They can also use individual attribute names to read/write selected data members in a window. Compared to available parallel I/O interfaces such as parallel HDF and MPI-IO, parallel I/O in our rocket simulation code has been greatly simplified from the users’ point of view.

7. APPLICATION TO ROCKET SIMULATION

Figure 7 shows a schematic UML sequence diagram of the interactions between the modules of our rocket simulation code. A *Rocomm* application has a driver or orchestration module, in this case *Rocman*, which is responsible for system setup and invoking the registered functions in turn. Each *Rocomm*-compliant module must provide a `load_module()` routine, which creates a 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 through *Rocomm*, the driver dynamically loads a set of modules into the runtime system. Through *Rocomm*’s calling mechanism, *Rocman* then invokes the functions of the physics and service modules, which in turn can also invoke functions provided by other modules.

Because *Rocman* needs to invoke service modules on behalf of the physics modules, it requires the privilege to obtain the attribute handles of the physics modules. To achieve better modularity, for each type of physics module *Rocman* provides an *agent*, which serves as the adapter of a physics module for interacting with other modules. The agent is a trusted “friend” of the physics module and has knowledge about the structure of the windows of that module. It provides a set of callback routines to that module, as described later.

7.1 Specification of Physics Modules

Besides `load_module()` and `unload_module()`, each physics module must provide three basic interface functions:

- `initialize(G, initialTime, communicator, callBack, ...)` allocates and initializes internal data structures and creates two windows (one for jump conditions and one for internal data to be stored/restored for restart and predictor-corrector iterations). The `callBack` arguments

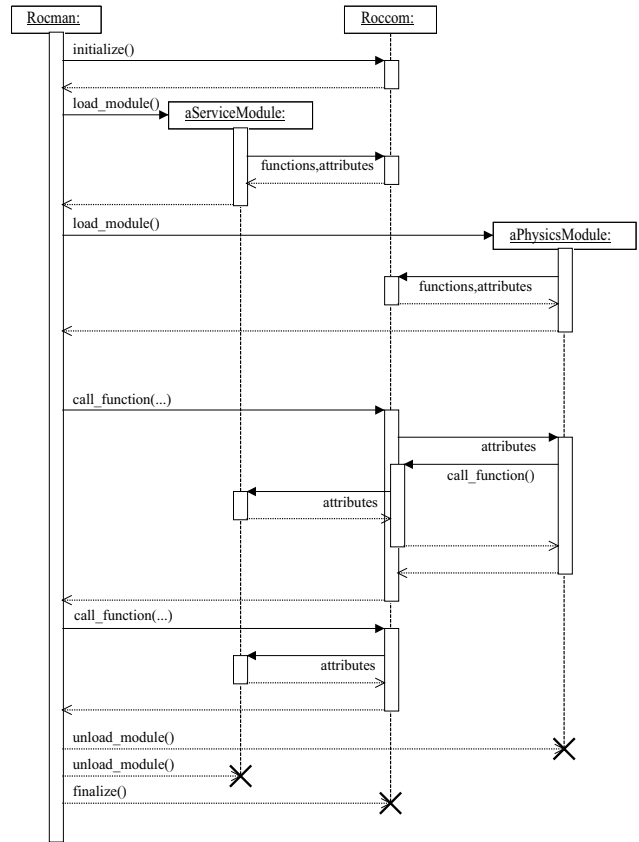


Figure 7: Interaction between modules shown on UML sequence diagram.

are integer handles of functions provided by its agent in *Rocman* for allocating buffer space and loading restart files.

- `update_solution(G, currentTime, timeStep, callBack, ...)` updates the physical solution for one system time step. It obtains jump conditions by calling *Rocman*-provided functions, which fill the module’s pre-registered input buffers. At the end of this subroutine, it updates the pre-registered outgoing buffers.
- `finalize(G)` deallocates memory and destroys the windows.

The first argument, *G*, of these routines is a context object encapsulating the internal states of a module. It is typically registered as a window attribute, of which the interface functions are declared as member functions. For verification and validation, a physics module can provide optional routines for sanity checking, such as computing the total volume, mass, surface area, etc.

The callback routines provided by *Rocman* are implemented using service modules in a data-structure independent fashion. They specify only high-level descriptions of jump conditions, so modules modeling the same physics but using different mesh types or data structures can invoke the same set of callback routines. To run a physics module stand-alone, it simply needs to supply and register its

own callback routines with *Rocom*. A user can easily control what modules to use by customizing the window names through the command line or configuration files.

7.2 Advantages of New Architecture

Earlier generations of our rocket simulation code were integrated without using *Rocom*. Compared with the old generations, the current version using *Rocom* and *Rocman* has the following notable advantages:

First, simpler inter-module interfaces. Without *Rocom*, each module must provide additional functions to store and restore its states for implicit coupling and provide their own I/O routines, unnecessarily exposing additional details of the coupling schemes to physics modules. With *Rocom*, such actions are provided opaquely to physics modules through *Rocman*.

Second, cleaner implementations of jump conditions. In earlier generations, jump conditions were embedded in the physics modules because they depend on specific mesh types and data structures. With *Rocman* and the data-structure independent service modules, the jump conditions are now implemented in a centralized and high-level fashion, enhancing both productivity and readability.

Third, easy switching of modules. Because of the flexible function registration and invocation mechanism, a user can easily select which modules to use for a given run. The ability of physics modules to run in stand-alone mode also enables progressive paths for integration and verification.

7.3 Performance Results

An indirect function invocation through *Rocom* is about two orders of magnitude more expensive (about $7.5\mu s$ on an IBM SP) than direct invocation of a function call ($15ns$ on an IBM SP), which is comparable with other frameworks, such as CCA [12]. The overhead of accessing the metadata of attributes through *Rocom* is also of about the same order. Because the granularity of computations in multiphysics simulations is usually relatively large (typically on the order of tens of milliseconds or higher), the overhead of data management and calling mechanism is negligible. In a parallel environment, *Rocom* itself does not incur spurious interprocess communication, and hence an integrated system should deliver good efficiency if the individual components are efficient.

To demonstrate the above claim of efficiency, we tested our GEN2.5 rocket simulation code for a scaled problem (fixed problem size per processor) on ASCI Frost, an IBM SP located at Lawrence Livermore National Laboratory. The machine has 64 POWER3 375 MHz 16-way SMP compute nodes running AIX 4.3, each with 16GB memory, connected by an SP Switch2. Our experiments used 15 processors per node. Figure 8 shows the wall clock times of the runs up to 960 processors. Our integrated system achieved excellent absolute performance (150Mflops per processor, which is about 10% of the theoretical peak of the machine) and demonstrated excellent scalability, showing that our framework incurs no performance penalty while boosting productivity and allowing quicker and more flexible prototyping of coupling schemes.

8. RELATED WORK

In recent years, several software frameworks have been developed for large-scale scientific applications, including Cac-

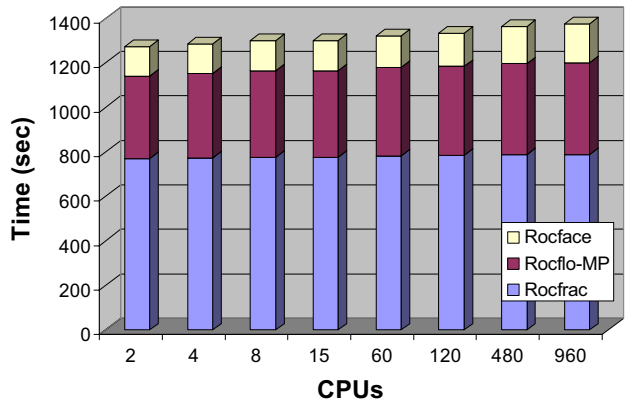


Figure 8: Wall clock times of GEN2.5 on IBM SP for scaled problems.

tus [1], CCA [2], Alegra [5], Overture [3], Pooma [13], and Sierra [15]. These frameworks share some similar objectives with *Rocom*, including extensibility, modularity, and code reuse. Unlike these other frameworks, *Rocom* has a unique object-oriented abstraction of interface data, which enables cleaner and simpler inter-module interfaces. On top of the abstraction, our framework provides a set of service components fine-tuned for quick integration of multiphysics simulations.

The Common Component Architecture (CCA) is a framework specification that allows a user to use CCA compliant software objects to build a scientific application. The communication mechanism of CCA is orthogonal to that of *Rocom*, in that CCA prohibits a component to expose its internal states to the outside world, and all inter-module interactions are through function interfaces. Such an approach provides more strict data protection, but unfortunately leads to lower-level and more sophisticated inter-module interfaces. Cactus shares more similarities with *Rocom* in its dynamic loading of modules and registration of the states of modules with the runtime system. However, Cactus provides no unified view of datasets and leaves data management to the modules.

Some other existing frameworks, including Alegra, Overture, Pooma, and Sierra, are based on a pervasive programming model, which requires the use of the framework-provided data structures and parallelization models, and in turn requires substantial modification to application codes. Like many other traditional frameworks, they are “designed for extension, not for integration” [11]. *Rocom* is unique in the respect that it distinguishes physics, orchestration, and service modules, provides machinery to meet the needs of each, and in turn minimizes changes to application codes and hides more details of the framework and services from application codes.

9. CONCLUSION

We have presented an object-oriented framework, *Rocom*, for interfacing modules in large-scale multiphysics simulations. Unlike other frameworks, *Rocom* concentrates on the abstraction and management of interface data, which enables simpler and cleaner inter-module interfaces. Using this abstraction, we have developed a set of service com-

ponents to support rapid and flexible prototyping of various coupling schemes, independently of the physics modules. Through the function invocation mechanism of *Roccom*, we provide plug-and-play capability for modules and also automatic tracing and profiling of functions. Because the granularity of the modules is generally large, in our experience with the rocket simulation code this methodology boosts productivity of system integration without sacrificing runtime performance. *Roccom* provides an alternative approach for component-based architectures. Although the data protection model of *Roccom* is less restrictive than some other architectures, through the design patterns of persistent objects and immutable references, *Roccom* can also enforce data integrity with minimal runtime overhead. As future work, we are developing other service utilities on top of *Roccom*, including error estimators and remeshing tools.

10. ACKNOWLEDGEMENTS

The authors would like to thank our many colleagues at CSAR, our long-term collaboration with whom has inspired the design of the framework presented in this paper. Among the service components, *Rocpanda* was developed by Xiaosong Ma; *Rocblas* and *Rocmap* were implemented with the help of Greg Mackey and Phil Alexander; the design of *Rocman* benefited from many contributions by Andreas Haselbacher and Prof. Philippe Geubelle. We thank Milind Bhandarkar, Jie Zheng, and Prof. Eric de Sturler for helpful discussions during early design stages of *Roccom*. We thank Mark Brandyberry for his useful comments on an early draft of the paper. We also express our gratitude to the anonymous referees, whose questions on data integrity inspired a more coherent presentation in the paper.

11. REFERENCES

- [1] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Proceedings of Supercomputing '01*, Nov. 2001.
- [2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, Nov. 1999.
- [3] F. Basseti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan. Overture: An object-oriented framework for high performance scientific computing. In *Proceedings of Supercomputing '98*, Nov. 1998.
- [4] M. Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395, Dec. 2000.
- [5] K. G. Budge and J. S. Peery. Experiences developing ALEGRA: A C++ coupled physics framework. In *Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Oct. 1998.
- [6] H. Hakonen, V. Leppanen, T. Raita, T. Salakoski, and J. Teuhola. Improving object integrity and preventing side effects via deeply immutable references. In *Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, FUSST'99*, pages 139–150, 1999.
- [7] X. Jiao and M. T. Heath. Efficient and robust algorithm for overlaying nonmatching surface meshes. In *10th International Meshing Roundtable*, pages 281–292, 2001.
- [8] B. Liskov, M. Castro, L. Shriram, and A. Adya. Providing persistent objects in distributed systems. *Lecture Notes in Computer Science*, 1628:230–257, 1999.
- [9] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, Aug. 2001.
- [10] X. Ma, X. Jiao, M. T. Campbell, and M. Winslett. Flexible and efficient parallel I/O for large-scale multicomponent simulations. In *The 4th Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*, Apr. 2003.
- [11] M. Mattsson, J. Bosch, and M. E. Fayad. Framework integration: Problems, causes, solutions. *Communications of the ACM*, 43:81–87, Oct. 1999.
- [12] J. Ray, H. N. Najm, and S. Lefantzi. CCA-component based simulation of flows on adaptively refined structured meshes, *SIAM Conference on Computational Science and Engineering*, Feb. 2003.
- [13] J. Reynders et al. POOMA: A framework for scientific simulations on parallel architectures. In G. Wilson and P. Lu, editors, *Parallel Programming Using C++*, pages 553–594, 1996.
- [14] P. Stevens and R. Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.
- [15] L. M. Taylor. Sierra: A software framework for developing massively parallel, adaptivity, multi-physics, finite element codes. In G. Wilson and P. Lu, editors, *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.