



The Integrated Plasma Simulation (IPS) Framework

Wael R. Elwasif
elwasifwr@ornl.gov

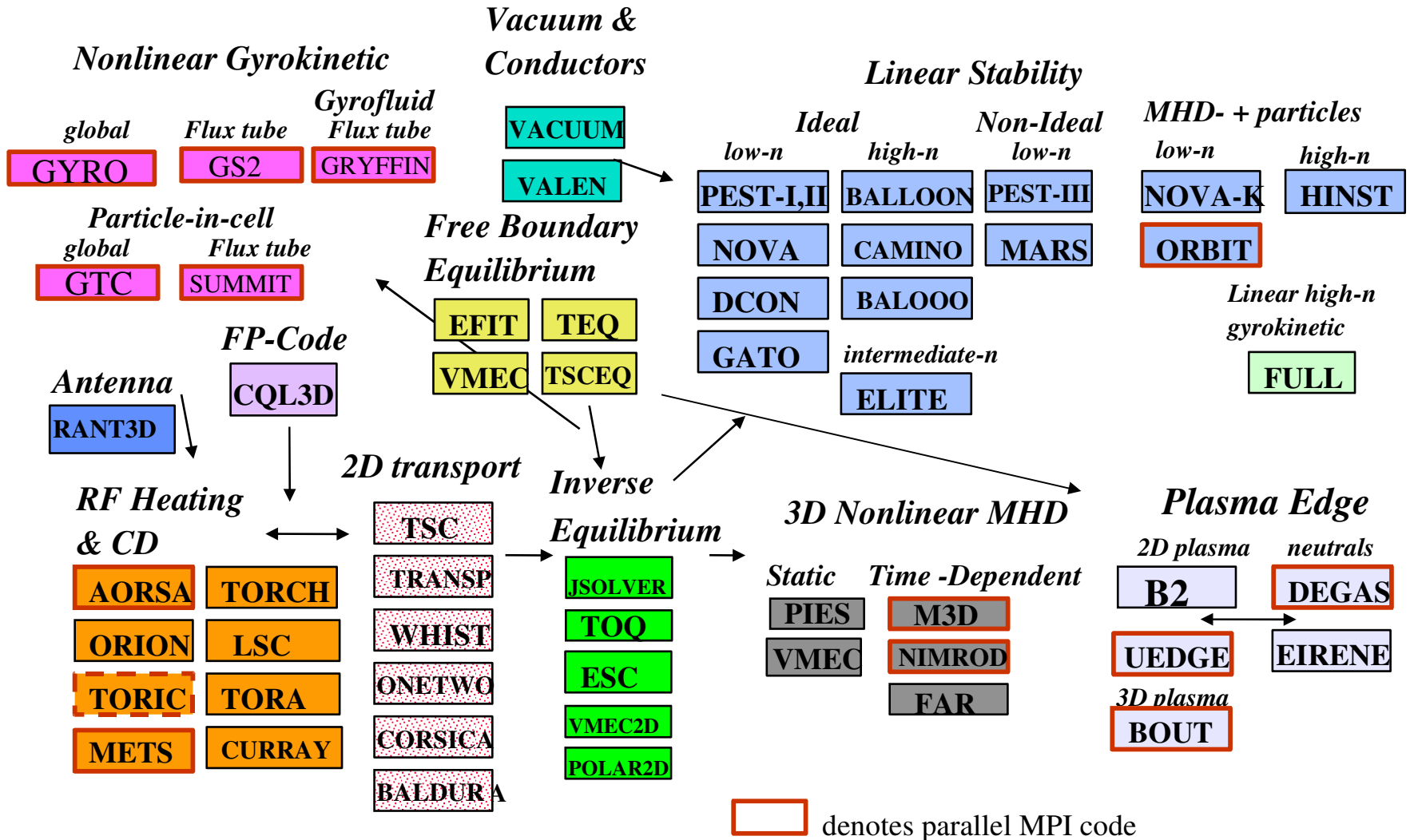
and

The SWIM Project Team

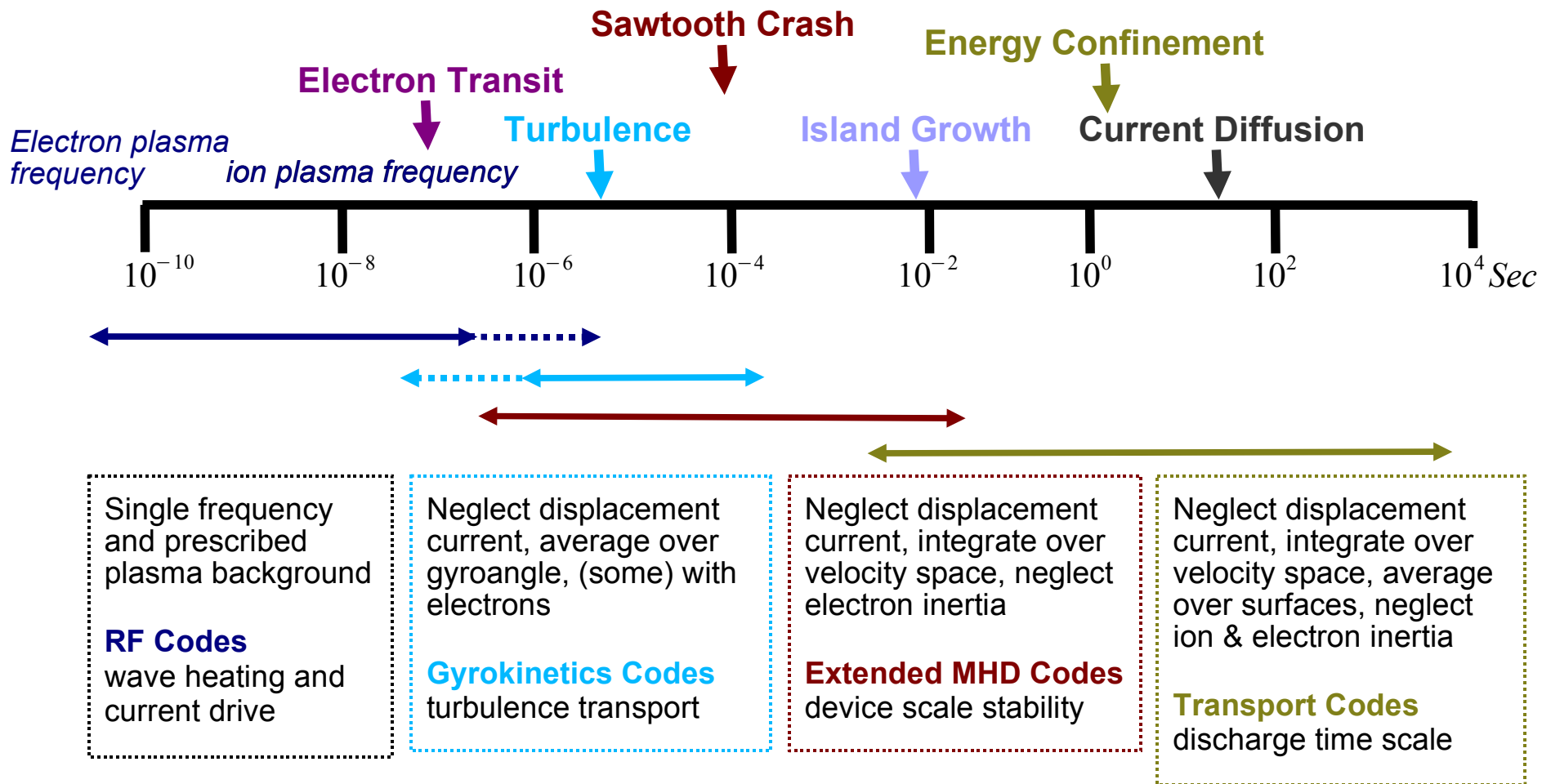
Outline

- Challenges in integrated fusion simulation.
- Integrated Plasma Simulation (SWIM - IPS)
- Goals and requirements.
- System architecture.
- Component interfaces and framework services.
- Simulation configuration management.
- Execution environment.
- FSP Questions

Coupled Simulation Challenges: Codes



Simulation Challenges: Time Scales



The IPS Approach to Coupled Simulation

- Part of c**SWIM**, the center for **S**imulation of RF **W**ave **I**nteractions with **M**agneto hydrodynamics.
- Primary directive:
“Explore the targeted coupled physics interactions while constituent codes evolve independently, minimizing impact on long lived codes and other research/production use”.
- Code re-factoring and/or rewriting ruled out.

Framework Design Guidelines

- Rapid, flexible coupling of *existing* simulation codes.
- Minimal (aka **NO**) change to underlying codes.
 - Not feasible to manage multiple branches.
- Simple coupling and integration protocol.
 - Maintainability, and *debug-ability*.
- Integrated data management.
 - Simulation run as an experiment.
- Extensible simulation structure.
 - New physics added as needed.

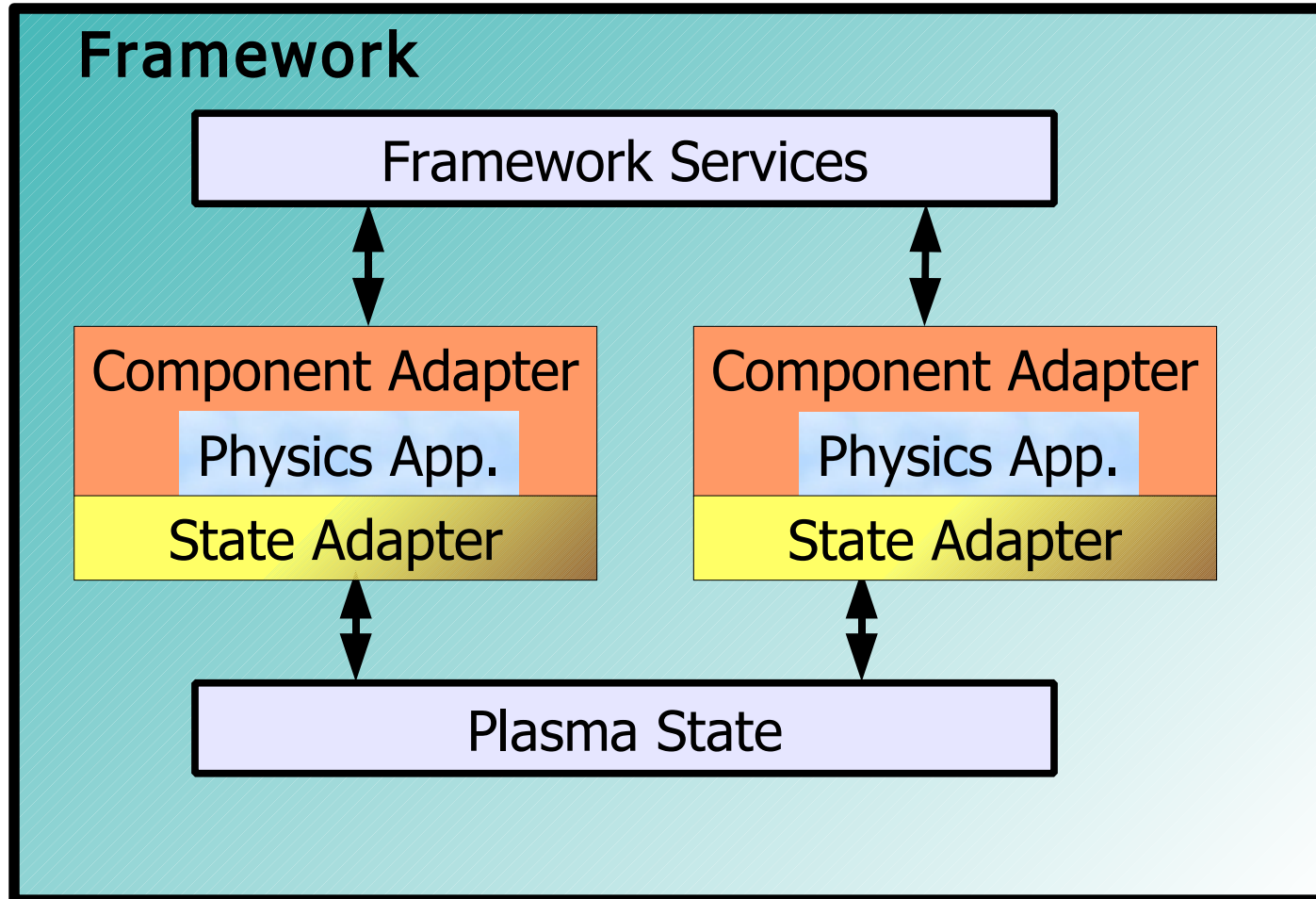
Major Design Features

- Component-based approach
 - *Extensibility*, V&V, independent development.
- Common plasma state layer
 - Data repository.
 - Conduit for inter-component data exchange.
- File-Based data transfer
 - No change to underlying codes.
 - Simplify *"unit testing"*

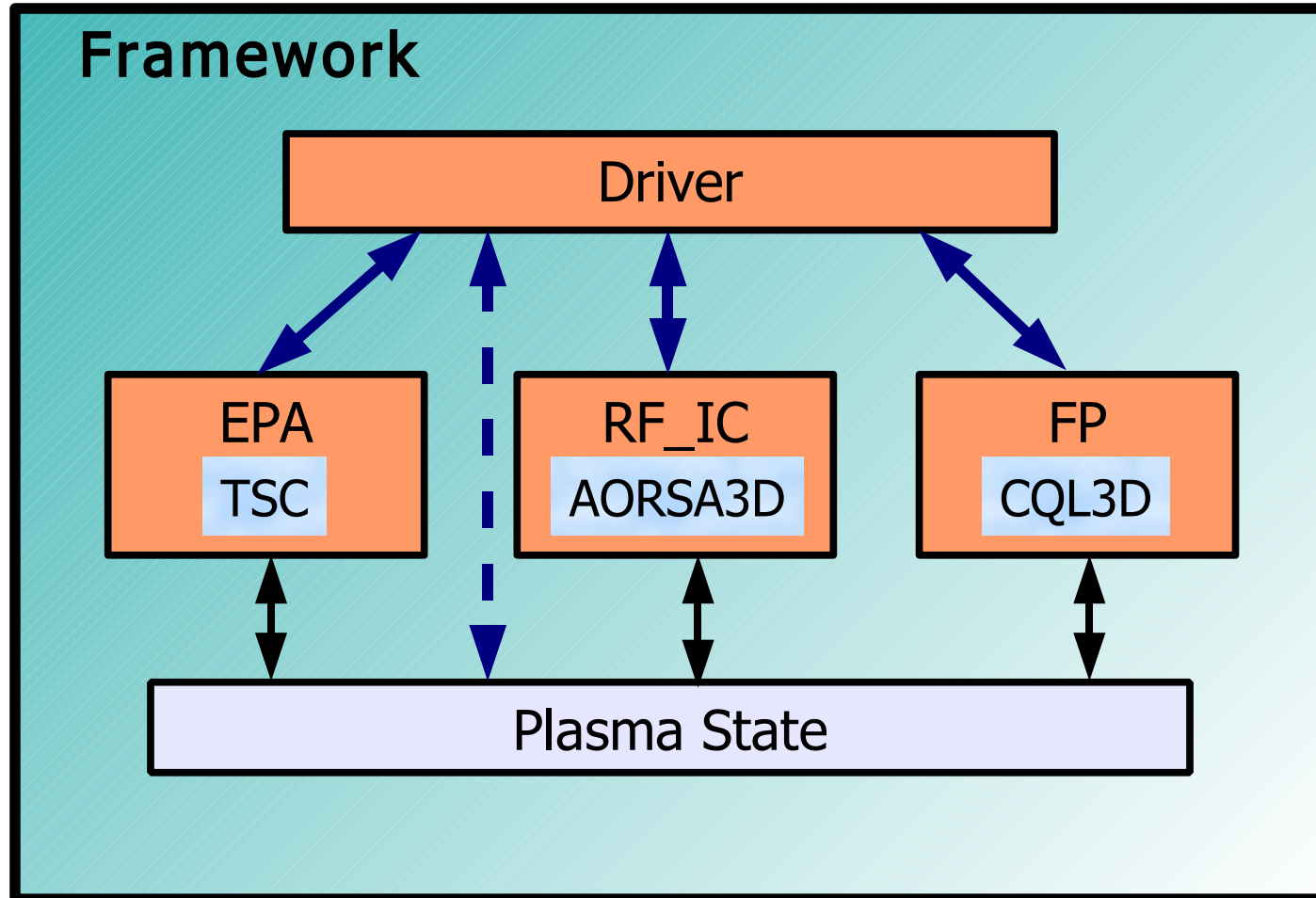
Major Design Features (2)

- Scripting Based Framework (Python)
 - RAD.
 - Adaptability, changeability, and flexibility.
- Simple component connectivity pattern
 - Driver/workers topology.
- Codes as components:
 - Focus on *code-coupling* vs *physics-coupling* as first step.
- Simple unified component interface
 - `init()`, `step()`, `finalize()`. *Too Simple??*

IPS Framework Layout



Sample IPS Application Structure



Framework Responsibilities

- *Configuration Management*
 - Simulation configuration.
 - Component instantiation and connection.
- *Task Management*
 - Mediate inter-component method invocation.
 - Manage execution of underlying applications.
- *Data Management*
 - Stage component input files.
 - Mediate shared access to plasma state files.
 - Archive component output files.

Framework Responsibilities (2)

- *Resource Management*
 - Manage access to computing resources (mainly compute nodes) for concurrent components.
- *Event Management*
 - Support asynchronous publish/subscribe model of data exchange in a running simulation.
- *Simulation Monitoring*
 - *Publish events to web-based SWIM portal.*

IPS Component Structure

- Component adapter:
 - Make a standalone app. into a component.
 - Utilize framework services and implement component interface.
- Underlying application:
 - Use *unchanged* in a coupled simulation.
- Plasma state adapter:
 - Map native app. data into common plasma state.
 - Receiver makes right.
 - Data definition and provenance??.

Simulation Configuration File

- Four major sections:
 - Global configuration options.
 - *Ports* configuration.
 - Components configuration.
 - Time loop specification.
- Platform configuration:
 - Capture needed information for execution platform (currently: Jaguar, Franklin, and PPPL viz/mhd).

Configuration structure can be changed and/or extended easily as needed.

Configuration: Global Data

```
IPS_ROOT = /home/elwasif/ips/trunk           # Root of IPS component and binary tree
SIM_NAME = AORSA_SIM                        # Name of current simulation
SIM_ROOT = $IPS_ROOT/$SIM_NAME             # Where to put results from this
simulation
PLASMA_STATE_WORK_DIR = $SIM_ROOT/work/plasma_state
                                           # Where to put plasma state files as the
simulation evolves
RUN_ID = $SIM_NAME
OUTPUT_PREFIX =
CURRENT_STATE = ${RUN_ID}_ps.cdf
PRIOR_STATE = ${RUN_ID}_psp.cdf
CURRENT_EQDSK = ${RUN_ID}_ps.geq
                                           # What files constitute the plasma
state
PLASMA_STATE_FILES = $CURRENT_STATE $PRIOR_STATE $CURRENT_EQDSK
SIMULATION_MODE = SINGLE_STEP | RESTART    # Simulation mode
```

Configuration: Ports

```
[PORTS]
  NAMES = DRIVER INIT RF_IC EPA LINEAR_STABILITY FOKKER_PLANCK
  [[DRIVER]]                # REQUIRED Port section
    IMPLEMENTATION = AORSA_CQL3D_DRIVER
                                # How is the simulation initialized
                                # (generate the very first state - if needed)
  [[INIT]]                  # REQUIRED Port section
    IMPLEMENTATION = AORSA_CQL3D_INIT

  [[RF_IC]]
    IMPLEMENTATION = AORSA

  [[EPA]]
    IMPLEMENTATION = TSC

  [[FOKKER_PLANCK]]
    IMPLEMENTATION = CQL3D
```

Configuration: Components

```
[AORSA]
CLASS = rf                                # Component categorization
SUB_CLASS = ic
NAME = aorsa                              # Component's Python class name.
NPROC = 4                                 # Number of processors.
BIN_PATH = $IPS_ROOT/bin                 # Where to look for application
                                          # Where to look for input files.

INPUT_DIR = $IPS_ROOT/components/$CLASS/$NAME

                                          # List of input files
INPUT_FILES = aorsa2d.in grfont.dat ZTABLE.TXT g096028.02650

                                          # List of "important" output files
OUTPUT_FILES = out_swim out15 aorsa2d.ps aorsa2d.in

SCRIPT = $BIN_PATH/rf_ic_aorsa.py        # Where to find the component
...
...
# Can add extra component-specific configuration entries here.
```



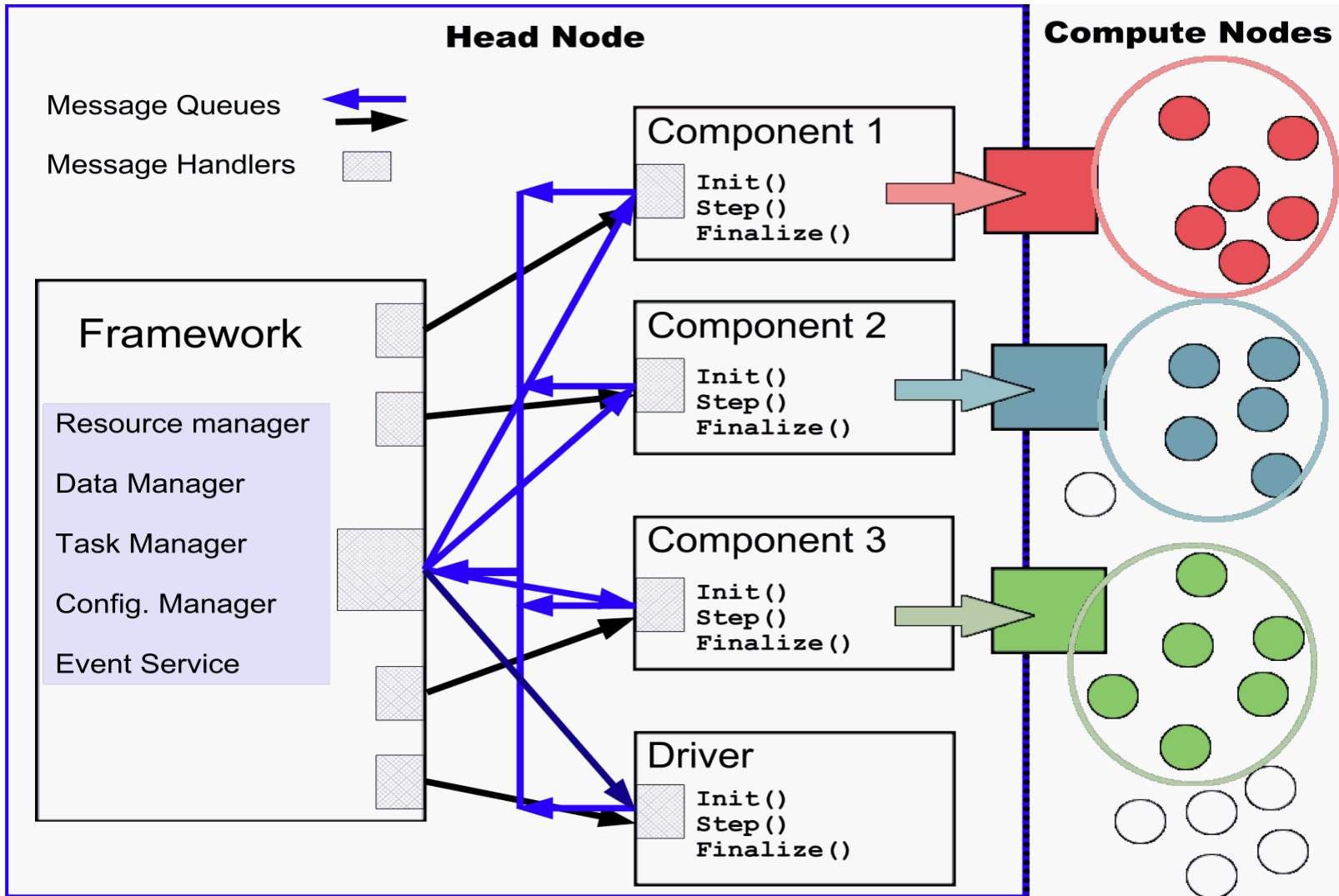
Configuration: Time Loop

```
# For MODE = REGULAR, the framework uses the variables  
# START, FINISH, and NSTEP  
# For MODE = EXPLICIT, the framework uses the variable VALUES  
# (space separated list of time values)  
[TIME_LOOP]  
  MODE = EXPLICIT  
  START = 3.5  
  FINISH = 3.7  
  NSTEP = 2  
  VALUES = 3.4 3.5 3.6 3.7
```

Component Details & Execution

- Separate work directory per component instance.
- Framework always executes component code in the same directory
(`$SIM_ROOT/work/$CLASS_$SUB_CLASS_$NAME_$INSTANCE#`).
- Direct access to component configuration variables (through the Python `self` variable).
- Access to framework services through invocation on the `self.services` variable.

IPS Execution Environment



Simulation Execution

- Framework invoked in batch script, running on the *head node*.
- Framework support multiple concurrent simulation runs:
 - Specified via command line configuration file(s) arguments.
 - Efficient utilization of computing resources in the presence of large variances in code scalability.
- Advanced logging capabilities:
 - Control granularity at individual component level.
 - Separate log file per simulation instance.

Simulation Execution: Plasma State Sharing

- Plasma State:
 - *Coarse granularity, file-based shared memory.*
- *Caching* to individual component work directory.
- Serialize updates to plasma state
 - Done centrally in the data manager.
 - Can be used along with event services to implement simple cache-coherency protocol (if needed).

Framework Services (1)

- Task Manager:

- `call()`
- `call_nonblocking()`
- `wait_call()`
- `launch_task()`
- `wait_task()`
- `wait_task_nonblocking()`
- `kill_task()`
- `kill_all_tasks()`

} Component Method
Invocation

} Application Code
Execution

Framework Services (2)

- Configuration Manager:
 - `get_config_parameter()`
 - `set_config_parameter()`
 - `get_port()`
 - `get_time_loop()`
 - `get_working_dir()`
- Event Management
 - `Publish()` `subscribe()`
 - `unsubscribe()` `process_events()`

Framework Services (3)

- Data Manager:
 - `stage_input_files()`
 - `stage_output_files()`
 - `stage_plasma_state()`
 - `update_plasma_state()`
 - `merge_plasma_state()`
- Logging Services
 - `Debug()` `info()` `warning()`
 - `error()` `exception()` `critical()`

FSP Questions : Users

- Who are your present users?
Code developers? **Yes** Theorists? Experimentalists?
Casual versus "power" users? **Both**
- Who are your intended users?
Code developers, ***Integrated Modelers (simulators)***.
- To what extent, and how, were users (present and intended) consulted in designing your framework?
Users part of the high level design process. Power users drive drive requirements and design updates.
- What kinds of concerns did users bring to the initial design process?
 - Code maintenance, forking.

SWIM Component Roster

- RF_IC
 - AORSA, TORIC
- Fokker-Planck
 - CQL3d
- Equilibrium & Profile Advance
 - TSC, JSOLVER
- Neutron-Beam:
 - NUBEAM
- Linear MHD:
 - PEST II, Balloon, Nova-k
- Miscellaneous:
 - Sim. Monitoring.
- In progress:
 - M3D, M3D-C1, NIMROD, GENRAY, TRXPL, ORBIT-RF



FSP Questions : Users (2)

- Did the initial consultations lead to the specification of use scenarios? How did these impact the framework design?
 - Informal use case specification, design and implementation continuously refined with user feedback
- What kind of ongoing feedback do you get from users, and how?
 - Feature requests, usability feedback.
- What kind of upgrade cycle do you have for your framework, and how does user feedback figure into that?
 - No formal upgrade cycle.

FSP Questions: Framework

- What is the goal of the framework?
 - Enable experimental, efficient coupling of many existing codes in a coherent extensible environment.
- What restrictions influenced the design?
- Minimal or no changes to existing codes.
Evolvability to incorporate new physics/codes as needed.
- What language is the framework written in? Python
- What language constraints does it impose on physics components?
 - Python wrappers of codes in any language.
- What physics components is the framework currently coupling together? See roster on prior slide.

FSP Questions: Framework

- Describe the process of componentization (incorporating a component into the framework)
 - Identify relevant Plasma State Input/Output variables.
 - Code pre and post processors to map native data from/to plasma state.
 - Implement the component API (typically starting from an existing component as a template).
 - Develop the component-specific section of the simulation configuration file.
- How does the framework share data with the physics components?
 - Framework provides data (file) management services.
 - Plasma State primary holder of shared component data.

FSP Questions: Framework

- How does a user create a flow of execution (driver) using the framework?
The driver is a special IPS component that uses framework services to connect to other components, invoking their standard API according to simulation logic.
- How well documented are the standards and assumptions of the framework?
 - Project Web Site, Papers, Presentations (no formal “user guide”).
- How is this framework distributed? (How can the framework analysis team get access?)
 - Project Subversion repository.

FSP Questions: Framework

- What are the steps to build and execute a typical populated framework on an example HPC site ?
 - Framework usable on any Python-enabled HPC system.
 - Components & Build system supported only on SWIM-targeted platforms.
 - “Manual” addition of new platforms.
 - Generalization to arbitrary platforms not attempted.
- What did you take from outside community, and what did you build yourself, and why ?
 - Component concepts adopted from CCA & ESMF.
 - Extensive use of Python modules
 - Build everything else from scratch to meet specific project needs

FSP Questions: Framework

- How do you think your framework meets the needs of the other proto-FSPs ?
 - No attempt made to address the need of other proto-FSP's.
 - Address a need in FSP for flexible, loose coupling capability.
 - “*FSP Framework*” will need to incorporate ideas from all proto-FSPs.
- Possible paths from proto-FSP to FSP
 - Fresh start : Using ideas from proto-FSPs.
 - Evolutionary strategy:
 - Kepler + IPS+ CCA
 - Re-implement IPS in CCA compliant form.
 - Retain proto FSPs, make them interoperable:
 - CCA+Kepler demonstrated
 - IPS is CCA-like, could be made interoperable.

Utilities

- To what extent can utilities be standardized?
 - Standardization = change to underlying apps.
- Utilities usually get their data from output files. How standardized are the individual components in file formats?
 - Plasma State = Standard file format (netcdf)
 - “component-specific” outputs not standardized, they should.
- What language are most utilities written in?
- What utilities are needed to ease the ability to perform verification and validation studies?
- How important is having common visualization tools?
 - Monitoring, “*light weight vis*” (Elvis from PPPL)
 - Analysis, “*heavy duty vis*” (Visit)

Workflow

- What are the current paradigms for workflow management?
 - “*traditional*” **computing** workflow via code in the driver component.
 - Simple **simulation** workflow (SWIM Portal, Elvis monitoring)
 - More sophisticated data management & workflow needed as runs reach critical mass.
- How user-friendly is the workflow?
- How easy is it for users to adapt the workflow software to their needs?
- How different is the workflow between working on a cluster, and working at the LCFs?
- What requirements do the LCFs impose on workflow software?
- How do workflow tools help with verification and validation?



Engineering

- Bug reporting and tracking – Trac
- Communication: project mailing lists, regular conference calls, coding camps, yearly all-hands meetings.
- Cross-platform build systems. make
- Documentation - informal (wiki, papers, presentations, emails)
- Package management for dependencies.
- Regression testing:
 - No formal testing environment.
- Release process and tracking? Informal
- Revision control - subversion
- Unit tests - N/A