EUROfusion Integrated Modelling workflows Documentation

Release 3.0

WPCD

Feb 15, 2019

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION TO CODE DEVELOPMENT FOR INTEGRATED MODELLING

The EUROfusion Project on Code Development for Integrated Modelling (WP-CD) supports the achievement of the European Fusion Roadmap at Horizon 2020 goals, via the development of existing modelling codes with a particular focus on integrated modelling. The primary objectives of WPCD are: 1. Provide a suite of codes that can be validated on existing machines and used for JT-60SA, ITER and DEMO predictions:

- build on the existing modelling codes developed by the EUROfusion Consortium members including the Integrated Modelling (EU-IM) infrastructure, toolset and codes developed under the former EFDA ITM Task Force,
- add new physics to the existing models
- couple codes into integrated workflows
- optimize codes.

2. Specific ITER simulation work in support of ITER IO and F4E with specified deliverables. WPCD operates under a work plan aiming to provide in the long term a full suite of integrated simulation workflows, incorporating core-edge-SOL/PFC coupling, first-principles models and control elements. A central task is the development of the new modular European Transport Simulator, ETS, which is being deployed to JET modelling infrastructure for validation and application to experimental analysis. In addition to code and workflow development, rigorous code verification is also performed under WPCD, within the EU-IM framework; whereas validation of the released integrated modelling workflows against the experiments is performed under the rerelated Task Forces. EUROfusion WPCD webpage. The EU-IM Team includes both EUROfusion WPCD and WPISA CPT contributors, see EU-IM Team. This list reproduces the status of members in 2015 and is possibly not exhaustive.

1.1 The European Integrated Modelling (EU-IM) approach

The choice of Integrated Modelling made by the former EFDA ITM and pursued now under EUROfusion WPCD is unique and original: it entails the development of a comprehensive and completely generic tokamak simulator including both the physics and the machine, which can be applied for any fusion device. The simulation platform was designed to be fully modular, flexible, and independent of a programming language.The choice of modularity implies that each module contains a single physical model and that the communication between the modules is standardised: a set of common common rules (ontology) clearly specify the format of the data to be consistently exchanged between modules (data-structure). The complexity of coupling the codes together is therefore transferred to the definition of a generic data-structure (allowing to describe and exchange information concerning both physical quantities and technical objects, not assuming the origin of those), extensible to allow the integration of new physics, as well as more elaborate machine geometries and experimental data in the future. A central project is the development of the so-called **European Transport Simulator (ETS)** aimed to meet all the EU-IM requirements, namely modularity, flexibility and standardized interfaces. In terms of the physics, the ETS is designed to solve the standard set of one-dimensional time dependent equations which describe the evolution of the core plasma. The solver itself is designed with a modular approach enabling the separation of the physics from the numerics, thereby facilitating the testing/usage of the numerical schemes that best suit a particular physical simulation.

1.2 Mission

The EU-IM operated under EFDA from 2004 until 2013. The main mission of EU-IM was to provide a software infrastructure framework for EU integrated modelling activities as well as a validated suite of simulation codes for the modelling of present experiments, ITER and DEMO plasmas. The EU-IM operated until 2013 under a work programme formulated to support this goal, structuring the EU modelling effort around existing experiments and ITER scenario prediction while maintaining a long term strategic aim to provide a validated set of European modelling tools for ITER exploitation.

1.3 Achievements

During the first phase of the EU-IM, surveys and cross-verification of the available European models and numerical codes were performed within the individual IMPs and the data-structure was extensively discussed and defined. Equilibrium, linear MHD stability, core transport and RF wave propagation, as well as the poloidal field systems and a few diagnostics were the first topics addressed. Data structures have been finalised for these and then expanded to address, among others, non-linear MHD, edge physics, turbulence and neutral beam propagation. In parallel to the development of the physics concepts, EU-IM developed the tools to manipulate the data structure and use it in fully flexible and modular simulation workflows. The EU-IM database contains machine descriptions from JET, Tore Supra, MAST, FTU, FAST, AUG, ITER as well as some experimental data from Tore Supra and JET. The EU-IM futher achieved the development of the first release version of a fully modular and versatile simulator, the ETS, with all the essential functionalities. The validation of the ETS simualtor started in 2010 against the state-of-the-art transport codes and ETS now starts to be used for the first physics applications. Next steps are the validation of the simulator for a complete discharge on existing experimental data with the available modules, the integration of more quantitative physics models ("ab-initio") and the integration of the whole modelling of the device. Some posters that describe the EU-IM were presented at an ITM EXPO at the 2011 EPS fusion conference in Strasbourg.

1.4 Structure

The EFDA ITM was structured into four Integrated Modelling Projects (IMPs) focusing on the following physics areas:

- **IMP12** plasma equilibrium and MHD
- **IMP3** transport code and whole discharge evolution
- **IMP4** transport and micro-instabilities
- **IMP5** heating, current drive (H&CD) and fast particles

The "Infrastructure and Software Integration Project" (**ISIP**) was in charge of developing, maintaining and operating the code platform structure and implementing the EU-IM data-structure. A key function of ISIP was to provide infrastructure support to the IMPs. At present, under EUROfusion, this expertise and tasks are ensured by the **Core Programming Team** under the **Infrastructure and Support Activities Work Package (WP-ISA CPT)**. Two further projects ensured the link with the experimentalists and the provision of the experimental databases:

- **AMNS** the "Atomic, Molecular, Nuclear Surface" Data
- **EDRG** "Experimentalists and Diagnosticians Resource Group"

The "ITER Scenario Modelling Working Group" (**ISM**) was established in 2007 as part of EU-IM with the aim to assist in systematic predictive modelling of all ITER reference scenarios by using the major existing integrated modelling tools, whilst the EU-IM code platform was in development. ISM also supported the verification and validation of the ETS, which aims to become the main tool for EU modelling activity.

1.5 Contributors

EU-IM contributors are defined in the Appendix of G.L. Falchetto et al., Nuclear Fusion 54,043018, 2014. This list reproduces the status of EU-IM members in 2012 and is not exhaustive. A grateful thank you to all those who contributed and promoted EU-IM since its beginnigs.

1.6 Glossary

Collaborative Development Environment (CDE) A **collaborative development environment (CDE)** is an online meeting space where a software development project's stakeholders can work together, no matter what timezone or region they are in, to discuss, document , and produce project deliverables. The name was coined by Grady Booch.

Consistent Physical Object (CPO) A Consistent Physical Object (CPO) is a physics based, hierarchical data structure employed by the EU-IM for a complete description of a physics area, e.g. equilibrium. All EU-IM code modules interact through the exchange of CPOs. The CPOs also form the basic block of data written to the EU-IM database.

Content Management System (CMS) A **content management system (CMS)** is the collection of procedures used to manage work flow in a collaborative environment. These procedures can be manual or computer-based. The procedures are designed to:

- Allow for a large number of people to contribute to and share stored data
- Control access to data, based on user roles. User roles define what information each user can view or edit
- Aid in easy storage and retrieval of data
- Reduce repetitive duplicate input
- Improve the ease of report writing
- Improve communication between usersq

In a CMS, data can be defined as nearly anything - documents, movies, pictures, phone numbers, scientific data, etc. CMSs are frequently used for storing, controlling, revising, semantically enriching, and publishing documentation.

FC2K FC2K is a tool for wrapping a Fortran or C++ source code into a Kepler actor. Before using it, your physics code should be EU-IM-compliant (i.e. use CPOs as input/output).

Gforge Gforge hosts all projects (software and infrastructure) under the EU-IM.

EU-IM Gateway The EU-IM Gateway is a compute cluster located at Portici (near Napoli in Italy). It is uses for development and fusion simulations in the EU-IM.

EU-IM Portal The EU-IM Portal is the web portal for the EU-IM, i.e. it hosts the EU-IM web pages and projects under Gforge.

Integrated Simulation Editor (ISE) The Integrated Simulation Editor ISE allows you to visualize and edit data from an EU-IM database entry. It also allows running a Kepler workflow based on the opened data entry.

Universal Access Layer (UAL) The UAL (Universal Access Layer) is a multi-language library that allows exchanging Consistent Physical Objects (CPOs) between various modules, and to write to an EU-IM database.

actor Actors take execution instructions from a director. In other words, actors specify what processing occurs while the director specifies when it occurs. In the EU-IM, actors are usually modules which contain physics codes like EQUAL or HELENA.

calibration The process of adjusting numerical or physical modelling parameters in the computational model for the purpose of improving agreement with experimental data.

data mapping An XML file containing all the mapping essentials for mapping from a local experimental database for a specific tokamak device to the EU-IM database. The mapping essentials include for instance the download method, local signal names, gains and offsets, time base, and eventual interpolation option to ensure that only one time base is set for each CPO that is built from multiple local signals. A java code (exp2ITM developed under ISIP), with the MD and DM files as inputs, is then run to connect to the local device database, retrieve the required experimental data and populate the EU-IM database instance for that shot/device and dataversion.

director A director controls (or directs) the execution of a workflow, just as a film director oversees a cast and crew.

error A recognisable deficiency in any phase or activity of modelling and simulation that is not due to lack of knowledge.

kepler Kepler is a software application for the analysis and modeling of scientific data. Kepler simplifies the effort required to create executable models by using a visual representation of these processes. These representations, or “scientific workflows”, display the flow of data among discrete analysis and modeling components.

machine description The machine description (MD) of a device basically builds on the set of engineering and diagnostic settings characterising a tokamak device. This includes, for instance, the vessel/limiter description, the PF coils and circuiting and lines of sight of diagnostics. In practice, all MD information is encapsulated in an XML file that emanates from the MD tagged datastructure schemas. An MD instance of a given device is then stored into the EU-IM database as shot 0 for that device database.

model A representation of a physical system or process intended to enhance our ability to understand, predict, or control its behaviour.

- A **conceptual model** consists of the observations, mathematical modelling data, and mathematical (e.g., partial differential) equations that describe the physical system. It will also include initial and boundary conditions.

- The **computational model** is the computer program or code that implements the conceptual model. It includes the algorithms and iterative strategies. Parameters for the computational model include the number of grid points, algorithm inputs, and similar parameters, etc.

modelling The process of construction or modification of a model

prediction Use of a model to foretell the state of a physical system under conditions for which the model has not been validated.

simulation The exercise or use of a model.

uncertainty A potential deficiency in any phase or activity of the modelling process that is due to the lack of knowledge.

validation The process of determining the degree to which a model is an accurate representation of the real world form the perspective of the intended uses of the model.

verification The process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model.

1.7 Support

1.7.1 Getting support for the EU-IM platform and Gateway

The EU-IM provides several ways to get support when you run into problems. Which one to choose depends on the nature of your problem. This page tries to give an overview.

1.7.2 Support for problems related to the EU-IM Gateway

The official documentation of the ITM Gateway can be found at https://wiki.eufus.eu.

1.7.3 Support for problems related to the EU-IM Platform and Software

All ITM-specific software and the whole ITM platform is supported by the Core Programming Team (CPT). You can submit trouble tickets to them via the General Support Project in the GForge system. To get more effective help, have a look at the guidelines prepared here: How to report an issue.

To directly submit a trouble ticket, go to: General Support Tracker (https://gforge6.eufus.eu/gf/project/generalsupport/tracker/).

Use this support tracker if your problem falls in the following categories:

- Problems using the UAL, FC2K, HPC2K or similar tools
- Problems running Kepler or Kepler workflows
- Visualization tools: VisIt, Python
- Integrated Simulation Editor (ISE)
- Any software project that is hosted in GForge
- Any kind of scientific software

1.7.4 Feature requests for EU-IM Software

Feature requests for software developed within the ITM can be submitted to a separate tracker.

To submit a feature request, please go to General Feature Request Tracker.

If you are unsure whether to file a bug report of feature request, have a look at these guidlines: How to report an issue.

1.8 Links to related external projects

- EUFORIA Project
- MAPPER Project
- EFDA High Level Support Team (HLST)
- EFDA Goal Oriented Training in Theory (GOTiT)

CHAPTER TWO

INFRASTRUCTURE

The part of the EU-IM documentation addresses infrastructure related issues like the EU-IM Gateway, EU-IM database access, the Universal Access Layer (UAL), and useful tools for handling CPOs.

2.1 Universal Access Layer (UAL)

The UAL (Universal Access Layer) is a multi-language library that allows exchanging Consistent Physical Objects (CPOs) between various modules, and to write to an EU-IM database. The documentation here is provided for rather experienced users who want to practice the UAL in their test programs. Regular KEPLER users do not need to know anything about the UAL. KEPLER manages transparently the UAL calls, which are embedded in the physics code wrappers. No UAL calls should be made inside physics modules. Prior using the UAL, the environment must be configured. It is recommended to use the `EU-IMv1` script for this, which simultanesouly sets i) the database environment (to the private database of the user) ii) the UAL libraries environment iii) the Kepler environment.

```
source $EU-IMSCRIPTDIR/EU-IMv1 KEPLERFOLDER
MACHINENAME DATAVERSION
```

e.g.:

```
source $EU-IMSCRIPTDIR/EU-IMv1 kepler tore_supra 4.08a
```

This scripts does not prevent you from using databases from other users or the public one, you must then use the UAL function `euitm_open_env` in your program to do so. The EU-IMv1 script uses the two following scripts: to set the database environment variables (mandatory prior UAL usage), use:

```
source $EU-IMSCRIPTDIR/set_itm_data_env USERNAME
 MACHINENAME DATAVERSION
```

e.g.:

```
source $EU-IMSCRIPTDIR/set_itm_data_env myname jet 4.08a
```

Then to set the path to the right UAL libraries, use:

```
source $EU-IMSCRIPTDIR/set_itm_env DATAVERSION
```

e.g.:

```
source $EU-IMSCRIPTDIR/set_itm_env 4.08a
```

UAL libraries are installed in /afs.eufus.eu/isip/project/switm/ual. The source code is stored in a subversion repository in /afs/edfa-itm.eu/isip/project/portal/gforge/storage/svnroot/ual. To check out a subversion working copy of the repository, storing it in subdirectory *ual*, do

```
svn co https://gforge6.eufus.eu/svn/ual
```

2.2 Handling CPOs

The tools below allow you to perform useful operations on the EU-IM datastructures including entire CPOs in your Fortran90 workflows. They are not meant as a replacement of the Universal Access Layer (UAL) but rather as a complement especially in situations where the UAL is not available or not practical, e.g. in U.S. - EFDA collaborations.

2.2.1 Module deallocate_stuctures

This Fortran90 module allows you to deallocate in a very simple way any EU-IM data structure which is defined in euitm_schemas.f90. Add the following use statement to your code:

```
use deallocate_structures
```

The Fortran syntax for deallocating a cpo is then:

```
call deallocate_cpo(cpo)
```

where cpo is a single time slice or a pointer array of a CPO (or other EU-IM data structure). With

```
call set_deallocate_verbosity(verbosity)
```

you can set a verbosity level for the deallocate routines. verbosity = 0 produces no output, whereas verbosity > 0 produces verbose output. The module deallocate_structures.f90 is hosted by the Gforge project itmshared. Check it out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/tools target_dir
```

Two static libraries libdeallocate_pgi.a and libdeallocate_g95.a have been prebuilt on the EU-IM Gateway .

2.2.2 Module copy_structures

This Fortran90 module allows you to copy in a very simple way any EU-IM data structure which is defined in euitm_schemas.f90. Add the following use statement to your code:

```
call set_deallocate_verbosity(verbosity)
```

you can set a verbosity level for the deallocate routines. verbosity = 0 produces no output, whereas verbosity > 0 produces verbose output. The module deallocate_structures.f90 is hosted by the Gforge project itmshared. Check it out with

```
use copy_structures
```

The Fortran syntax for copying a cpo is then:

```
call copy_cpo(cpo_source, cpo_target)
```

where cpo_source and cpo_target are single time slices or arrays of a CPO (or other EU-IM data structure) of the same derived type, real scalars, or real arrays (1D - 7D). The allocation of the elements of the target structure is done automatically. With

```
call set_copy_verbosity(verbosity)
```

you can set a verbosity level for the copy routines. verbosity = 0 produces no output, whereas verbosity > 0 produces verbose output. The module copy_structures.f90 is hosted by the Gforge project itmshared . Check it out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/tools target_dir
```

Two static libraries libcopy_pgi.a and libcopy_g95.a have been prebuilt on the EU-IM Gateway .

2.2.3 Module euitm_copy

This Fortran90 module allows you to copy in a very simple way any EU-IM data structure which is defined in euitm_schemas.f90 including entire trim traces. Add the following use statement to your code:

```
use euitm_copy
```

The Fortran syntax for copying a cpo via assignment is then:

```
cpo_target = cpo_source
```

where cpo_source and cpo_target are single time slices or arrays of a CPO (or other EU-IM data structure) of the same derived type. The allocation of the elements of the target structure is done automatically. The program files are hosted by the Gforge project itmshared . Check them out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/perlcopy target_dir
```

To build the Fortran90 module, run

```
creacopy.pl
```

It takes euitm_schemas.f90 from the directory $UAL/fortraninterface/ . Or supply the file from a given directory

```
creacopy.pl $MYDIR/euitm_schemas.f90
```

2.2.4 Module is_set_structures

This Fortran90 module can be used to check whether EU-IM data structures including entire CPOs have been set. The subroutines in is_set_structures.f90 write out the name of each element in the data structure together with ‘T’ if it has been set or ‘F’ if not. Add the following use statement to your code:

```
use is_set_structures
```

The Fortran syntax for checking a cpo is then:

```
call is_set_cpo(cpo, "name of cpo")
```

where cpo is a single time slice or an array of a CPO (or other EU-IM data structure) and “name of cpo” is a string containing the name of the CPO. The module is_set_structures.f90 is hosted by the Gforge project itmshared . Check it out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/tools target_dir
```

Two static libraries libis_set_pgi.a and libis_set_g95.a have been prebuilt on the EU-IM Gateway .

2.2.5 Module size_of_structures

The subroutines in size_of_structures.f90 write out the name of each element in the data structure with its size, the size of each entire substructure, and the size of the entire CPO. The size can be given in bytes or in a more human friendly format depending on the value of the logical parameter human_readable . The indentation is done in steps of 2 blanks with an initial indentation of 1. The maximum depth to which the results are displayed is specified by a call to set_size_of_maxlevel . Output of empty fields can be suppressed by setting the verbosity to zero with a call to set_size_of_verbosity . In all cases, sums are carried out over all levels. Add the following use statement to your code:

```
use size_of_structures
```

The Fortran syntax for calculating the size of a cpo is then:

```
call size_of_cpo(cpo, total_size, human_readable, "name of cpo")
```

where cpo is a single time slice or an array of a CPO (or other EU-IM data structure) and “name of cpo” is a string containing the name of the CPO. total_size is an integer and should be set to zero before the call. human_readable is a flag (true => human friendly format). Set the verbosity with:

```
call set_size_of_verbosity(verbosity)
```

verbosity = 0 => no output of empty fields verbosity > 0 => full output Set the maximum depth with:

```
call set_size_of_maxlevel(level)
```

with level being an integer. The module size_of_structures.f90 is hosted by the Gforge project itmshared . Check it out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/tools target_dir
```

Two static libraries libsize_of_pgi.a and libsize_of_g95.a have been prebuilt on the EU-IM Gateway.

2.2.6 Module write_structures

This Fortran90 module can be used to write EU-IM data structures including entire CPOs to disk. The corresponding file is opened with

```
call open_write_file(unit_no, file_name)
```

where unit_no is the file handle (integer) and file_name a string with the file name (possibly including the path). The file is closed with

```
call close_write_file
```

Add the following use statement to your code:

```
use write_structures
```

The Fortran syntax for writing a cpo to disk is then:

```
call write_cpo(cpo, "name of cpo")
```

where cpo is a single time slice or an array of a CPO (or other EU-IM data structure) and “name of cpo” is a string containing the name of the CPO. With

```
call set_write_verbosity(verbosity)
```

you can set a verbosity level for the write routines. verbosity = 0 produces no output, whereas verbosity > 0 produces verbose output. The module write_structures.f90 is hosted by the Gforge project itmshared . Check it out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/tools target_dir
```

Two static libraries libwrite_pgi.a and libwrite_g95.a have been prebuilt on the EU-IM Gateway .

2.2.7 Module read_structures

This Fortran90 module can be used to read EU-IM data structures including entire CPOs from disk. The corresponding file is opened with

```
call open_read_file(unit_no, file_name)
```

where unit_no is the file handle (integer) and file_name a string with the file name (possibly including the path). The file is closed with

```
call close_read_file
```

Add the following use statement to your code:

```
use read_structures
```

The Fortran syntax for reading a cpo from disk is then:

```
call read_cpo(cpo, "name of cpo")
```

where cpo is a single time slice or an array of a CPO (or other EU-IM data structure) and “name of cpo” is a string containing the name of the CPO. The module automatically deallocates any fields already allocated in cpo and allocates all required fields automatically. It is absolutely essential that “name of cpo” is identical with the one chosen when the cpo was written. With

```
call set_read_verbosity(verbosity)
```

you can set a verbosity level for the read routines. verbosity = 0 produces no output, whereas verbosity > 0 produces verbose output. The module read_structures.f90 is hosted by the Gforge project itmshared . Check it out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/tools target_dir
```

Two static libraries libread_pgi.a and libread_g95.a have been prebuilt on the EU-IM Gateway .

2.2.8 Module diff_structures

This Fortran90 module can be used to compare two CPOs or other EU-IM data structures. It was developed to facilitate benchmarks and automated test suites for the code development. It was kept flexible through the use of function arguments in the argument list of the subroutines of diff_structures. This allows the user to specify his own function set for the analysis and evalutation of the differences between the two CPOs. A call to diff_cpo simply writes out the result of this user defined function. Add the following use statements to your code:

```
use diff_structures
use error_analysis
```

The Fortran syntax for calculating the differences between two cpos is then:

```
call diff_cpo(reference_cpo, test_cpo, name_root, func)
```

where reference_cpo is the reference CPO or other EU-IM data structure and test_cpo is the test CPO or other EU-IM data structure. name_root is a string which defines the root of the field names to be displayed, e.g. 'equilibrium'. func is a function argument to the subroutine diff_cpo. It can be any user defined function with the following constraints:

- It must be defined inside the module error_analysis (an example version with various error analysis functions is provided in error_analysis.f90).
- It follows the structure (dummy arguments, interface, overloading) as demonstrated in error_analysis.f90. The function always has a header function with a list of optional dummy arguments. Depending on which actual arguments are specified, this functions calls the overloaded function with the correct arguments. The interim function is required because of Fortran90/95 limitations. The actual error analysis is carried out inside the overloaded functions. Two fields of these functions are intent(inout) variables:

  ```
  diff_counter : to count the number of difference
  error_level  : to allow for sums or averages over entire CPOs (see examples)
  ```

  These two variables are private to the error_analysis module. To access them please use the functions

  ```
  get_diff_counter()
  ```

  and

  ```
  get_error_level()
  ```

  The function

  ```
  set_error_level(err_level)
  ```

  may be used to specify an initial value for the variable error_level .

With

```
call set_diff_verbosity(verbosity)
```

you can set a verbosity level for the diff routines. verbosity = 0 produces no output, whereas verbosity > 0 produces verbose output. The file check_equilibrium.f90 represents a simple example for a program to compare two equilibrium CPOs one of which is used as a reference for test cases in code development. It clearly demonstrates the use of the diff_structures module. The module diff_structures.f90 and the auxiliary file error_analysis.f90 and check_equilibrium.f90 are hosted by the Gforge project itmshared . Check them out with

```
svn checkout https://gforge6.eufus.eu/svn/itmshared/branches/tools target_dir
```

Two static libraries libdiff_pgi.a and libdiff_g95.a have been prebuilt on the EU-IM Gateway .

CHAPTER THREE

CORE TRANSPORT SIMULATOR (ETS)

3.1 ETS source in FORTRAN

You can checkout the FORTRAN ETS workflow from gforge / project ETS following instructions from ETS User Guide

If you did not use ETS before, first you need to request access to the code via the EFDA EU-IM Portal by following the GForge tab, following the project ETS and requesting access.

Once you have access to the code, it can be checked out of SVN using

```
svn co https://gforge6.eufus.eu/svn/ets
```

to access the whole repository, or

```
svn co https://gforge6.eufus.eu/svn/ets/trunk/ETS
```

to access just the trunk version of the ETS.

The ETS project on Gforge also includes:

- A wiki
- Some documentation
- Trackers
- News
- Mailing lists
- The SVN repository (web interface)

3.2 Documentation for the ETS

- Current ETS Timeline (PDF) (MS Project)
- Description of the ETS
- Form of the standardize equations
- ETS User Guide
- ETS Status
- ETS Doxygen Documentation (PDF) (HTML)

- Pellets in ETS

3.3 Presentations that discuss the ETS

- Presentation at ICNSP-2009 on the ETS
- Movie from the presentation showing the evolution of the flux surfaces
- Movie from the presentation showing the evolution of the plasma

3.4 ETS Verification & Validation

3.5 Other ETS related information

- Visualization of the repository activity (x264)
- Visualization of the repository activity (wmv2)

CHAPTER FOUR

CODES

4.1 IMASviz

The IMASViz code is used for IMAS visualisation.

4.2 FC2K

FC2K is a tool for wrapping a Fortran or C++ source code into a Kepler actor. Before using it, your physics code should be EU-IM-compliant (i.e. use CPOs as input/output).

After running the EU-IMv1 script (to properly set up the environment variables), FC2K can be run simply by typing fc2k in the Linux command line.

`fc2k` was developed by ISIP in Java/Pytho. The program source is stored in the Gforge subversion repository fc2k. To check out a subversion working copy of the repository, storing it in subdirectory *fc2k*, do

```
svn co https://gforge6.eufus.eu/svn/fc2k
```

Executables etc are in $SWEU-IMDIR/ihm/fc2k/.

There are a few tools for managing actors, in $EU-IMSCRIPTDIR/ (put it in your $PATH):

- rmactor : remove an actor from your Kepler version ($KEPLER).
- extract_actor : export an actor from $KEPLER.
- import_actor

  ```
  cd $KEPLER
  ant buildkarlib
  ```

4.2.1 How to turn a C++ code into a Kepler actor

This document is based on material provided by Yann Frauel and describes how to make your C++ code EU-IM compliant and how to turn it into a Kepler actor.

Adapt your C++ function

You must include the header file UALClasses.h:

```
#include "UALClasses.h"
```

The function arguments that are arrays or strings must be declared as pointers, as usual. All other arguments must be passed by reference (i.e. they must be declared with an ampersand):

```
void mycppfunction(double * vector, char * string, int & scalar)
```

The function arguments that are CPOs must be declared with types ItmNs::Itm::cpo_type or ItmNs::Itm::cpo_typeArray. The first form is for time-independent CPOs or a single slice of a time-dependent CPO. The latter is for a complete time-dependent CPO. Note that in all cases, the CPO is considered as a single object, not an array, so it must be passed by reference as mentioned above:

```
void mycppfunction(
ItmNs::Itm::limiter & lim,
ItmNs::Itm::coreimpur & cor,
ItmNs::Itm::ironmodelArray & iron)
```

The syntax is identical for input and output arguments. For output CPOs, do not forget to use the usual methods to assign strings and allocate arrays:

```
lim.datainfo.dataprovider.assign("test_limiter");
iron.array.resize(3);
iron.array(j).desc_iron.geom_iron.npoints.resize(3);
```

Otherwise, the content of CPOs is accessed as usual:

```
cout << lim.datainfo.dataprovider << endl;
cout << iron.array(j).desc_iron.geom_iron.npoints(i);
```

How to use code parameters

The code parameters are passed as the last argument with ItmNs::codeparam_t& type:

```
void mycppfunction(..., ItmNs::codeparam_t & codeparam)
```

Each field of the param structure is a vector of 132-byte strings, not necessarily terminated by 0-character! (This does not follow C/C++ standards and should be changed in the future.)

Compile your function as a library

You need to include the header directories for the UAL and Blitz:

```
-I$(UAL)/include -I$(UAL)/lowlevel -I$(UAL)/cppinterface/ -I/afs/efda-
itm.eu/gf/project/switm/blitz/blitz-0.9/include/
```

Same for linking:

```
-L$(UAL)/lib -lUALCPPInterface -lUALLowLevel -L/afs/efda-
itm.eu/gf/project/switm/blitz/blitz-0.9/lib -lblitz
```

Additionally, you must compile with the -fPIC option.

Full example

We want to generate an actor that has three different types of actors as inputs and three different types of actors as output. Additionally, we have an integer as input/output, a vector of doubles as output and a string as output. We also want to use code parameters. Content of mycppfunction.cpp:

```
#include "UALClasses.h"

typedef struct {
      char **parameters;
      char **default_param;
      char **schema;
} param;

void mycppfunction(
      ItmNs::Itm::summary  SUM,
      EU-IMNS::EU-IM::ANTENNAS & ANT,
      EU-IMNS::EU-IM::EQUILIBRIUMARRAY & EQ,
      INT & X,
      EU-IMNS::EU-IM::LIMITER & LIM,
      EU-IMNS::EU-IM::COREIMPUR & COR,
      EU-IMNS::EU-IM::IRONMODELARRAY & IRON,
      DOUBLE * Y,
      CHAR * STR,
PARAM & CODEPARAM)
{

      /* DISPLAY FIRST LINE OF PARAMETERS */
      COUT &LT< codeparam.parameters[0] << endl;
      cout << codeparam.default_param[0] << endl;
      cout << codeparam.schema[0] << endl;
      /* display content of inputs */
      cout << "x=" << x << endl;
      cout << sum.time << endl;
      cout << sum.datainfo.dataprovider << endl;
      cout << ant.datainfo.dataprovider << endl;
      cout << eq.array(0).datainfo.dataprovider << endl;
      for (int k=0; k<3; k++) {
           for (int i=0; i<4; i++) {
                 cout << eq.array(k).profiles_1d.psi(i)<< " ";
           }
           cout << endl;
      }
      /* fill limiter CPO */
      lim.datainfo.dataprovider.assign("test_limiter");
      lim.position.r.resize(5);     // allocate vector
      for (int i=0; i<5; i++) {
            lim.position.r(i)=(i+1);
      }
      /* fill coreimpur CPO */
      cor.datainfo.dataprovider.assign("test_coreimpur");
      cor.flag.resize(3);          // allocate vector
      for (int i=0; i<3; i++) {
            cor.flag(i)=(i+1)*10;
      }
      cor.time=0; // don't forget to fill time for time-dependent CPOs
      /* fill ironmodel CPO */
      iron.array.resize(3);        // allocate slices
      for (int j=0; j<3; j++) {
            char s[255];
            sprintf(s,"test_ironmodel%d",j);
            iron.array(j).datainfo.dataprovider.assign(s); // allocate vector
            iron.array(j).desc_iron.geom_iron.npoints.resize(3);
            for (int i=0; i<3; i++) {
                  iron.array(j).desc_iron.geom_iron.npoints(i)=j*i;
            }
            iron.array(j).time=j;       // fill time for time-dependent CPOs
      }
      /* assign value to non CPO outputs */
      x=5;
      for (int i=0; i<10; i++) {
            y[i]=i;
      }
      strcpy(str,"This is a test string");
}
```

Content of Makefile:

```
CXXFLAGS=-g -fPIC -I$(UAL)/include -I$(UAL)/lowlevel -I$(UAL)/cppinterface/
-I$SWEU-IMDIR/blitz/blitz-0.9/include/
LDFLAGS=-L$(UAL)/lib -lUALCPPInterface -lUALLowLevel -L/afs/efda-
itm.eu/gf/project/switm/blitz/blitz-0.9/lib -lblitz
libmycppfunction.a: mycppfunction.o
      ar -rvs libmycppfunction.a mycppfunction.o
mycppfunction.o: mycppfunction.cpp
clean:
      rm mycppfunction.o libmycppfunction.a
```

How to fill the FC2K window

First tab (Argument):

- set number of input and output arguments (combined)
- select type of arguments from drop-down menu
- tick if argument is a single time slice
- tick if argument is array (not for pointers)
- if necessary define size of arrays
- tick if argument is input argument
- tick if argument is output argument (multiple ticks possible)

The fields Kepler, Ptolemy, and UAL are automatically filled with the values which you set by running the `EU-IMv1 script`.

Second tab (HasReturn):

- specify return parameters (type, array, size)

Third tab (HasParameters):

- tick if subroutine uses code specific parameters
- specify (or browse for) XML code parameter input file
- specify (or browse for) XML default code parameter file
- specify (or browse for) W3C XML schema file (XSD)

For information on code specific parameters, please see *How to handle code specific parameters*.

Fourth tab (Source):

- specify programming language of source code
- select appropriate compiler
- tick Parallel MPI if code module is using MPI
- tick Batch if code module shall be run in batch mode rather than interactively when running Kepler workflows

- specify (or browse for) library file containing the code module
- specify (or browse for) other libraries required by the code module

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search