

FACETS – A Multiphysics Parallel Component Framework

Svetlana Shasharina
Tech-X Corporation
5621 Arapahoe Ave Suite A
Boulder, CO 80303
1 (720) 563 0322
sveta@txcorp.com

Gregory R. Werner
University of Colorado
5621 Arapahoe Ave Suite A
Boulder, CO 80309
1 (303) 735 2461
greg.werner@colorado.edu

John R. Cary
Tech-X Corporation
5621 Arapahoe Ave Suite A
Boulder, CO 80303
1 (303) 448 0728
cary@txcorp.com

Scott Kruger
Tech-X Corporation
5621 Arapahoe Ave Suite A
Boulder, CO 80303
1 (720) 974 1841
kruger@txcorp.com

Ammar Hakim
Tech-X Corporation
5621 Arapahoe Ave Suite A
Boulder, CO 80303
1 (303) 444 2451
ammar@txcorp.com

Alexander Pletzer
Tech-X Corporation
5621 Arapahoe Ave Suite A
Boulder, CO 80303
1 (303) 996 2031
pletzer@txcorp.com

ABSTRACT

Coupling of separately developed codes offers an attractive method for increasing the accuracy and fidelity of the computational modeling. Examples include the earth sciences and fusion integrated modeling. This paper describes the architecture of FACETS (Framework Application for Core-Edge Transport Simulations). As is common in scientific application development, FACETS has not adopted any particular previously existing framework. However, it has adopted the component concept and integrated pieces of existing frameworks. This paper describes how this approach addresses the challenges of coupling multiple physics models; including flexible algorithms, parallelism, mixed languages and legacy applications integration.

Categories and Subject Descriptors

D.3.3: Language Constructs and Features – *frameworks*. J.2: Physical Sciences and Engineering – *physics*.

General Terms

Design, Standardization, Languages.

Keywords

Fusion Simulation Project, FACETS, Components, Frameworks.

1. FUSION SIMULATION PROJECT AND FACETS

Faced with multiple time and spatial scales of fusion plasmas, most computational efforts traditionally concentrated on solving partial differential equations applicable to a distinct range of time and spatial scales. Multiple independent codes, RF (Radio-Frequency), Gyrokinetic, MHD (Magnetohydrodynamics), and transport cover ranges of time scales from 10^{-10} to 10^4 sec. With the advance of new fusion devices, such as the upcoming International Thermonuclear Experimental Reactor (ITER), fusion theory will need to provide predictive full physics modeling for planning, interpretation, and optimization of the operation of devices. Concurrently, computing infrastructure is now ready to support the massive simulations required.

The US Department of Energy, realizing the challenge of full-device and multiphysics modeling [1], has funded three SciDAC integration projects [1, 2-4] that are addressing different computer science and physics aspects of this problem. FACETS is about integration of core-edge-wall modeling (see Fig. 1) with the

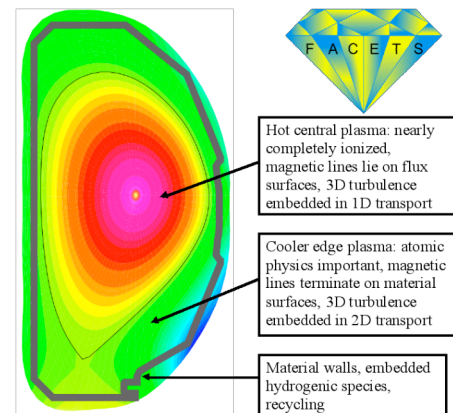


Figure 1. FACETS goal is tight-coupling framework for core-edge-wall.

conditions in the core, which determine fusion energy production, being strongly influenced by the boundary conditions imposed by the edge region, which in turn is influenced by its interaction with the wall, which can store and release hydrogenic species, ultimately determining plasma density. Each of these regions has rapid internal equilibration time scales (μ s or less), much shorter than the system modeling time (1000 s) over which the plasma discharge lasts. This leads to the need for implicit time advance of the combined system. The resulting tight coupling loop dictates in-memory intercomponent communication rather than through files. Furthermore, effective use of massively parallel, leadership class facilities dictates that components should be run on different processor subsets.

2. COMPONENT FRAMEWORK AND TECHNOLOGY ADOPTION

Multiphysics modeling is naturally addressed through use of component technology. In this approach, each of the physics units

just described is represented by a computational component, i.e. an entity with a well-defined interface. To combine these components into an application one must have registries for abstract interfaces through which components will interact, implementations and instances to allow runtime component creation and discovery, and a means to specify the composition. This software comprises the framework *superstructure*. Additional utility software may be needed to develop new components. This software should provide domain specific utilities for such tasks as messaging, I/O, meshing, and domain decomposition. This software comprises the framework *infrastructure*. This separation is not perfect, as newly developed components may want to rely on containers of the superstructure, while the superstructure may want to use the messaging and I/O utility software. Regardless, the *Framework* is the combination of infrastructure and superstructure that allows one to model the multiphysics system.

The “build or buy” decision is critical [5] in any computational project. Such decisions must be made in three different areas: infrastructure, superstructure, and computational components. The decisions are based on the capability provided, risk, and ability to mitigate risk. If a particular technology ends up providing only a small amount of what is needed for some part of the framework, then it can be easier simply to build, so that one has entire control of that part, and one avoids the complexities associated with external dependencies. If associated with adoption is considerable risk, and that risk cannot be mitigated, then computational scientists, who might lose several years of career development from being involved in a failed project, must demur.

For infrastructure, such decisions are relatively easier, as they can be isolated to tasks of the project, and the risk associated with adoption (loss of control, disappearance of the technology) is smaller. For example, adoption of MPI, an established technology with proven track record and many implementations, is of little risk. For any technology, one can mitigate the risk by wrapping and coding to an interface that could be reimplemented if the adopted technology disappears. Thus, FACETS has made extensive use of existing technologies, including MPI, PETSc, Babel, TAU, VisIt, and others, in some cases through wrapping.

The decision about adopting a superstructure is much more problematic. First, accepting part of a superstructure is difficult; one must usually accept it wholly or not at all. Second, superstructures tend to want to be in charge of `main()`, and this make difficult isolation of the associated risk. Third, it is difficult to find superstructures that address the major issues of a domain-specific application, such as: What are the correct interfaces for particular classes of computational physics components? How does one have a component construction process that allows for generality along with tight coupling, where direct memory access into a component may be necessary. What language can describe the constructions of a particular computation, including the physics, algorithms, and parallel distribution. Upon consideration of these issues, the FACETS decision was to develop its own framework superstructure.

The decision to adopt or redevelop computational components brings yet another set of issues. Adoption of existing (“legacy”) components may yield faster time to deployment, as one can reuse proven software, and validating scientific software is a time consuming process. On the other hand, adoption requires at minimum wrapping the existing application so that it can be used within the framework. In addition, it may be necessary to provide

robust (or any) parallelization, bring the component up to modern software engineering standards, and even rewrite parts to deal with supercomputers (e.g., a python application may not work on the compute nodes due to lack of shared libraries). Ultimately, the multiphysics application team must make a hard-nosed assessment of the modifications needed to bring legacy software into a modern framework. This can be difficult in the face of decades of personal investment by some in a particular piece of legacy software.

If, in the end, one does decide on redevelopment, one should assess what can be gained by extracting modules from any existing legacy components. For example, it might be possible to easily extract modules that compute fundamental interactions, such as atomic cross-sections, and reuse those in the development of new implementations. FACETS has made effective use of this approach. Recognizing that the core solver had to be reimplemented because there existed no parallel core solver, FACETS developed a new core solver while simultaneously extracting and wrapping components for calculation of the turbulent fluxes needed by the core solver.

To conclude this section, with regard to the framework and technology adoption, FACETS is adopting as much infrastructure as it can with minimal risk, while developing any other infrastructure that it needs. Adopted infrastructure includes tools for messaging, linear and nonlinear solves, inter-language operability, and performance analysis. Given the risk associated with adopting a superstructure framework, the difficulty in mitigating such risk, and the fact that such frameworks may bring minimal assistance in dealing with the difficult issues such as implementation validity, FACETS is building its own superstructure. FACETS is adopting much infrastructure from other projects and building some of its own. The FACETS superstructure and infrastructure derive from the earlier VORPAL framework [6], which can be used to construct flexible physics modeling applications. Finally, FACETS is dealing with computational components on a case-by-case basis, redeveloping when necessary and otherwise maximizing reuse.

3. FACETS ARCHITECTURE

The goal of the FACETS framework is to allow tight coupling of multiple dynamic components running in parallel (sequentially and concurrently). This leads to multiple requirements. Here we discuss further the following subset:

- Ability to incorporate legacy codes as components
- Ability to build new components using the infrastructure
- Construction process that allows for direct memory access
- Separation of algorithms from data
- Language for defining the simulation
- Run time discoverable implementations and instances

3.1 External components

External components are those that were developed prior to the FACETS project and that do not use the FACETS infrastructure. They have to be adapted to the basic FACETS component interface, which is composed of three parts – startup, advance and dump/restore interfaces.

The startup interface is designed to break up actions in a way that allows the component to read a configuration file, know what processors it is working on, expose different interfaces on different processors, allocate memory, and then fill that memory from either a default data or data saved from at the end of the last run. The

dump/restore interfaces allows dumping all the needed (for restore) data into a file and a file node and restore the state from the file or a file node.

The interface for the advance phase allows one to advance the state of a component in time as prescribed by their standalone physics behavior and to set and get data exchanged between components. If the advance of any component fails, such as due to solution divergence, a component can be reset to the last valid state. This advance phase interface was written assuming that external components are being used for substantial computations, so that the overhead of getting and setting data by name is not significant. It was also written with the first target, core-edge coupling, in mind. For that target the separate components need send and receive only a few scalars. This is now being generalized as we move to more data intensive interactions.

3.2 Internal components, direct memory access, and algorithm separation

Internal components (newly developed for FACETS) are built on the FACETS infrastructure, which provides extensive libraries for messaging, reading and writing data, grids, data structures, interpolation, component composition, etc. With this limited space, we will concentrate on a few salient aspects of internal components.

Components are nested. They have a reference to their containing *parent* and hold their *children*. Thus, a FACETS simulation may contain a core and an edge. The core contains sources and turbulent fluxes. The sources may contain geometry, and so on.

FACETS enforces a separation of data and algorithms that advance the data in time – *updaters* in the FACETS language. The implementation of the advance method of internal components is delegated to updaters, which contain direct references to the data structures of components and manipulate them to dynamically advance the state of a component. To provide this access, the interface of the internal component is expanded by a method called `buildSolver`. This method is called after memory allocation of all objects, and before initialization or restoration of any object. At this time, one object can acquire direct access to the memory locations of the data of another object, so that at advance time, interface overhead disappears.

Different updaters implement different algorithms and each component can have a complicated sequence of updaters. Thus, the user can specify that the data structures representing the profiles of density and temperature be updated using a Crank-Nicholson algorithm or some other, non-time-centered algorithm. Or the user can specify that the core sources are computed by combining power from two particular neutral beam sources and electron-cyclotron resonance heating. Finally, the update step order is used to combine the various updaters in a particular order. If any update returns a failure code, then all components are reset to their last valid step, and the step is retried with a smaller time step, at which convergence should be easier.

Coupling of components in FACETS is also performed by special updaters. In a coupled situation, the coupled components are contained within a parent component that might prescribe when to exchange data at every time step or impose some kind of criteria (full convergence for example) before coupling. The coupling updater belongs to the parent component and specifies names of

data being passed. Then appropriate methods of the advance interfaces (set and get by name) are then called.

3.3 Composition language

The FACETS input file describes the simulation composition. As always, the input file has to contain the parameters needed to describe the simulations. But for a flexible application as described above, the input file must also describe (1) the containment hierarchy, (2) the other objects needed a particular object for its advance. For this purpose FACETS developed a simple XML-like language with allows certain tags, setting global constants and a means to describe numerical vectors.

The bulk of the input file is defining the simulation components (their data structures, grids and updaters). In addition, input file describes how coupling is performed. For example, the following input file defines a coupled core-edge simulation. It starts with defining a top parent component `facets` containing two children `core` and `edge` of kinds `coreComponent` and `edgeComponent`. Within each item, the input specifies components updaters and data structures being updated:

```
# This is a top composite component
<Component facets>
  kind = updaterComponent
# First child component
  <Component core>
    kind = coreComponent
# Data structs of core
    <DataStruct qOld>
      kind = distArray1D
      onGrid = coreGrid
    </DataStruct>
# Similar qNew is skipped
# Updater calculation qNew from qOld
    <Updater accept>
      kind = linCombinerUpdater
      in   = [qOld]
      out  = [qNew]
    </Updater>
  </Component>
# Second child component
  <Component edge>
    kind = edgeComponent
  </Component>
```

Coupling of components is performed by the updater, `myCoreEdgeUpdater`, which is of kind, `explicitCoreEdgeUpdater`. It specifies the names of the parameters, which are passed from one component to another:

```
# Updater coupling core and edge
<Updater myCoreEdgeUpdater>
  kind = explicitCoreEdgeUpdater
  coreName = core
  edgeName = edge
# variables to pass from core to edge
  core2EdgeVars = ["heatFlux"]
# variables to pass from edge to core
  edge2CoreVars = ["temperature"]
</Updater>
</Component>
```

By default, the data exchange is performed after every time step (once components advance by `dt` using their internal updaters not shown in the example).

3.4 Component Composition and Registries

For our dynamic discovery of components, we separate the concepts into the 3 I's: *Interface*, *Implementation*, and *Instantiation*. For components the interface includes the basic methods that all components must have, as shown in Secs. 3.1-2. For updaters, there is another interface, without the dump and

restore. At startup, constructors for all implementations are stored in an associative array that allows one to construct such an object from a string. Thus, upon parsing the above file, and seeing that a component of kind `edgeComponent` is needed, FACETS does a lookup of the name, `edgeComponent`, and is returned a new instance of the class, `FcEdgeComponent`, which has the name, `edge`. That object is then given its section of the input file, which describes how that object will be constructed, and it is put into the component registry.

Having both implementation and instance registries provides flexibility. An implementation registry provides a mechanism for discovery of available implementations. For example, a core source provides the power input to the core from some source, such as neutral beams or electromagnetic radiation (RF). As such, there are multiple implementations, including different implementations for the same physics but having varying degrees of fidelity and computational intensity.

An important task of any Framework is to ensure a valid composition. For example, the simulation can contain no more than one core and one edge; a core cannot contain a core, but it can contain core sources. This can be problematic with previous frameworks for the following reasons. First, for flexibility, FACETS components have identical methods. Thus, one cannot distinguish by *ports*, essentially interfaces. But this fact is innate. At a surficial interface between two components, both components need to return the same quantities, e.g., temperature and energy flux; the coupling involves interchanging these identical quantities. Hence, one cannot use interfaces for ensuring valid component compositions.

Thus, one must develop another method for ensuring valid compositions. One could perform a check using some kind of schema on the input file but it is possible that the initial configuration might need to change (for example, one might want to switch to another solver). Thus there is a need for a run-time mechanism to verify the correctness of composition. Recent discussions have led us to move to using C++ run-time identification (RTTI) and member function pointers registries for ensuring valid composition. Coupling updaters (which know what kinds of components and what kinds of data they need) will check the runtime type of the coupled components to ensure that only the allowed implementations are constructed.

3.5 Multilanguage Support and Parallelism

FACETS is using F90 modules representing turbulent transport models such as `glf23` and `mmm95`. In addition, it brings in F90/Python codes such as `UEDGE` [7] and C-based `WallPSI`. These codes are rewritten as libraries with several methods exposed and wrapped into Babel's `SIDL` [8] so that they can be called from the C++ FACETS code.

The default execution of FACETS is concurrent parallel execution. Load balancing is of two types: internal to a component and intra-component. For the former, the arrays are domain decomposed using an algorithm, which tries to equalize the amount of work per processor. For the latter, the "loads", i.e. the amount of relative work per component, are specified in the input file.

4. SOFTWARE ENGINEERING

Space limitations prevent discussing this in great detail. It suffices to say that FACETS does incorporate a cross-platform build

system (autotools), revision control (subversion), and both regression and unit testing.

5. STATUS AND FUTURE DIRECTIONS

The FACETS team developed a new core component that solves conservation equations for plasma density and energy densities of electrons and ions, taking into account neoclassical transport, energy exchange, plasma turbulence, and heating via external sources. The solution uses adaptive and implicit time integration schemes combined with a nested iteration approach whereby the advance equations are solved on increasingly coarse then fine meshes. In particular, the entire class of diagonally implicit Runge-Kutta schemes, which contains backwards Euler, half implicit Crank-Nicholson, and various flavors of IMEXSSP2 methods can be implemented with no change to the source code, i.e. by assembling "updaters" from a single generic type in the input file. The FACETS core solver runs in parallel, scaling up to one cell per processor.

Using the general component interface, the FACETS team componentized and parallelized the `UEDGE` code and started on componentization of the wall model `WallPSI`. The componentized `UEDGE` and the core component were coupled using scalar values of fluxes averaged over magnetic surfaces.

The next challenge of FACETS will be to use its infrastructure and superstructure to perform a coupled simulation that can be validated against an experiment or other simulation. Next computer science tasks include dynamic load balancing, interface generalization for volumetric coupling and implementing mechanisms for composition validation.

6. ACKNOWLEDGMENTS

This work was supported by DOE grant #DE-FC02-07ER54907.

7. REFERENCES

- [1] Department of Energy, Facilities for the Future of Science: A Twenty-Year Outlook, http://www.sc.doe.gov/sub/Mission/Mission_Strategic.htm, 2004.
- [2] Wael R. Elwasif, David E. Bernholdt, Lee A. Berry, and Don B. Batchelor, Component Framework for Coupled Integrated Fusion Plasma Simulation, in HPC GECO/CompFrame 2007, 21-22 October, Montreal, Quebec, Canada, 2007.
- [3] <http://cswim.org>.
- [4] <http://www.cims.nyu.edu/cpes>.
- [5] V. R. Basili, D. Cruzes, J. C. Carver, L. M Hochstein, J. K. Hollingsworth, M. V. Zelkowitz, and F. Shull, IEEE Software **08**, 0740 (2008).
- [6] C. Nieter and J. R. Cary, "VORPAL: a versatile plasma simulation code", J. Comp. Phys. **196**, 448-472 (2004).
- [7] http://www.mfescience.org/mfedocs/uedge_man_V4.39.pdf
- [8] <https://computation.llnl.gov/casc/components/babel.html>.