Whole-device modeling capability: issues and future

JR Cary, Tech-X

This document is inspired by FACETS (Framework Application for Core-Edge Transport Simulations), which created a single executable (MPI for distributed memory computing but otherwise one memory space) to mirror the evolution of the plasma, its interaction with its environment, and its generation of raw diagnostic data. This approach allowed taking advantage of massively parallel, distributed-memory hardware. Python scripts were used for workflow purposes, including acquiring data from experiments for initial profiles and for equilibrium, and from the latter getting geometric data, which is used for the profile evolution and for edge evolution, and putting core 1D evolution data on the equilibrium mesh to understand the spatial variation of the plasma.

1. Difficulty of knowing what components actually provide or how to get them

A WDM model has to rely on existing fusion components, but they vary in capability and quality:

- Is this component truly predictive? (As opposed to requiring input of some ad hoc diffusivities.)
- Can it be used within a dynamical code? (E.g., a component that predicts only a pedestal pressure height cannot be used without supplement to evolve density and separate temperatures.)
- Can this component be easily built across the platforms now in use by scientists? Scientists now use Linux and OS X, whose compilers have changed significantly in recent years. With Windows support, a scientist could be productive on his/her laptop regardless of which other programs they use for word processing, presentation preparation, and project management.
- Can it be built both shared and static as the need arises?
- Are there internal entanglements that will prevent the use of just the needed part of a component? (FACETS had to disentangle internal dependencies on legacy embedded visualization libraries.)
- Can the component be easily incorporated into any language?
- Has the component namespaced its methods? (As opposed to having methods named init and run, which will collide with the same names of other components.)

2. Integration of components developed with different approaches

The components contributed by the fusion community have been developed very differently. A large number of components have an extensive Fortran 77 heritage, with some moves towards Fortran 90/95, but beyond that FACETS encountered (1) compiled language (Fortran, C or C++) executables, (2) Basis wrapped Fortran, (3) Python wrapped Fortran, (4) home-grown Fortranish dialects converted to Fortran through use of code generators, (5) pure IDL solutions. Thus, the difficulty in integration in a performant manner is very high for both specific cases and in general, where one might have to integrate all types.

- FACETS had a project to disentangle TEQ from Basis, so that TEQ could more easily be reused, but the taskee eventually refused to be further involved; the work was tedious and unrewarding.
- Preservation of globals (such at `MPI_COMM_WORLD`) across both a statically linked, purely compiled component (NUBEAM) and a dynamically linked, mixture of Python, C, C++, and Fortran, as done in UEDGE, was needed to prevent run-time crashes.
- Extensive use of code generators in NUBEAM (no longer needed with advances in Fortran) and an adopted interlanguage tool (Babel) made understanding the code and debugging difficult.

FACETS addressed these challenges because it had to, but many of the challenges are simply not present when components are developed in a compiled language with minimal use of code generators, well-namespaced, and buildable either statically or dynamically.

3. Missing or obsoleted components or infrastructure

A particular need that came out of the FACETS project was to have reduced models for the edge. As noted, such models should be predictive, not requiring input of ad hoc models for diffusivity, they should be able to predict the full range of quantities needed for coupling to a core component.

At the next level one should have 2D fluid components that work with a dynamical equilibrium.

During the period of this project, the BABEL tool lost its ASCR funding when CCA was terminated.

4. Build infrastructure

With continuous integration of living components (those being updated and extended), one has to be able to rebuild components as soon as they change or their dependencies change. This involves the package management layer and the build layer. For cross-platform (Linux, OS X, Windows) package management, FACETS created Bilder (https://svn.code.sf.net/p/bilder/code/trunk). For individual components without decent build systems, FACETS would install a CMake build system using the SciMake (https://svn.code.sf.net/p/scimake/code/trunk) set of modules that set standard variable

definitions and work for the many packages of computational physics.

5. Standard I/O

The fusion community needs to settle on a standard format for I/O for high-performance computing. This is not just saving signals. It is putting out massive data that lives on meshes. The file format should be HDF5, given its wide acceptance and cross platform usability. (FACETS developed VizSchema to solve at least the visualization part of this.)

6. Other software engineering issues

Only a very small fraction of fusion components have adequate documentation, nightly tests, issue trackers, issue dashboards, and project management.

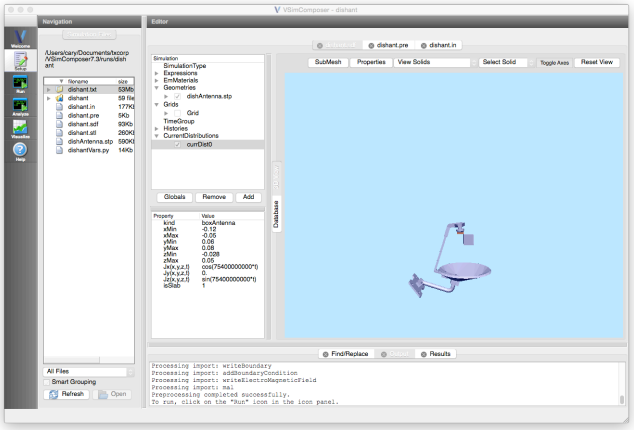

Fig. 1. A modern interface for a computational application having a tree view of the objects in the simulation, including geometric objects, radiation sources, the locations where history data is being taken.

7. Ease of use

If the WDM modeling code is to move beyond having only developers or very advanced PhD's as users, it will have to address ease of uses, such as providing graphical setup like that shown in Fig. 1. Ease of use is considered a critical part of any commercial computational science/engineering code, as it dramatically shortens the time to productive for new users, and it decreases problem setup time for advanced users.

8. Personnel

Attracting the right personnel is difficult because fusion computation does not transfer. The applicability of fusion computations outside of fusion is minimal. Thus, a career spent on developing a fusion specific tool does little for the resumé of a computational scientist who might eventually want to switch jobs, especially if the computationalist has not learned the skills desired by Amazon, Microsoft, NVidia, etc. A job that lacks general opportunities for growth is more difficult to fill.

9. Computational applications as facilities

A computational application is a facility. It has a lifetime that consists of initiation, design, development (construction), release, upgrade, and termination/shutdown. There should be a plan for providing upgrades to releases while also pushing forward with new feature development for the next release. A software facility should make best use of those in the academic, government, and commercial sectors, as do hardware facilities.

With two well funded teams (of order 7 members) the many requirements from developing a faithful representation of the physics to covering basic software engineering could be covered, with the competition in place to prevent complacency. (There may be additional common needs related to infrastructure that could be outsourced.) Ideally, both teams would have independence from any experimental facility with assessment to help ensure credibility. Each team should have a succession plan.

The assessment should define measures of success. Part of this is reviews by outsiders of the design. Does it reflect the physical system? Is it well layered with sufficient abstraction and separation of concerns? Some time after release, a project can be evaluated on the science that it produces. Bibliometrics can provide some of this, as they do for physical facilities. How many papers came from use of this computational facility?

For this to work the agency must make a long-term commitment to a computational facility as it does for a hardware facility.

Even so, like a hardware facility, a computational application has a finite lifetime. Better and more flexible ways of code development arise, and, eventually, those cannot be incorporated, as such would require essential and dramatic structural changes. We recognize this in the physical facility world; the Tevatron and TFTR are no longer -- similarly in software. There need to be procedures for shutdown as well. Such procedures should take into account that a restart is possible in the short term, and so one should have curation processes defined. In the long term, the software may not be resurrectable, and so one must figure out how to ensure that a minimal understanding of what was done is preserved.