41st AIAA/ASME/SAE/ASEE Joint Propulsion Conference, July 10–13, 2005, Tucson, Arizona

An Integration Framework for Simulations of Solid Rocket Motors

X. Jiao* G. Zheng† O. Lawlor‡ P. Alexander§ M. Campbell§

M. Heath¶ and R. Fiedler‖

Center for Simulation of Advanced Rockets

University of Illinois, Urbana IL

Simulation of solid rocket motors requires coupling physical models and software tools from multiple disciplines, and in turn demands advanced techniques to integrate independently developed physics solvers effectively. In this paper, we overview some computer science components required for such integration. We package these components into a software framework that provides system support of high-level data management and performance monitoring, as well as computational services such as novel and robust algorithms for tracking Lagrangian surface meshes, parallel mesh optimization, and data transfer between nonmatching meshes. From these reusable framework components we construct *domain-specific* building blocks to facilitate integration of parallel, multiphysics simulations from high-level specifications. Through examples, we demonstrate the flexibility of our framework and its components.

I. Introduction

Many real-world systems involve complex interactions between multiple physical components. Examples include natural systems, such as climate models, as well as engineered systems, such as automobile, aircraft, or rocket engines. Simulation of such systems helps improve our understanding of their function or design, and potentially leads to substantial savings in time, money, and energy. However, simulation of multicomponent systems poses significant challenges in the physical disciplines involved, as well as computational mathematics and software systems.

At the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois, we have been developing a software system for detailed simulation of solid rocket motors. A rocket motor is a complex system of interactions among various parts—propellant, case, insulation, nozzle, igniter, core flow, etc.—involving a variety of mechanical, thermal, and chemical processes, materials, and phases. Many issues must be addressed in developing such a high-performance system, including

- easy-to-use software environment for exploiting parallelism within as well as between components
- reusable parallel software components and high-performance physics modules
- system tools for assessing and tuning performance of individual modules as well as the integrated system
- management of distributed data objects for inter-module interactions
- parallel computational methods, such as data transfer between disparate, distributed meshes

*Research Scientist

†Postdoctoral Research Associate

‡Current address: Department of Mathematics and Statistics, University of Alaska Fairbanks

§Research Programmer

¶Director; Professor, Computer Science; Director, Computational Science and Engineering

‖Technical Program Manager

In this paper we describe the software framework developed at CSAR for large-scale integrated rocket simulations. We provide a technical overview of the computational and computer science support for these rocket simulations, and complements the results on integrated simulation presented at this conference.[1,2] While presenting a comprehensive overview of computer science efforts, we focus mostly on our most recent developments on surface propagation and the high-level orchestration framework.

The remainder of the paper is organized as follows. Section II briefly overviews the *Rocstar* software system developed at CSAR. Section III presents our novel integration framework for multicomponent systems and some middleware services built upon it. Section IV describes a few key computational problems arising from the integration of such systems and our solutions to them. Section V describes the new high-level orchestration framework for the integrated rocket simulations, and Section VI concludes the paper with a discussion of some remaining challenges.

II. System Overview

Rocstar is our high-performance, integrated software system for detailed, whole-system simulation of solid rocket motors, currently under development at CSAR. We briefly overview the methodology and the software components of this system.

A. Coupling Methodology

Simulation of a rocket motor involves many disciplines, including three broad physical disciplines—fluid dynamics, solid mechanics, and combustion—that interact with each other at the primary system level, with additional subsystem level interactions, such as particles and turbulence within fluids. Because of its complex and cross-disciplinary nature, the development of *Rocstar* has been intrinsically demanding, requiring diverse backgrounds within the research team. In addition, the capabilities required from the individual physical disciplines are at the frontier of their respective research agendas, which entails rapid and independent evolution of their software implementations.

To accommodate the diverse and dynamically changing needs of individual physics disciplines, we have adopted a *partitioned* approach, to enable coupling of individual software components that solve problems in their own physical and geometrical domains. With this approach, the physical components of the system are naturally mapped onto various software components (or modules), which can then be developed and parallelized independently. These modules are then integrated into a coherent system through an integration framework, which, among other responsibilities, manages the distributed data objects, and performs inter-module communications on parallel machines.

B. System Components

To enable parallel simulations of rockets, we have developed a large number of software modules. Figure 1 shows an overview of the components of the current generation of *Rocstar*. These modules serve very diverse purposes and have diverse needs in their development and integration. We loosely group these modules into the following four categories.

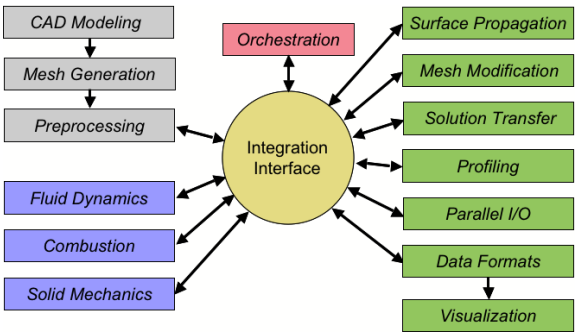

Figure 1. Overview of *Rocstar* software components.

Physics modules solve physical problems in their respective geometric domains. In general, they are similar to stand-alone applications, are typically written in Fortran 90, and use array based data structures encapsulated in derived data types.

Service modules provide specific service utilities, such as I/O, communication, and data transfer. They are typically developed by computer scientists but driven by the needs of applications, and are usually written in C++.

Integration interface provides data management and function invocation mechanisms for inter-module interactions.

Control (*orchestration*) *modules* specify overall coupling schemes. They contain high-level, domain-specific constructs built on top of service modules, provide callback routines for physics modules to obtain boundary conditions, and mediate the initialization, execution, finalization, and I/O for physics and service modules through the integration interface.

In *Rocstar*, the above categories correspond to the components at the lower-left, right, center, and top, respectively, of Figure 1. In the following sections, we describe various parallel aspects associated with these modules. In addition, our system uses some *off-line tools*, such as those in the upper-left corner of Figure 1, which provide specific pre- or post-processing utilities for physics modules.

III. Integration Framework and Middleware Services

To accommodate rapidly changing requirements of physics modules, we have developed a software framework that allows the individual components to be developed as independently as possible and integrated subsequently with little or no changes. It provides maximum flexibility for physics codes and can be adapted to fit the diverse needs of the components, instead of requiring the opposite. This framework is different from many traditional software architectures and frameworks, which typically assume that the framework is fully in control, and are designed for extension instead of integration.

A. Management of Distributed Objects

To facilitate interactions between modules, we have developed an object-oriented, data-centric integration framework called *Roccom*. Its design is based on an important observation and assumption of *persistent objects*. An object is said to be *persistent* if it lasts beyond a major coupled simulation step. In a typical physics module, especially in the high-performance regime, data objects are allocated during an initialization stage, reused for multiple iterations of calculations, and deallocated during a finalization stage. Therefore, most objects are naturally persistent in multipcomponent simulations.

Based on the assumption of persistence, *Roccom* defines a registration mechanism for data objects and organizes data into distributed objects called *windows*. A window encapsulates a number of *data attributes*, such as the mesh (coordinates and connectivities) and some associated field variables. 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. A module constructs windows at runtime by creating attributes and registering their addresses. Different modules can communicate with each other only through windows, as illustrated in Figure 2.

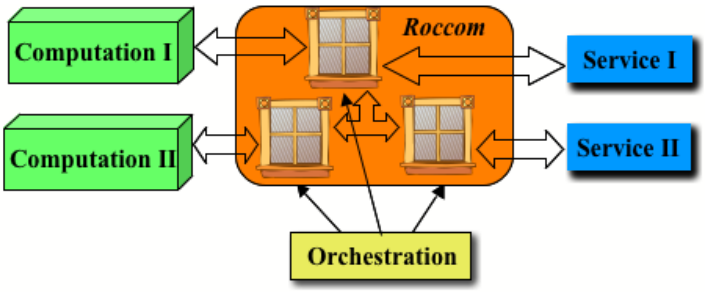

Figure 2. Schematic of windows and panes.

Roccom also introduces the novel concept of *partial inheritance* of windows to construct a sub-window

by *using* or *cloning* a subset of the mesh or attributes of another window. In addition, the registered attributes in *Roccom* can be referenced as an aggregate, such as using "mesh" to refer to the collection of nodal coordinates and element connectivity. These advanced features enable performing complex tasks, such as reading or writing data for a whole window, with only one or two function calls.[3]

On top of *Roccom*, we have developed a number of reusable service modules, including *middleware* services, such as communication and performance tools, which we describe in the following subsections, as well as mathematical services, which we discuss in the next section.

B. Interpane Communication

Traditional message-passing paradigms typically provide general but low-level inter-process communications, such as send, receive, and broadcast. In physical simulations using finite element or finite volume methods, communications are typically across panes or partitions, whether the panes or partitions are on the same or different processes. The *Roccom* framework provides high-level inter-pane communication abstractions, including performing reductions (such as sum, max, and min operations) on shared nodes, and updating values for ghost (i.e., locally cached copies of remote values of) nodes or elements. Communication patterns between these nodes and elements are encapsulated in the *pane connectivity* of a window, which can be provided by application modules or constructed automatically in parallel using geometric algorithms. These inter-pane communication abstractions simplify parallelization of a large number of modules, including surface propagation and mesh smoothing, which we will discuss in the next section.

C. Performance Monitoring

There is a wide variety of tools for the collection and analysis of performance data for parallel applications. However, the use of external tools for performance tuning is too laborious a process for many applications, especially for a complex integrated system such as *Rocstar*. Many tools are not available on all the platforms for which we wish to collect data, and using a disparate collection of tools introduces complications as well. To obviate the need for external tools, we have extended the *Roccom* framework's "service-based" design philosophy with the development of a performance-profiling module, *Rocprof*.

Rocprof provides two modes of profiling: module-level profiling and submodule-level profiling. The module-level profiling is fully automatic, embedded in *Roccom*'s function invocation mechanism, and hence requires no user intervention. For submodule-level profiling, *Rocprof* offers profiling services through the standard MPI_Pcontrol interface, as well as a native interface for non-MPI based codes. By utilizing the MPI_Pcontrol interface, applications developers can collect profiling information for arbitrary, user-defined sections of source code without breaking their stand-alone codes. Internally, *Rocprof* uses PAPI[4] or HPM (http://www.alphaworks.ibm.com/tech/hpmtoolkit) to collect hardware performance statistics.

IV. Parallel Computational Methods

In *Rocstar*, a physical domain is decomposed into a volume mesh, which can be either block-structured or unstructured, and the numerical discretization is based on either a finite element or finite volume method. The interface between fluid and solid moves due to both chemical burning and mechanical deformation. In such a context, we must address a large number of mathematical issues, three of which we discuss here.

A. Surface Propagation

In *Rocstar*, the interface must be tracked as it regresses due to burning. In recent years, Eulerian methods, especially level set methods, have made significant advancements and become the dominant methods for moving interfaces.[5,6] In our context, Lagrangian representation of the interface is crucial to describe the boundary of volume meshes of physical regions. However, preexisting numerical methods, either Eulerian or Lagrangian, have difficulties in capturing the evolving singularities (such as ridges and corners) in solid rocket motors.

To meet this challenge, we have developed a novel method, called *face-offsetting methods*,[7] based on a new *entropy-satisfying Lagrangian formulation*. Our face-offsetting methods deliver an accurate and stable entropy-satisfying solution without requiring Eulerian volume meshes. A fundamental difference between face-offsetting and traditional Lagrangian methods is that our methods solve the Lagrangian formulation

face by face, and then reconstruct vertices by constrained minimization and curvature-aware averaging, instead of directly moving vertices along some approximate normal directions. This method allows part of the surface to be fixed or to be constrained to move along certain directions (such as constraining the propellant to burn along the case). It supports both structured and unstructured meshes, with an integrated node redistribution scheme that suffices to control mesh quality for moderately moving interfaces. Figure 3 shows the propagation of a block-structured surface mesh for the fluids domain of the Attitude Control Motor (ACM) rocket, where the front and aft ends burn along the cylindrical case.

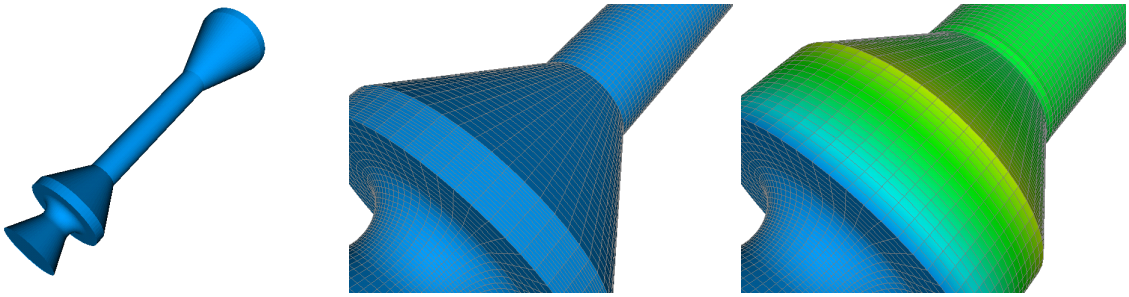

Figure 3. Simulation of burning of Attitude Control Motor along the case with block-structured meshes using face offsetting. Left subfigure shows initial geometry; middle and right subfigures show meshes of initial geometry and after 30% burn, respectively. Colors indicate magnitude of total displacements of vertices.

When coupled with mesh adaptation, the face-offsetting method can capture significant burns. Figure 4 shows a sample result of the burning of a star grain section of a rocket motor using the face offsetting method coupled with surface remeshing using MeshSim from Simmetrix (http://www.simmetrix.com). The interior (the fins) of the propellant burns at uniform speed and exhibits rapid expansion at slots and contraction at fins. The fin tips transform into sharp ridges during propagation, as captured by the face offsetting method.

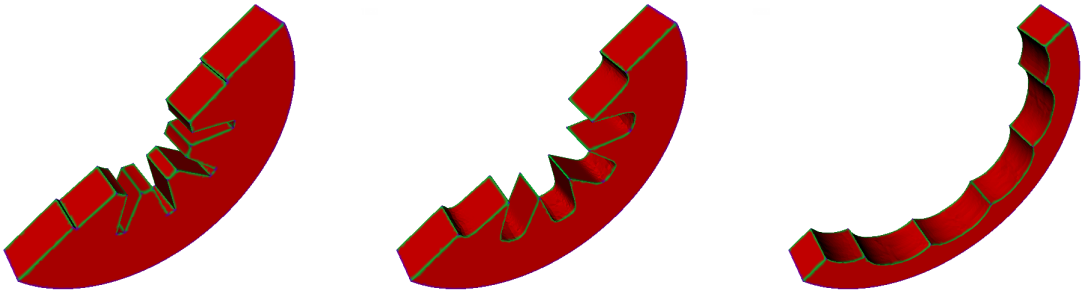

Figure 4. Simulation of uniform burning of section of star grain of solid rocket using face offsetting and mesh repair. Green curves indicate ridges in evolving geometry.

B. Mesh Optimization

In *Rocstar*, each physics module operates on some type of mesh. An outstanding issue in integrated rocket simulations is the degradation of mesh quality due to the changing geometry resulting from consumption of propellant by burning, which causes the solid region to shrink and the fluid region to expand, and compresses or inflates their respective meshes. This degradation can lead to excessively small time steps when an element becomes poorly shaped, or even outright failure when an element becomes inverted. Some simple mesh motion algorithms are built into our physics modules. For example, simple Laplacian smoothing is used for unstructured meshes, and a combination of linear transfinite interpolation (TFI)[8] with Laplacian smoothing is used for structured meshes in *Rocflo*. These simple schemes are insufficient when the meshes undergo major deformation or distortion. To address this issue, we take a three-tiered approach, in increasing order of aggressiveness: mesh smoothing, mesh repair, and global remeshing.

Mesh smoothing copes with gradual changes in the mesh. We provide a combination of in-house tools and integration of external packages. Our in-house effort focuses on parallel, feature-aware surface mesh

optimization, and provides novel parallel algorithms for mixed meshes with both triangles and quadrilaterals. To smooth volume meshes, we utilize the serial MESQUITE package[9] from Sandia National Laboratories, which also works for mixed meshes, and we parallelized it by leveraging our across-pane communication abstractions.

If the mesh deforms more substantially, then mesh smoothing becomes inadequate and more aggressive mesh repair or even global remeshing may be required, although the latter is too expensive to perform very frequently. For these more drastic measures, we currently focus on only tetrahedral meshes, and leverage third-party tools off-line, including Yams and TetMesh from Simulog and MeshSim from Simmetrix, but we have work in progress to integrate MeshSim into our framework for on-line use. Remeshing requires that data be mapped from the old mesh onto the new mesh, for which we have developed parallel algorithms to transfer both node- and cell-centered data accurately, built on top of the parallel collision detection package developed by Lawlor and Kalé.[10] Figure 5 shows an example where the deformed star grain is remeshed with the temperature field of the fluids volume transferred from the old to the new mesh.

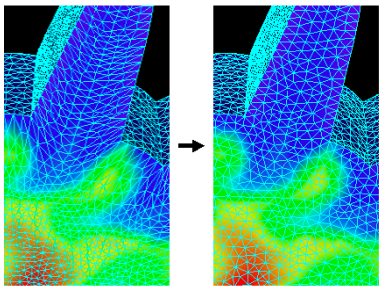

Figure 5. Example of remeshing and data transfer of deformed star grain.

C. Intermodule Data Transfer

In multiphysics simulations, the computational domains for each physical component are frequently meshed independently, which in turn requires geometric algorithms to correlate the surface meshes at the common interface between each pair of interacting domains to exchange boundary conditions. These surface meshes in general differ both geometrically and combinatorially, and are also partitioned differently for parallel computation. To correlate such interface meshes, we have developed novel algorithms to constructs a *common refinement* of two triangular or quadrilateral meshes modeling the same surface, that is, a finer mesh whose polygons subdivide the polygons of the input surface meshes.[11] To resolve geometric mismatch, the algorithm defines a *conforming homeomorphism* and utilizes locality and duality to achieve optimal linear time complexity. Due to the nonlinear nature of the problem, our algorithm uses floating-point arithmetic, but nevertheless achieves provable robustness by identifying a set of consistency rules and an intersection principle to resolve any inconsistencies due to numerical errors.

After constructing the common refinement, we must transfer data between the nonmatching meshes in a numerically accurate and physically conservative manner. Traditional methods, including pointwise interpolation and some weighted residual methods, can achieve either accuracy or conservation, but none could achieve both simultaneously. Leveraging the common refinement, we developed more advanced formulations and optimal discretizations that minimize errors in a certain norm while achieving strict conservation, yielding significant advantages over traditional methods, especially for repeated transfers in multiphysics simulations.[12] For parallel runs, the common refinement also contains the correlation of elements across partitions of different meshes, and hence provides the communication structure needed for inter-module, inter-process data exchange.

V. Orchestration Framework

Coupled rocket simulations involve interactions of physics modules including fluids, solids, and combustion. These individual physics modules may need to proceed at their respective time steps, while the boundary (or jump) conditions must be exchanged periodically among them to conduct a coherent simu-

lation. The numerical coupling algorithms for these simulations frequently also require customization to accommodate needs of different applications.

These coupled simulations pose a number of challenges and complexities in orchestrating the interactions among physics modules. First, the orchestration module must provide a flexible and easy construction of complex coupling algorithms and provide readability of the orchestration module, so that an end-developer can experiment with novel coupling algorithms easily. Second, the data transfer between physics modules may impose certain causality constraints and also may require buffer data for manipulating the jump conditions. This in turn requires sophisticated task and buffer management of the orchestration module. Finally, the orchestration module must provide a simple interface for the numerical and geometrical services utilities, including mesh modification, intermodule data transfer, and surface propagation.

To meet these challenges, we have developed *Rocman* — the control and orchestration module to coordinate multiple physics modules in coupled simulations and provide facilities to extend and implement new coupling schemes. It is the front-end of the coupled code that directly interacts with end-developers of coupled simulations. *Rocman* is a high-level infrastructure, built on top of the *Roccom* integration framework. With a novel design using the idea of action-centric specification and automatic scheduling of reusable actions to describe the intermodule interactions, *Rocman* facilitates the diverse needs of different applications and coupling schemes in an easy-to-use fashion.

A. *Rocman* Components

Rocman contains five types of key components: top-level iterations, agents for physics modules, actions, schedulers, and coupling schemes.

- One of the major tasks of *Rocman* is to drive the simulation. For this purpose, it provides *top-level iterations* including *time-marching schemes* for both steady and unsteady-state calculations. In the driver code, *Rocman* invokes time integration of the coupling scheme by passing in the current time and obtaining a new time, until the system reaches a designated time or a converged state.

- An *agent* serves a physics module. It represents a domain-specific simulation (fluid, solid, or combustion) in a coupling scheme. The most basic task of an agent is to initialize the physics module and manage its persistent buffer data for use during intermodule interactions on behalf of the physics module.

- Interactions between physics modules are encapsulated in *actions*. An action is a functional object implementing a designated calculation. An action also defines the input data, on which it operates and the output data produced by the calculation.

- A *scheduler* is a container of actions, and is responsible for determining the orders of initialization, execution, and finalization of its actions. A scheduler provides a procedure `add_action()` to its user for registering actions. After all the actions have been registered with a scheduler, the scheduler can then automatically schedule these actions based on the data flow among actions. The automatic scheduling constructs a call graph, which is a directed acyclic graph (DAG) for the actions, in which each edge between a pair of actions is identified by the data passing from one action to the other. This automatic scheduling of actions greatly simplifies the work of an end-developer, who now needs to be concerned about only the data movement among actions without having to worry about the order of its execution. Furthermore, constructing a call graph of actions exposes parallelism among actions and potentially enables concurrent execution of all independent actions that have their input data ready. In the future, we plan to extend the run-time scheduling to allow concurrent execution of actions.

- A *coupling scheme* is composed of a number of agents and a scheduler. The scheduler determines the orders that must be followed for invoking initialization, execution, and finalization of agents and actions. The coupling scheme is the only code an end-developer of a new coupling scheme needs to write. *Rocman* provides a rich set of predefined basic actions, which can then be used as building blocks for new coupling schemes.

B. Coupling Scheme Visualization

Understanding and debugging a complex coupling scheme poses a great challenge for a user when a variety of *schedulers* and *actions* are involved. *Rocman* provides a visualization tool that displays the data flow of actions to help users comprehend and debug coupling schemes. When a coupling scheme is constructed, an output file is generated that describes the coupling scheme and its schedulers and actions in the Graph Description Language (GDL). The output file can then be visualized by tools such as AiSee.[13]

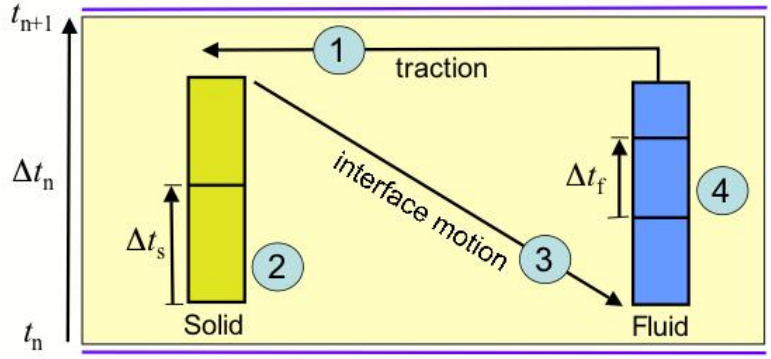

Figure 6. Illustration of simplified time stepping scheme for fluid-solid interaction.

As a concrete example, Figure 6 illustrates a simplified fluid and solid coupling scheme with subcycling of individual physics modules. In a "system time step", the tractions are first transferred from the fluids interface mesh onto the solids interface mesh (step 1), and a finite-element analysis of elasticity is then performed to compute the displacements of the interface (step 2). During the process, the solids module may perform multiple smaller time steps based on its stability limit, and obtain jump conditions (tractions) from *Rocman*, which performs interpolation in time. After the solids module reaches the designated system time step, *Rocman* transfers the displacements of the interface (step 3). The fluids module then solves for tractions by obtaining mesh motion and solids velocity as boundary conditions (step 4).

Figure 7 shows the visualization of this simplified coupling scheme. In the graph, each node represents an *action* or a *scheduler* (a container of actions), corresponding to the steps in the above description of the coupling scheme. Each edge represents the execution order of actions and is labeled with data passed between actions. This figure was generated automatically using the GDL output of *Rocman*, except for the circled numbers which were added manually. A scheduler node can be unfolded in AiSee graph viewer to reveal the details of the actions that the scheduler contains. This visualization capability helps development of new coupling schemes by allowing them to be debugged visually at a high level.

VI. Conclusion

In this paper, we provided an overview of some software and algorithmic issues in developing high-performance simulation tools for multicomponent systems. We described the data management, middleware services, and some computational tools, with focus on recent developments of surface propagation and high-level orchestration module. We presented examples of individual computational modules and the orchestration module to demonstrate the functionality and flexibility of the framework. Examples of coupled physical simulations can be found in companion papers.[1,2] While substantial progress has been made, many challenges remain, such as parallel mesh repair and adaptation.

Acknowledgments

We thank many of our colleagues at CSAR, especially Prof. Philippe Geubelle, Drs. Andreas Haselbacher, Luca Massa, Ali Namazifard, and Bono Wasistho for their input on the *Rocman* framework. The CSAR research program is supported by the U.S. Department of Energy through the University of California under subcontract B523819.

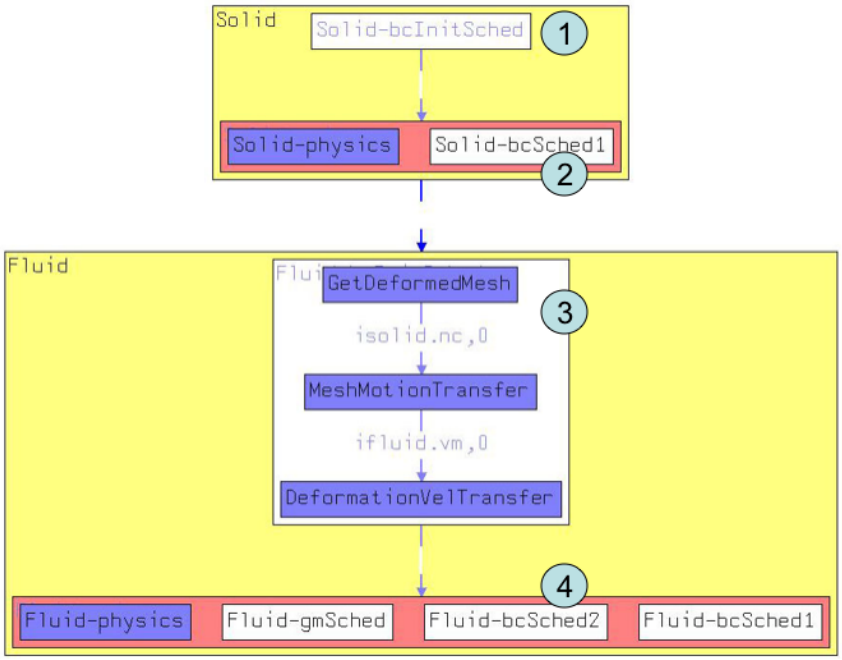

Figure 7. Sample visualization of fluid-solid coupling scheme using aiSee.

References

[1]Fiedler, R. A., Haselbacher, A., Breitenfeld, M. S., Alexander, P., Massa, L., and Ross, W. C., "3-D Simulations of Ignition Transients in the RSRM," *41st AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, 2005, AIAA-2005-3993.

[2]Brandyberry, M. D., Fiedler, R. A., and McLay, C., "Verification and Validation of the Rocstar 3-D Multi-physics Solid Rocket Motor Simulation Program," *41st AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, 2005, AIAA-2005-3992.

[3]Jiao, X., Campbell, M. T., and Heath, M. T., "Roccom: An Object-Oriented, Data-Centric Software Integration Framework for Multiphysics Simulations," *17th Ann. ACM Int. Conf. Supercomputing*, 2003, pp. 358–368.

[4]Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P., "A Portable Programming Interface for Performance Evaluation on Modern Processors," *Int. J. High Perf. Comput. Appl.*, Vol. 14, 2000, pp. 189–204.

[5]Osher, S. and Fedkiw, R., *Level Set Methods and Dynamic Implicit Surfaces*, Springer, 2003.

[6]Sethian, J. A., *Level Set Methods and Fast Marching Methods*, Cambridge University Press, 1999.

[7]Jiao, X., Heath, M. T., and Lawlor, O. S., "Face-Offsetting Methods for Entropy-Satisfying Lagrangian Surface Propagation," In preparation.

[8]Thompson, J. F., Soni, B. K., and Weatherill, N. P., editors, *Handbook of Grid Generation*, CRC Press, Boca Raton, 1999.

[9]Freitag, L., Leurent, T., Knupp, P., and Melander, D., "MESQUITE Design: Issues in the Development of a Mesh Quality Improvement Toolkit," *8th Intl. Conf. Numer. Grid Gener. Comput. Field Sim.*, 2002, pp. 159–168.

[10]Lawlor, O. S. and Kalé, L. V., "A Voxel-Based Parallel Collision Detection Algorithm," *Proc. Internat. Conf. Supercomputing*, June 2002, pp. 285–293.

[11]Jiao, X. and Heath, M. T., "Overlaying Surface Meshes, Part I: Algorithms," *Int. J. Comput. Geom. Appl.*, Vol. 14, 2004, pp. 379–402.

[12]Jiao, X. and Heath, M. T., "Common-Refinement Based Data Transfer Between Nonmatching Meshes in Multiphysics Simulations," *Int. J. Numer. Meth. Engrg.*, Vol. 61, 2004, pp. 2401–2427.

[13]"aiSee Graph Layout Software," http://www.aisee.com.