

IPS Driver Component and Plasma State

D. B. Batchelor

Proto-FSP Frameworks Workshop

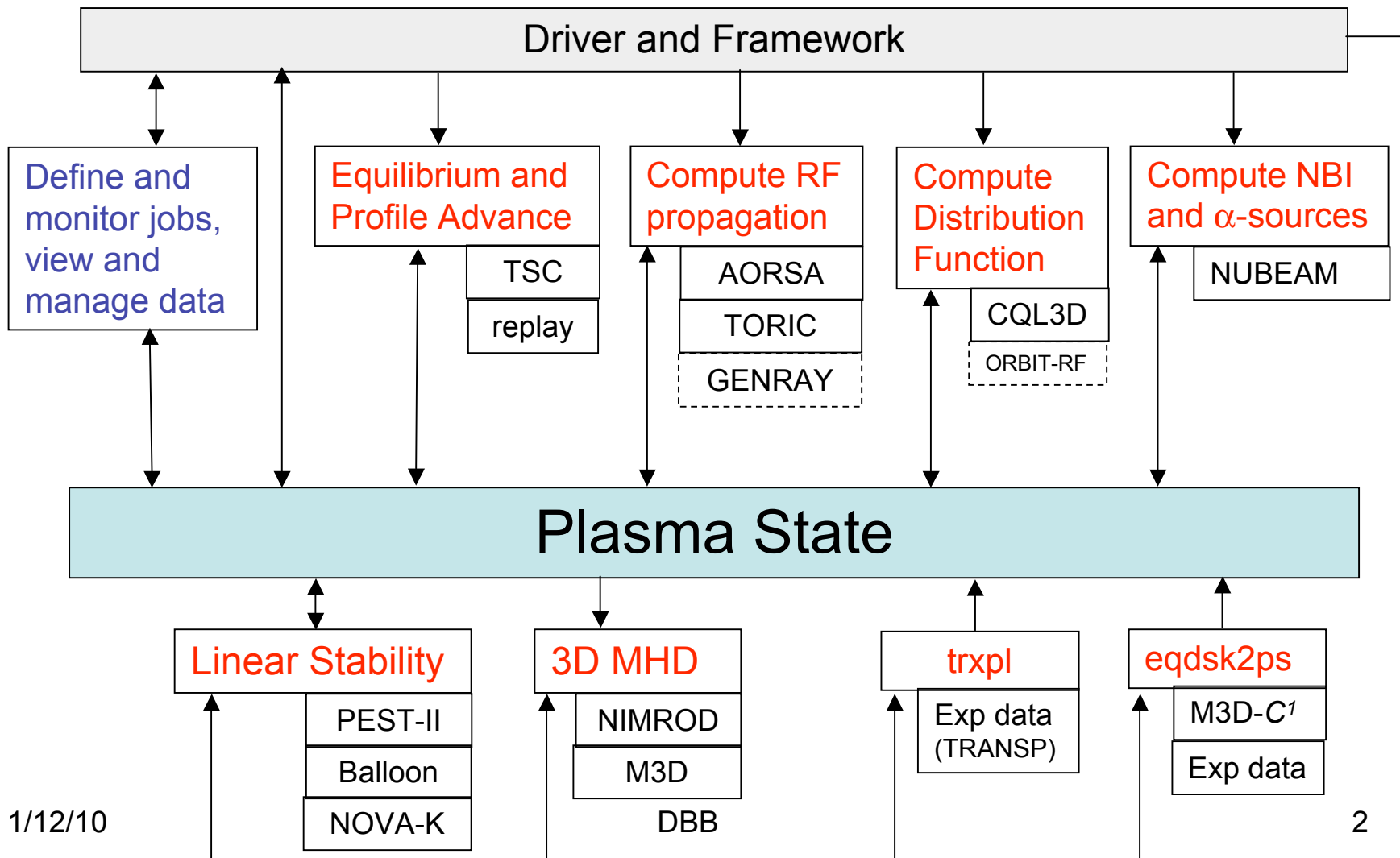
January, 11-13, 2010

ORNL

- **Plasma state**
- **Drivers**
- **Portal/ELVis example (maybe)**



A Physicists View of the IPS



All Simulation data exchanged between components goes through Plasma State – but components can produce other files

In a narrow sense “Plasma State” is a flat data structure

- **Fortran 90 Module – now at version 2.023 (also C++ interface)**
 - Distributed by NTCC, used in TRANSP, PTRANSF, FACETS ...
 - netCDF for backend storage
- Supports **multiple state instances** (very important!) → e.g. current/prior state, pre-/post-sawtooth, etc
- PS data conventions (names, units, etc.) for IPS determined by **(benevolent) dictator** → extended as needed
- Data stored “as produced” → **Consumer is responsible for adapting units/grids as needed**
- Code is automatically generated from state specification text file → **ease and accuracy of update**
- Some types of data we don’t know how to deal with yet → **distribution functions are just code dependent filenames**

In a broader sense “Plasma State” of a simulation is a collection of files that are moved as needed by the framework

- **Current Plasma State file**
- **Previous Plasma State file**
- **Next Plasma State file**
- **Current eqdsk file**
- **Current dql file (quasilinear operator file used by AORSA and CQL3D)**
- **Current cql file (distribution function file used by AORSA and CQL3D)**
- **Current jsdsk file (used by linear MHD component)**

What does the Plasma State software consist of?

A data structure that contains data shared by components

- **Time evolving data – scalars, 1D profiles, 2D profiles, file paths to more complicated data (e.g. eqdsk file)**
 - Species densities and temperatures, magnetic field, source profiles like heat deposition or driven current
 - A plasma state instance contains one set of data (time slice) not a time history. A time history consists of many plasma states
 - Data is grouped by component and all data is “owned” by a specific component
 - During a time step each component updates its own data as it finishes – data is only ‘consistent’ at the beginning and end of a valid time step
- **Static data that is common between components – tokamak geometry, definition of auxiliary systems such as RF and neutral beam characteristics ...**

A collection of routines defining the user interface

- **Functions to store and access state data → get, store, update, commit**
- **Functions to initialize plasma state structure and to manipulate data → allocate state arrays, copy plasma state objects, grid interpolation**

The basic interface is simple – simple things are easy

Initializing some plasma state arrays and storing them

```
! Get plasma state definition
USE plasma_state_mod
```

This gets you all plasma state data declarations and access to all functions

```
! Get the current plasma state
call ps_get_plasma_state(ierr, trim(cur_state_file))
```

This gets you all plasma state data

```
! Set dimensions of some state arrays and allocate them
ps%nspec_th = nspec_th
ps%nrho = nrho
```

State arrays are allocated incrementally. This allocates arrays with newly assigned non-zero dimensions

```
.....
CALL ps_alloc_plasma_state(ierr)
```

```
! Set values for thermal species temperature profiles
DO i = 1, nspec_th
  CALL profgen( ti0, ti_edge, alpha_Ti, zone_center, ps%Ts(:,i) )
END DO
```

```
! write plasma state data file
```

```
CALL ps_store_plasma_state(ierr, cur_state_file)
```

This stores a complete Plasma State in a netcdf file

But there are dozens of routines to do more specialized things

Other Nifty Plasma State features

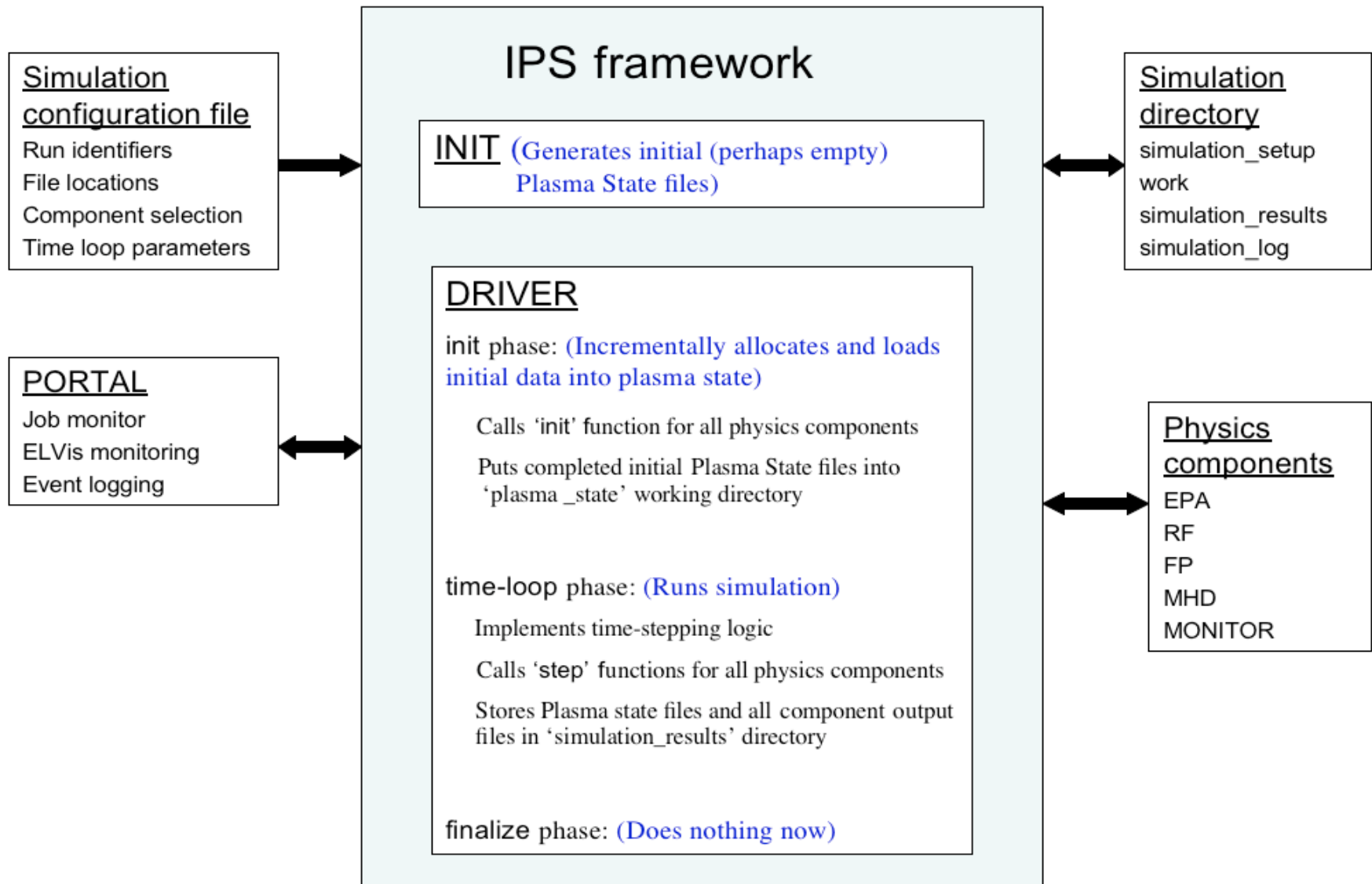
Examples of other useful Plasma State functions

- Write or read a partial plasma state files containing only items that have changed
- Merge data from two plasma state instances
- Copy plasma state instances
- Write an eqdsk file from plasma state magnetic equilibrium data or set plasma state magnetic equilibrium data from an eqdsk file
- Interpolate profile data onto different radial grids with choice of interpolation method – match radial grids on components with those of Plasma State
- Functions to facilitate definitions and management of plasma particle species – thermal species, neutral beam species, fusion product species, impurities ...

Documentation

- Automatically generated plasma state content index
 - Check out in the repository:
https://cswim.org/svn/cswim/ips/trunk/components/state/src/plasma_state/
 - Alphabetical index of all plasma state variables – [plasma_state_index.txt](#)
 - Alphabetical index arranged by component – [ps_component_index.txt](#)

What does a driver do?



World's simplest IPS driver

```
#!/usr/bin/env python

from component import Component

class HelloDriver(Component):
    def __init__(self, services, config):
        Component.__init__(self, services, config)
        print 'Created %s' % (self.__class__)

    def init(self, timeStamp=0.0):
        return

    def step(self, timeStamp=0.0):
        try:
            worker_comp = self.services.get_port('WORKER')
        except Exception:
            self.services.exception('Error accessing worker component')
            raise
        self.services.call(worker_comp, 'step', 0.0)
        return

    def finalize(self, timeStamp=0.0):
        return
```

Look at a complete driver: driver self initialization

```
#!/usr/bin/env python

# version 3.0 5/12/08 (Batchelor)

import sys
import os
import subprocess
import getopt
import shutil
import math
from component import Component
from Scientific.IO.NetCDF import *
import Numeric

class epa_ic_nb_fus_driver(Component):

    def __init__(self, services, config):
        Component.__init__(self, services, config)
        print 'Created %s' % (self.__class__)

# -----
#
# init function
#
# -----

def init(self, timestamp=0):
    # Driver initialization ? nothing to be done
    return
```

STEP function: setup phase – get component references and time loop, call ‘init’ functions for all components, and store final initialized plasma state

```
def step(self, timestamp=0):

    services = self.services
    self.services.stage_plasma_state()
    self.services.stage_input_files(self.INPUT_FILES)

    # Get references to the components to run in simulation

    monitorComp = services.get_port('MONITOR')
    epaComp = services.get_port('EPA')
    nbComp = services.get_port('NB')
    fusComp = services.get_port('FUS')
    rfComp = services.get_port('RF_IC')

    # Get timeloop for simulation
    timeloop = services.get_time_loop()
    tlist_str = ['%.3f'%t for t in timeloop]
    t = tlist_str[0]

    # Call init for each component
    services.call(epaComp, 'init', t)
    services.call(rfComp, 'init', t)
    services.call(nbComp, 'init', t)
    services.call(fusComp, 'init', t)
    services.call(monitorComp, 'init', t)

    # Post init processing: stage plasma state, stage output
    services.stage_output_files(t, self.OUTPUT_FILES)
```

STEP function: time loop – call STEP function for all components and store final initialized plasma state

```
# Iterate through the timeloop
for t in tlist_str[1:len(timeloop)]:
    print ( ' ')
    print 'Driver: step to time = ', t
    services.update_time_stamp(t)

# call pre_step_logic
services.stage_plasma_state()
self.pre_step_logic(float(t))
services.update_plasma_state()

# Call step for each component
services.call(rfComp, 'step', t)
services.call(nbComp, 'step', t)
services.call(fusComp, 'step', t)
services.call(epaComp, 'step', t)
services.call(monitorComp, 'step', t)

self.services.stage_plasma_state()

# Post step processing: stage plasma state, stage output
services.stage_output_files(t, self.OUTPUT_FILES)
```

Complicated step logic can easily be programmed – in this example only some simple operations are performed at the beginning of each time step

```
def pre_step_logic(self, timeStamp):  
  
    cur_state_file = self.services.get_config_param('CURRENT_STATE')  
    prior_state_file = self.services.get_config_param('PRIOR_STATE')  
    next_state_file = self.services.get_config_param('NEXT_STATE')  
  
    # Copy data from next plasma state to current plasma state  
    shutil.copyfile(next_state_file, cur_state_file)  
  
    # Update time stamps  
  
    ps = NetCDFFile(cur_state_file, 'r+')  
    t1 = ps.variables['t1'].getValue()  
    self.services.log('ps%t1 = ', t1)  
  
    ps.variables['t0'].assignValue(t1)  
    ps.variables['t1'].assignValue(timeStamp)  
  
    ps.close()  
    shutil.copyfile(cur_state_file, prior_state_file)  
  
    return
```

Crude cross-component communication. Next state file from previous time step becomes current state.

Being replaced by framework event service

Update the start and end times of the step in plasma state.

This example shows direct interaction with Plasma State from Python

STEP function: after time loop – call ‘finalize’ function for all components call ‘finalize’ function for driver

```
# Post simulation: call finalize on each component
services.call(monitorComp, 'finalize')
services.call(rfComp, 'finalize')
services.call(nbComp, 'finalize')
services.call(fusComp, 'finalize')
services.call(epaComp, 'finalize')

# -----
#
# finalize function
#
# -----

def finalize(self, timestamp = 0):
    # Driver finalize - nothing to be done
    pass
```

Backup

Steps to running an IPS simulation

Step	Details
Get access to IPS	Check out from svn repository Do top level make
Figure out what physics components you need	Chase down binaries of physics executables – should be in <code>ips/<components></code> or <code>physbin</code> Get appropriate specific input files for the physics codes you plan to use – talk to developer or previous user. These tend to live with component scripts in <code>ips/components/<component_class>/<sub_class></code> of the ips directory
Generate a simulation configuration file	Specify overall simulation configuration: simulation/run identifiers, file naming conventions, what files constitute the plasma state data, what components to use Specify configuration for individual components: names of required input/output files, paths to binaries, number or processors required for executable, any other configuration data you choose
Launch job	<code>bin_path/ips --config= <configuration_file></code> , or appropriate batch script
Sit back and watch on portal	http://swim.gat.com:8000/monitor

In principle it's not complicated. But there is a lot (*bewildering amount*) of flexibility. We are trying to assemble a set of best practices to streamline the process. We need more users

Q. What do you get out of an IPS run?

A. The output directory + monitor file in W3_dir

/Simulation_output (example Run_1_CM0D_8021)

/work – current working files for this time step

/plasma_state – all current plasma state files

/epa

/tsc – all tsc input files, plasma state files → all tsc output files

/rf

/aorsa – all aorsa input files, plasma state files → all aorsa output files

/... other physics components

/simulation_results/history – archived data from all time steps

/plasma_state – plasma state files for all time stamps

/<time 0>**/components**

/epa

/tsc – all tsc output files

/rf

/aorsa – all aorsa output files

/<time 1>**/components**

/... other time steps

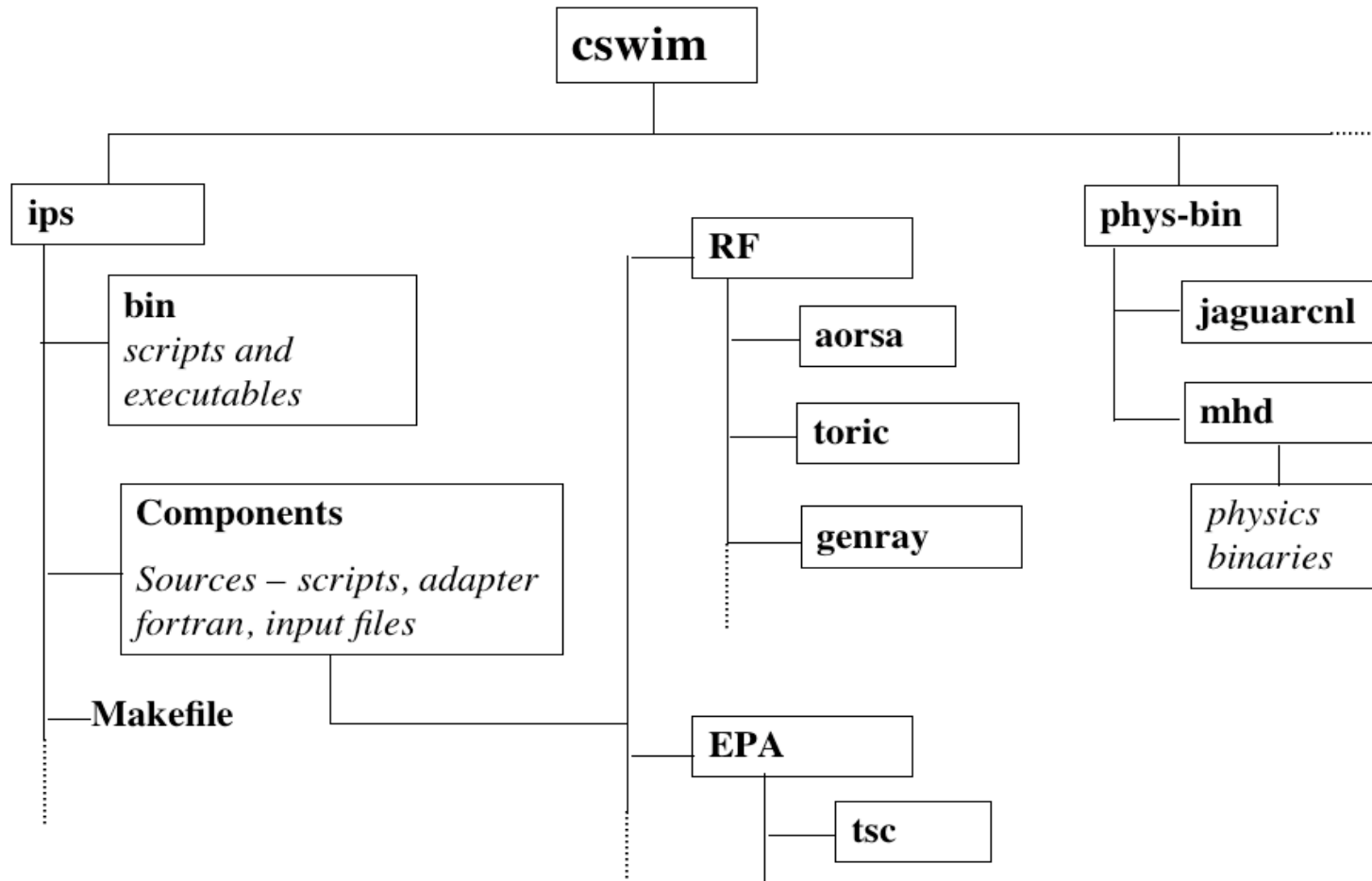
sim.config file

/simulation_setup – all input files at simulation initialization

/simulation_log – events published to the portal by the framework

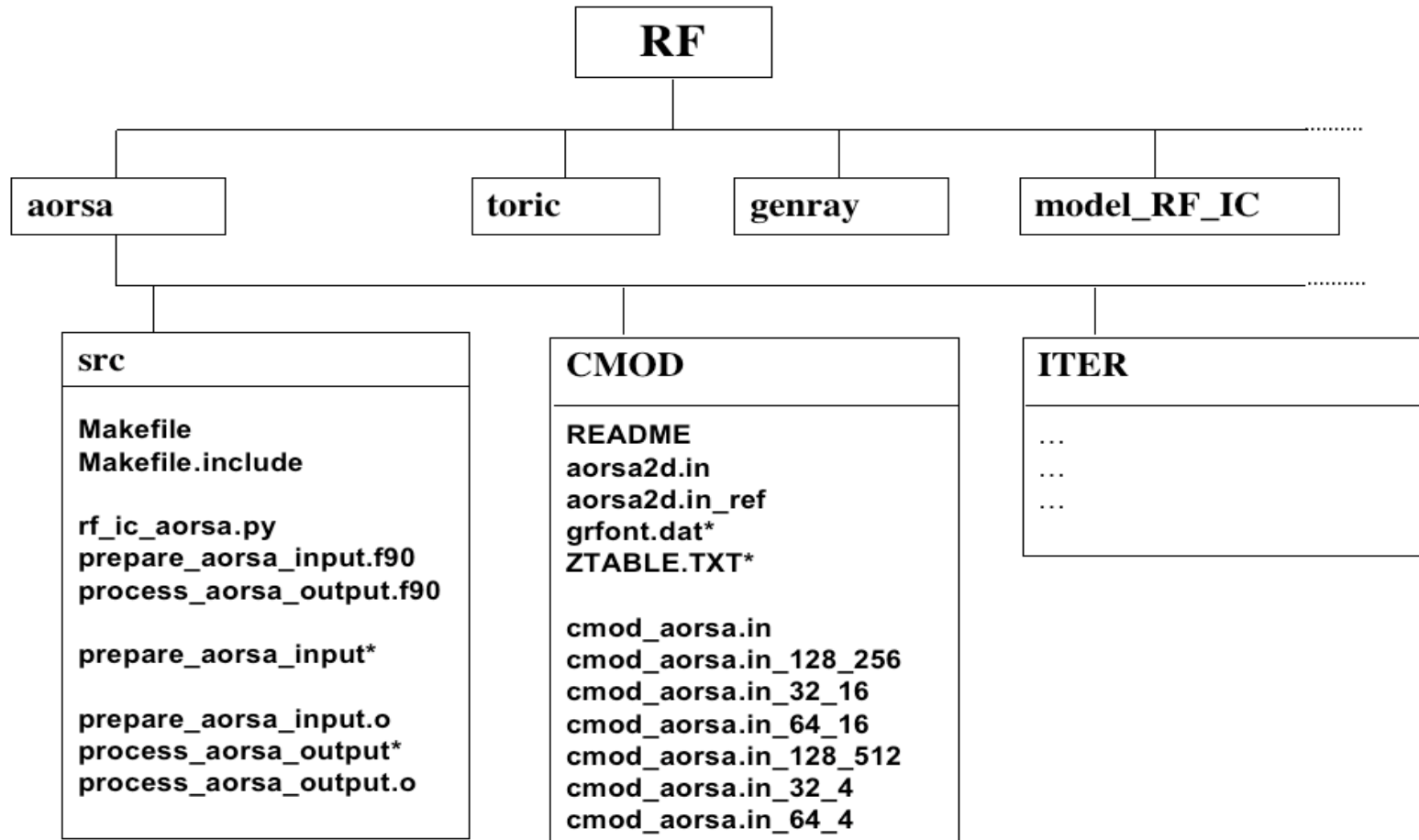
- **The framework builds the name, path and structure for the output directory from information in the config file – so you can change it if you want**

Treasures in the cswim.org svn repository



- **svn is very easy to use**
- **You use the same username/password to access the svn repository as you do for the www.cswim.org website. If you haven't signed up you really should**

Contents of “typical” component directory in svn tree



- **What you won't find in the aorsa/src directory is the AORSA source file or the binary executable**