Integrated Modeling of Tokamak Experiments with OMFIT*)

Orso MENEGHINI and Lang LAO[1)]

Oak Ridge Associated Universities, Oak Ridge, Tennessee, USA
[1)]*General Atomics, San Diego, California, USA*

(Received 8 December 2012 / Accepted 11 January 2013)

One Modeling Framework for Integrated Tasks (OMFIT) is a framework that allows data to be easily exchanged among different codes by providing a unifying data structure. The main idea at the base of OMFIT is to treat files, data and scripts as a uniform collection of objects organized into a tree structure, which provides a consistent way to access and manipulate such collection of heterogeneous objects, independent of their origin. Within the OMFIT tree, data can be copied/referred from one node to another and tasks can call each other allowing for complex compound task to be built. A top-level Graphical User Interface (GUI) allowing users to manage tree objects, carry out simulations and analyze the data either interactively or in batch. OMFIT supports many scientific data formats and when a file is loaded into the framework, its data populates the tree structure, automatically endowing it with many potential uses. Furthermore, seamless integration with experimental management systems allows direct manipulation of their data. In OMFIT modeling tasks are organized into modules, which can be easily combined to create arbitrarily-large multi-physics simulations. Modules inter-dependencies are seamlessly defined by variables referencing tree locations among them. Creation of new modules and customization of existing ones is encouraged by graphical tools for their management and an online repository. High level Application Programmer Interfaces (APIs) enable users to execute their codes on remote servers and creation application-specific GUIs. Finally, within OMFIT it is possible to visualize experimental and modeling data for both quick analysis and publication purposes. Examples of application to the DIII-D tokamak are presented.

Keywords: simulation, plasma, transport, workflow

DOI: 10.1585/pfr.8.2403009

1. Introduction

Integrating existing stand-alone numerical codes into a unified self-consistent simulation is an important way to improving the fidelity of a physical model, or capturing the complex interplay that different physical processes can have, often over a broad range of space and time scales. In the science of magnetic nuclear fusion, integrated numerical simulations routinely find their application in the analysis and interpretation of existing experiments, the prediction of future ones (experiments preparation, scenario and sub-system design, extrapolation to future devices) and testing of theoretical models.

From a researcher's point of view, integrated modeling entails running many different codes for pre- and post-processing, in addition to the ones required by the modeling of the physics itself. All the codes that comprise the end-to-end modeling system must be run in a very specific order with complex interdependencies, and managing their execution is often difficult even for a single run. If one also considers that a large number of runs that are often needed to get the sought results, then it becomes clear that managing these simulations manually is not a viable solution.

Hence, the magnetic fusion modeling community has been at work to develop software frameworks aiming at progressively integrating the available physics models. However, this is not a simple task, since these projects are faced with the challenge of enabling the inter-operation of programs which were originally developed to be self-contained units, aimed at solving only a specific problems. These programs are "self-centric", meaning that they come in many flavors and variations, each using different system requirements, programming languages, conventions, file formats, units and numerical grids. To make the integration problem even harder, these programs also tend to evolve rapidly and their number is rapidly growing.

The conventional wisdom to deal with this challenge, has been to enforce a rigid set of protocols and formats that the standalone codes must comply to. In this paper we introduce the One Modeling Framework for Integrated Tasks (OMFIT) integrated modeling framework, tackles this problem by treating files, data and scripts as a uniform collection of objects organized into a single, self-descriptive, hierarchical structure (the OMFIT tree structure). Through this unifying structure, data is easily exchanged among different components of the framework. This approach does not require third-party codes to comply to standard data structures, nor does it define a priori

author's e-mail: meneghini@fusion.gat.com

*) This article is based on the presentation at the 22nd International Toki Conference (ITC22).

what codes can interact and how.

This manuscript is organized as follows: Section 2 overviews the state of the art of integrated modeling in the field of magnetic fusion energy. Section 3 presents the design strategy and implementation details of the OMFIT framework. Section 4 briefly describes few application examples of OMFIT to tokamak experiments, with a special focus on the DIII-D experiment. Conclusions and prospects for future developments are outlined in Sec. 5.

2. Integrated Modeling Frameworks for Magnetic Fusion Research

In the field of magnetic nuclear fusion, existing integrated modeling frameworks could be categorized in many different ways (e.g. first principle vs experimental oriented; high performance computing vs general operation). In the following, we categorize frameworks depending on their different organizational structures and techniques with which standalone codes are coupled to one another. We have identified two main categories: *transport-based* and *workflow-based* frameworks.

2.1 Transport-based frameworks

Transport-based frameworks are organized around a core transport code which solves a set of 1D radial transport equations for current, energy, particles and toroidal momentum. These softwares are usually modular in nature, as they rely on existing codes to calculate the plasma equilibrium and the sources, sinks and fluxes which are part of the transport equations. The coupling between the core transport solver and the external components usually occurs at memory level in view of the frequent calls that the core solver has to make to its modules while integrating the transport equations in time. A non-exhaustive list of these codes are ONETWO [1,2], TRANSP/PTRANSP [3], TOPICS [4], ASTRA [5], CRONOS [6], CORSICA [7], FASTRAN [8], FACETS [9].

Generally speaking, coupling programs at memory level is a difficult and time consuming endeavor, since it requires substantial coding on the framework side and access to and understanding of the source code of the components. The level of code reuse offered by this approach is also low. All of this makes it very hard to incorporate new methods into the framework and, therefore, stifles the researcher's creativity. Finally, this approach often leads to large monolithic codes that are difficult to compile, maintain, extend and debug.

2.2 Workflow-based frameworks

Workflow managers assume that the main physics of intests has already been expressed in other standalone codes and their task is to allow data to "flow" through these components. To put things in perspective, in a workflow-based software a transport solver is just another component of the framework.

In contrast with the transport solvers approach, these types of frameworks generally rely on a loose, file-based coupling, assuming that only a modest amount of data (in frequency and volume) needs to be exchanged among its components. A non-exhaustive list of examples are the ITM-TF [10] (with Kepler [11]), FSP [12], SWIM [13] and TASK [14].

In workflow-based frameworks, the conventional strategy adopted to enable data inter-communication among components, is to require these to comply to standard data structures and protocols. This has the double advantage of allowing data to be accessed in a unified way across the framework, and of uniquely defining the location of the data and its properties across the framework. Consequently, the effort of coordinating the work of many people working across many institutions is simplified and in principle multiple implementations of the same type of physics could be easily interchanged.

However, these benefits come with additional complexities, costs and limitations. First , such an approach requires the creation of a large number of bi-directional interfaces, at least one for each standalone code. These need to be as simple as possible, while being general and flexible enough to accomodate the needs of all of the components and their future amendments. In addition to these non-trivial technical and software engineering difficulties, it is often the case that the human and organizational aspects are even more challenging. Definition and implementation of rules (standards and protocols) require unanimous agreement among the interested parties. Experience shows that for most IM frameworks a lot of time and resources can be spent on this process [10, 12], before the framework itself can be built and physics results delivered.

To further complicate things, at a community level there is not a general agreement about which standard to adopt and many different data structures have been developed, one for each framework. To name a few, the SWIM plasma state file (which is different from the ONETWO plasma state file) or the Consistent Physical Object [15] used by the ITM-TF, or the BPSD used in TASK. Such complications add more work to the developers of standalone codes, since it prevents reusing the work done in one framework for another. It is likely that the fusion community will eventually agree on a unique set of standards and protocols when ITER formally chooses one for its simulations.

A special note must be said about the different approach followed by the TASK software, which has been developed at the Kyoto University. The TASK framework circumvents the problem of imposing third party software to adhere to a standard data structure by developing all of its components "in-house". Although this approach requires a significant amount of resources and time to be invested in developing and validating each of the framework components, TASK has proven to be extremely successful and to date it is one of the frameworks which has the most

complete set of physics components.

3. OMFIT: One Modeling Framework for Integrated Tasks

Conventional integrated-modeling frameworks are designed around well-defined physics or technical goals, such as simulating an experiment from startup to ramp-down, coupling core and edge transport with very comprehensive edge physics, or use peta-scale computers. Although these frameworks have been successful at reaching their original goals, the case is that in the science of magnetic fusion, most researchers still manually integrate standalone codes.

The cause of this must be sought in the disconnect between the goals around which conventional frameworks are built and the everyday practical needs of most fusion theorists and experimentalists. In general, we have found that the "top-down" approach embraced by conventional frameworks inherently constrains the users and limits the overall scope of the framework.

To be attractive to a general audience of (fusion) researchers, a modeling framework should first of all be generic and flexible, since each researcher has specific requirements in terms of what codes need to be integrated and how (including pre-processing, execution, post-processing and analysis). Second, a framework should strive at maximizing the user's efficiency and make the benefits of adopting the framework worth the time and effort required when learning a new software. Hence, a framework should be capable of integrating and reuse "custom-made" codes, widgets, tools and scripts which have been developed over many years of research. In this context, one should consider that for most purposes only few codes need to be loosely-coupled, and requiring all of the outputs to comply to standard data structures is often not justified. Lastly, to promote the validation process and be experimentalist-friendly, a framework should allow direct access and manipulation of experimental data.

With these considerations in mind, we developed a new integrated modeling framework named OMFIT. OMFIT belongs to the class of workflow-based frameworks. In OMFIT, task inter-communication occurs through a unifying data structure and usage of a high-level language, which serves as a glue to tie modules and components together to rapidly create specialized applications. In some sense, the language becomes a scripting framework allowing fast prototyping of new applications.

Since the control of the simulation's workflow and data exchange is delegated to the users, we name this approach a *bottom-up* paradigm as opposed to a *top-down* paradigm, typical of conventional integrated modeling frameworks. The design and implementation details of the main components in the OMFIT framework are discussed in the following paragraphs.

3.1 The OMFIT tree data structure

OMFIT is a workflow manager which allows data to be easily exchanged among different components of the framework, without requiring each component to comply to a pre-defined data structure. This is accomplished by treating files, data and scripts as a uniform collection of objects organized into a single, self-descriptive, hierarchical structure (the OMFIT tree structure).

Objects residing in the tree are automatically endowed with different attributes and methods, depending on their type (e.g. files have a name, strings have a length, arrays can be plotted, scripts can be executed). Within OMFIT, the object-oriented programming concepts of (sub-/super-)classing, (multiple-)inheritance and polymorphism are used to effectively and consistently assign attributes and methods to different objects (e.g. a FORTRAN namelist is an ASCII file, that is a file that is an object).

When an object is loaded into OMFIT, its data is interpreted and populates the OMFIT tree structure. As an example, when loading a FORTRAN namelist file, this will become an object within the OMFIT tree, containing the FORTRAN namelist names and variables organized in a hierarchical (subtree) fashion. Figure 1 shows how a sample FORTRAN namelist is loaded into the OMFIT tree and appear in its Graphical User Interface (GUI).

The same concept is applied to many other scientific data formats (FORTRAN namelists, NetCDF files [16], IDL [17] save files, MATLAB [18] save files, to name a few). By supporting few scientific data formats, a great number of codes can be directly integrated into the OMFIT tree without the need to specify a priori from which codes the data comes.

In the limited number of cases when non-standard file-formats are used, it is often the case that tools have been developed over the years to translate these files into standard ones. Also worth pointing out is, while the OMFIT approach does not require the use of standard data structures, it does not exclude the possibility of using one (any) upon which other integrated modeling frameworks are based.

The tree-structure approach shares many similarities with a conventional filesystem or the MDS+ software [19], which has proven to be very successful system for the aggregation of heterogeneous experimental data and its management. Like these systems, the OMFIT tree is an ab-

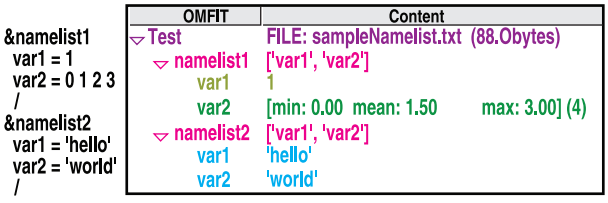

Fig. 1 Sample FORTRAN namelist and corresponding hierarchical structure within the OMFIT tree structure.

straction that allows access to the data and the metadata of the objects, independently of the objects' origin or their low-level storage details. This is why we say that all data objects residing in the tree data structure are accessed in a unified way. Worth remembering is that a conventional framework achieves this same functionality by imposing a standard data structure.

Certainly, the OMFIT approach most-likely implies dealing with a high level of diversity in data structures, units and numerical grids topologies and sizes. However, this must be put into context. First, many of the codes which researchers want to integrate already understand each others data (for example, most stability codes would accept as input the files generated by the EFIT [20] equilibrium code). Second, when data has to be exchanged in most cases, the required coupling is loose, meaning that a small volume of data is shared among standalone codes. And lastly, the OMFIT framework is designed to simplify this task by relieving the users of the burden to manage and access data from individual files or different sources.

In addition to editing, management and visualization functionalities, OMFIT has the ability to compare and selectively merge any two branches of the tree and provide a description of the objects in the tree by exploiting the context that the hierarchical structure inherently defines. Finally, when an OMFIT tree is saved (into what is called an OMFIT project), scripts, data and settings are saved into a single file, thus, easing the problem of tracing back in time how a certain output was obtained.

3.2 Direct access to experimental data

Data from experimental data management systems, such as MDS+ or SQL relational databases, is seamlessly integrated in the OMFIT tree. This means that within OMFIT there is no distinction between the data which was originally stored into a file on the local filesystem or a remote data management system, as both are accessed and manipulated in the same manner.

Such transparency facilitates modeling-experiment interaction and makes the framework especially appealing for validation purposes or developing experiment analysis tools. This functionality is especially crucial for automating simulations, for quick analysis (e.g. control room) of the experiments and for creating large experimental datasets to be pre-processed to run simulations.

3.3 Modules and their management

In OMFIT modeling tasks are organized into "modules", which can be easily combined (also hierarchically) to create arbitrarily-large multi-physics simulations.

Although the scope of each module is arbitrary, in general this tends to coincide with the set of data and scripts which allows the execution of a standalone code. Module structure is also chosen by the users, but it is usually organized into sub-branches each containing the relevant files, scripts, GUIs, plots and settings. Typically, in the "settings" sub-branch a set of entries defines the execution of the code, such as the location of the executable, the working directory and on which remote machine the execution should occur.

Tools that support the creation of new modules and customization of existing ones are available within OMFIT. The creation of new moules in their basic functionalities (remote execution and data management) is easy, especially if the data formats used by the standalone codes are already supported.

Sharing of modules among different users is encouraged and supported by an online modules repository, where users can download and upload new modules and updates of existing ones. The idea is to create a cooperative environment in which the components of the framework can grow in an grassroots fashion based on the specific problems which are of interest to the OMFIT user community. In this context, OMFIT comes with GUI tools that facilitate the import/export and merge from/to the modules repository.

In the OMFIT framework philosophy, users roughly fall into three classes distributed in a pyramid. At the top are the framework programmers who take charge to support more data formats and expanding the functionalities of OMFIT. The second and larger class of users, are those who produce their own OMFIT modules. Typically, these are experts of each of the standalone codes (though not necessarily the code developers). The third and largest class of users are those who combine existing modules for their own (integrated) modeling purposes.

3.4 Simulation workflow at user-level

In OMFIT, the control of the simulation workflow is delegated to user-level tasks, rather than being built into the framework. For this purpose, OMFIT makes use of an interpreted programming language to create a dynamic environment in which components can be tied together at a high level. This choice does not restrain the user's possibilities to a pre-defined set of operations.

Tasks are programmed in Python [21], a high-level, object-oriented scripting language which features the strong community support and offers basic facilities for interactive work and a comprehensive library on which more sophisticated systems can be built. Compared to other scripting languages, Python offers several advantages which are key to a successful component integration. These include its being cross-platform, open-source and modular. Also, Python has profiling, debugging, reflection, introspection and self-documentation capabilities. It's concise and almost pseudocode-like syntax which promotes code readability and thus maintainability. Finally, Python is an excellent "steering" language for scientific codes written in other languages, which ultimately enable OMFIT users to reuse the numerical tools which they have

developed over many years.

The typical script for the execution of a single standalone code requires only a few lines of Python programming and can be conceptually divided into tree parts: deployment of tree objects to a working directory (e.g. write namelist), execution of the standalone code, and insertion of the simulation results in the OMFIT tree (e.g read NetCDF). The same operations apply if the code execution is remote (Sec. 3.6). These operations correspond to nothing less than the "initialize()", "step()" and "finalize()" actions which are typical of conventional workflow managers.

Within the OMFIT framework, tasks which are stored in the OMFIT tree can call each other to create complex workflows. In OMFIT, the problem of inter-module dependencies is resolved by variables which can be defined and used by users within the scope of a module to point to any location within the OMFIT tree. In their scripts, users can access tree nodes by their absolute location from the OMFIT tree root or by their relative location from the root of the modules the scripts belong to. This feature allows user scripts to work within a module, independently to the locations of the scripts within a module and of the modules within the OMFIT tree. Also, a special type of object, named dynamic expression, can be used to reference and dynamically calculate quantities across the tree while ensuring data self-consistency.

In OMFIT, user-level scripts are completely independent from the framework itself. This has the advantage that tasks can be developed within the framework without the need to restart OMFIT every time a script is edited and re-executed. This and the ability to quickly inspect data within the OMFIT tree greatly expedites the debugging process.

3.5 Top-level graphical user interface

In OMFIT, a top-level GUI is available to the users to manage the data structure, carry out simulations and analyze the data interactively. This feature would not be possible without the unified data access which is provided by the OMFIT tree data structure.

The GUI is a key element of the framework as it enables fast prototyping. With it, users can selectively develop, execute and debug each step in their workflow before running them in batch. This interactive environment quickly accommodates the experimentation that is often required to arrive at effective solutions. Figure 2 shows a snapshot of the GUI, with some of the key elements highlighted.

The "tree browser" allows interactive manipulation of the OMFIT tree, tabbed browsing provides multiple simultaneous views of OMFIT tree. Within the OMFIT tree browser, all the relevant data and tasks are at reach for being edited, managed, inspected or executed. Objects in the OMFIT tree can be edited by a mouse double-click; additional functionalities (e.g. copy, paste, compare, plot) can be accessed by a mouse right-click. Hot-keys allow users to plot and over-plot numerical arrays, edit scripts in the text editor chosen by the users or execute them.

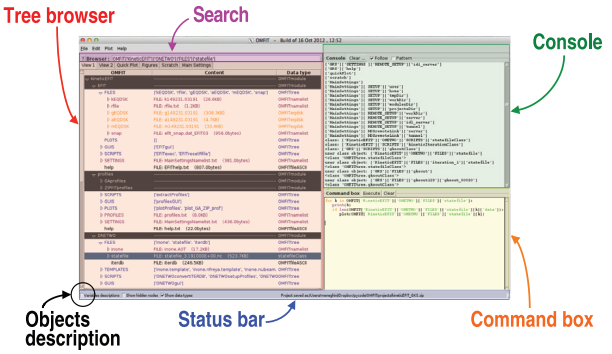

Fig. 2 The main graphical user interface of OMFIT with the key components highlighted.

The "navigation bar" provides the path to the selected tree location and allows quick-search within the selected location. The "command box" is a sandbox for interacting with the OMFIT tree and for prototyping scripts. The "console" provides real-time feedback (output and error) from scripts and standalone execution.

3.6 High-level API for remote execution, data management & generation of user-level GUIs

To simplify and streamline the development of its scripts, OMFIT comes with a set of services for task execution, data management, and inter-component communication. Of these, worth highlighting is the Application Programmer Interface (API) which allows remote code execution and data transfer on a remote host via Secure SHell (SSH), a workhorse in scientific computing. Instead of requiring that all standalone codes are executed on the same system where the framework resides, OMFIT (which could be running on a notebook) can spawn processes on different remote workstations or powerful servers (even if these are behind gateway hosts). Not only does this relieve scientists of the burden of dealing with remote connections, but it also eliminates the need to compile (a notoriously difficult and painful process) codes which are already available and working properly on workstations in the network. Services for concurrent execution of codes on multiple remote systems are also available. Figure 3 summarizes the functionalities provided by the APIs for remote execution.

GUIs can be very useful to hide underlying complexities to inexperienced users and ease streamline analysis. However, most scientists do not use GUIs to interact with their own programs because these can be relatively hard and time consuming to program. In OMFIT this task has been simplified to its minimum. Thanks to some high level APIs, programmers can create application-specific GUIs

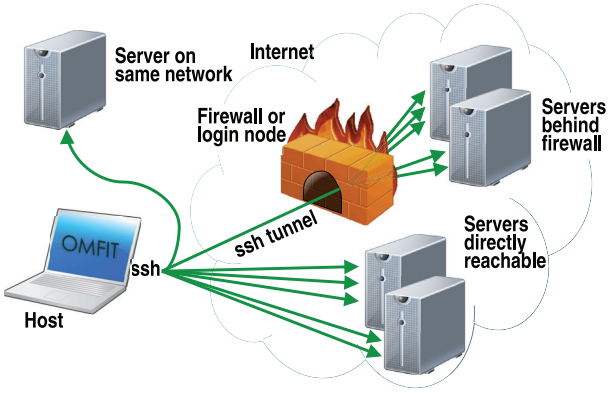

Fig. 3 High-level APIs available within the OMFIT framework enable scientists to execute standalone codes remotely (and concurrently) and to manage remote data.

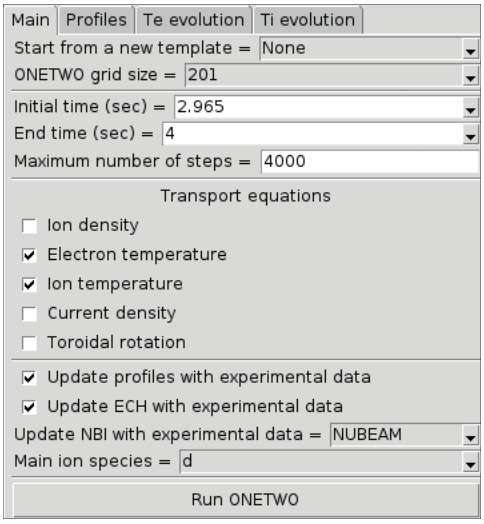

Fig. 4 Example of user-level GUI for the ONETWO transport code. User-level GUIs are easily built by defining each element in the GUI (type of graphical element, the tree location) to be edited and a description. As the user interacts with the GUI, the corresponding variables in the OMFIT tree are updated.

for their modules. The idea is that to each element that composes a GUI, the programmers have to specify only a description and a location in the OMFIT tree which will store the input from the user. With the same ease, multiple user-level GUIs from different modules can be combined in a hierarchical way to create larger and more complex GUIs, while ensuring self-consistency for those GUIs which access data across different modules. Since the low-level implementation details of the GUIs are hidden to the programmers, it is not even necessary to know how to do event programming - which is how typical GUIs are usually developed. As an example, Fig. 4 shows the user-level GUIs developed for the ONETWO transport code.

3.7 Post-processing & visualization of modeling & experimental data, comparison & merging

The process of data analysis is increased by quick visualization of 1D and 2D data arrays which reside in the OMFIT tree. Within the main GUI, this can simply be accomplished by the push of a button. The plots generated are objects that become part of the OMFIT tree themselves and that can be edited or saved for later reference.

More articulated plots can be generated using the *Matplotlib* [22] plotting library. Users can generate plots, histograms, power spectra, bar charts, error-charts, scatter-plots, contour-plots, etc, with just a few lines of code. Matplotlib can be used in the user-level Python scripts or interactively (ala MATLAB) in the OMFIT command box (see Sec. 3.5).

Within OMFIT, users can add custom attributes and methods to any object by sub-classing. With this method, custom plot commands can be associated to specific files (such as the output files of a code), thus allowing users to inspect multiple results of their simulations by simply selecting the output file objects and pressing the plotting or over-plotting hot-keys in the OMFIT master GUI.

Finally, all figures which are generated in the OMFIT environment can be viewed (zoom, pan), edited (copy, paste, change line attributes, etc) and saved (in a variety of hardcopy formats) in an interactive way.

3.8 Portability

For the purpose of easing the installation and maximizing the portability of the framework, the Python libraries upon which the framework depends on are all public, well-supported, and their number has been minimized. The *numpy* library allows the efficient storage and manipulation of large amounts of numerical data [23], *SciPy* [24] provides a vast collection of scientific algorithms, *Tkinter* [25] is used for the GUIs, and *Matplotlib* provides interactive plotting functionalities. These choices do not limit the freedom of OMFIT users, which in their scripts can use any Python libraries available on their system.

4. Examples of Application

The OMFIT framework has been used for the analysis of several physical processes of DIII-D discharges and, for this purpose, many OMFIT modules have been developed. These include modules for equilibrium (EFIT), MHD stability (GATO [26], PEST-3 [27]), transport (ONETWO and GCNMP [28]), ray-tracing (GENRAY [29]) and turbulent stability (GSK [30], TGLF [31]).

A notable example is the kinetic-EFIT module that enables realistic magnetic equilibrium reconstruction using kinetic pressure constraints and motional Stark effect (MSE) measurements. This procedure entails iterating between the equilibrium reconstruction code EFIT and the transport code ONETWO, since the kinetic profiles and

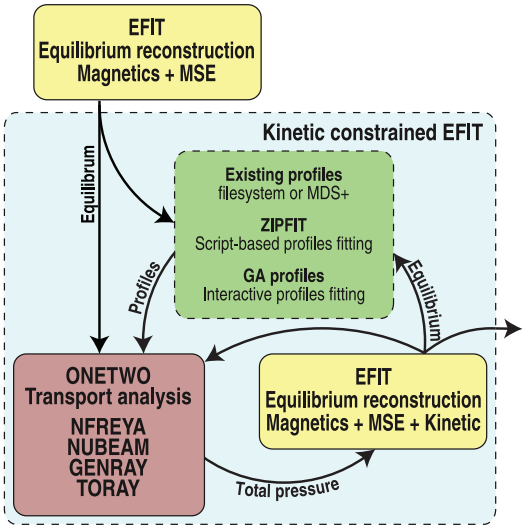

Fig. 5 Schematic showing data flow across different components in the OMFIT frameworks for a kinetic-constrained equilibrium reconstruction.

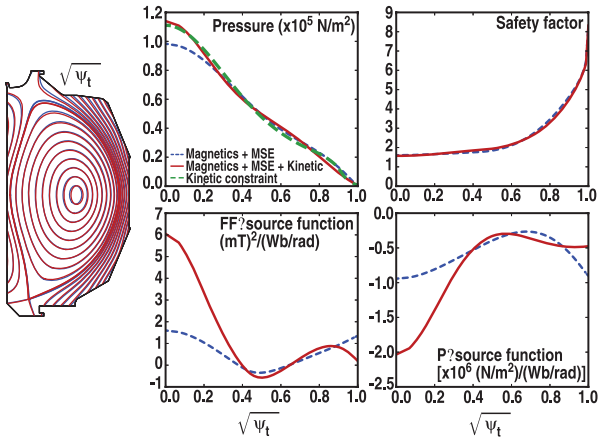

Fig. 6 Comparison of equilibrium reconstruction for DIII-D shot 147634 at 4965 ms with (red) and without (blue) kinetic pressure constraint as calculated by the workflow depicted in Fig. 5. The total pressure constraint (green) includes the neutral beam fast ion pressure contribution.

the auxiliary heating schemes deposition profiles (most importantly the fast-ions pressure which are generated by the neutral beams) are updated based on the magnetic equilibrium calculated at the previous step. Changes in the magnetic equilibrium also influence the reconstruction of density, temperatures, Z_{eff} and radiation profiles, which need to be updated at each step of the iteration. To speed this type step, we have developed a module which is dedicated to the problem of retrieving and managing the DIII-D experimental plasma profiles. Overall workflow of the kinetic constrained equilibrium reconstruction is shown in Fig. 5. Finally, a user-level GUI oversees all steps in the reconstruction and is used to assist users to streamline their analyses. Figure 6 shows the comparison of an equilibrium reconstruction from an MSE constrained kinetic-EFIT and an EFIT which only uses magnetic and MSE data.

The flexibility and the potential of this tool, used in conjunction with the TGLF modules, was demonstrated by the execution of kinetic-equilibrium reconstructions and the analysis of turbulent growth rates as a function of wavenumber and radius during DIII-D operations. In that case, multiple local linear stability simulations were run in parallel on different remote machines to obtain faster results and to deliver quick guidance to the experiment session-leader.

Most recently, the OMFIT framework is being used to compare the predictions of the resonant magnetic perturbation-induced magnetic flutter model and the Neoclassical Toroidal Viscosity model [32] with the temperature and toroidal rotation profiles measured in DIII-D experiments. These works required the interaction among multiple OMFIT modules (including kinetic and MSE constrained equilibrium reconstructions, transport and non-axisymmetric MHD calculations) as well as direct manipulation of the experimental data. For both models the response to non-axisymmetric fields is provided by the two-fluid MHD code M3D-C1 [33].

Finally, a workflow has been setup to study the interdependence between plasma equilibrium, turbulent transport, MHD stability, heating and current drive in a self-consistent way. This workflow will be used to optimize the performance of DIII-D advanced tokamak discharges and to support the design of the Fusion Development Facility (FDF) [34] experiment.

5. Conclusion

The OMFIT framework adopts a new paradigm to enable the integration of existing modeling tools, without specifying a priori what codes are to be coupled and how. Instead, the framework provides the users with the tools to manage the data flow and the codes execution. The control of the simulations workflow is delegated to user-level tasks which are programmed in Python, and the OMFIT tree data structure (a pivotal concept) allows users to seamlessly concentrate, manage and exchange the data gathered from different sources, including experimental data management systems.

Within the OMFIT framework, scientists are encouraged to write modules instead of stand-alone programs and scripts. This concept and the ability to share modules among users promotes code reuse and ultimately leads to rapid prototyping.

The OMFIT approach has a lot of strengths and benefits, as we have already witnessed a dramatic increase in our productivity, as well as a high level of code reuse. This is attested by the broad range of application modules which have been developed in support to the DIII-D research, since its development has started, less than a year ago. The spectrum of application ranges from processing of exper-

imental data for analysis and support of experiments, to prediction of discharge evolution and the implementation and validation of new physics models.

Finally, although the OMFIT design has been driven by the needs of the magnetic fusion community, its approach is so general and versatile that it could also serve the needs of a broader scientific community.

6. Acknowledgments

This work was supported in part by the US Department of Energy under under DE-FG02-95ER54309. The authors wish to thank S.P. Smith, C. Paz-Soldan, G. Li, P. Raum, and A. McCubbin for contributions in developing some of the OMFIT modules and H.E. St John, A.M. Garofalo, J. Candy, V.S. Chan, N.W. Ferraro, J.M. Park, G.M. Staebler, R. Prater, A.D. Turnbull, and S. Shiraiwa for their insights and support.

[1] W.W. Pfeiffer *et al.*, "ONETWO: A computer code for modeling plasma transport in tokamaks" Nuclear Fusion **1** (1980).
[2] H. St John *et al.*, Presented at the 15th International Conf. on Plasma Physics and Controlled Nuclear Fusion Research, Seville, Spain, 1994, **1** (1994).
[3] integrated modeling code, http://w3.pppl.gov/transp/
[4] H. Shirai *et al.*, Plasma Phys. Control. Fusion **42**, 1193 (2000).
[5] G. Pereverzev and P.N. Yushmanov, "ASTRA automated system for transport analysis in a tokamak" Max-Planck-Institut fuer Plasmaphysik, Garching, Germany (2002).
[6] J.F. Artaud *et al.*, Nucl. Fusion **50**, 043001 (2010).
[7] J.A. Crotinger, L. LoDestro, L.D. Pearlstein, A. Tarditi, T.A. Casper and E.B. Hooper, "Corsica: A comprehensive simulation of toroidal magnetic-fusion devices," Lawrence Livermore National Laboratory Report, Livermore, CA USA (1997).
[8] M. Murakami *et al.*, Nucl. Fusion **51**, 103006 (2011).
[9] J.R. Cary *et al.*, J. Physics: Conference Series **78**, 012086 (2007).
[10] A. Bécoulet *et al.*, Comput. Phys. Comm. **177**, 55 (2007).
[11] B. Ludäscher *et al.*, Concurrency and Computation: Practice and Experience **18**, 1039 (2006).
[12] Jill Dahlburg *et al.*, J. Fusion Energy **20**, 135 (2001).
[13] W.R. Elwasif *et al.*, in Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conf. on, Institute of Electrical and Electronics Engineers (IEEE), pages 419427 (2010).
[14] A. Fukuyama *et al.*, in Proc. 20th Fusion Energy Conf., Villamoura, Portugal (2004).
[15] B. Guillerminet *et al.*, Fusion Eng. Des. **83**, 442 (2008).
[16] R. Rew and G. Davis, IEEE Comput. Graph. Appl. **10**, 76 (1990).
[17] EXELIS, "DL. Exelis Visual Information Solutions," Boulder, Colorado.
[18] MATLAB, The MathWorks Inc., Natick, Massachusetts.
[19] J.A. Stillerman *et al.*, Rev. Sci. Instrum. **68**, 939 (1997).
[20] L.L. Lao *et al.*, Nucl. Fusion **25**, 1611 (1985).
[21] M.F. Sanner *et al.*, J. Mol. Graph. Model. **17**, 57 (1999).
[22] J.D. Hunter, *Comput. in Science & Engineering* (IEEE Computer Soc., 2007) pp.90–95.
[23] T.E. Oliphant, *A Guide to NumPy* **1** (Trelgol Publishing USA, 2006).
[24] E. Jones, T. Oliphant and P. Peterson, "SciPy: Open source scientic tools for Python," http://www.scipy.org/ (2001).
[25] F. Lundh, "An introduction to tkinter," URL: www.pythonware.com/library/tkinter/introduction/index.htm (1999).
[26] L.C. Bernard, F.J. Helton and R.W. Moore, Comput. Phys. Comm. **24**, 377 (1981).
[27] A. Pletzer, A. Bondeson and R.L. Dewar, J. Comput. Phys. **115**, 530 (1994).
[28] Holger St John, "Globally Convergent Newton Method Parallel (GCNMP) solver," under development, private communication.
[29] A.P. Smirnov and R.W. Harvey, "The GENRAY ray tracing code," CompX Report CompX-2000-01 (2001).
[30] Mike Kotschenreuther, G. Rewoldt and W.M. Tang, Comput. Phys. Comm. **88**, 128 (1995).
[31] G.M. Staebler, J.E. Kinsey and R.E. Waltz, Phys. Plasmas **14**, 055909 (2007).
[32] J.D. Callen, A.J. Cole and C.C. Hegna, Phys. Plasmas **19**, 112505 (2012).
[33] N.M. Ferraro, Phys. Plasmas **19**, 056105 (2012).
[34] V.S. Chan *et al.*, Nucl. Fusion **51**, 083019 (2011).