EUROPEAN COMMISSION

Community research

is-enes
INFRASTRUCTURE FOR THE EUROPEAN NETWORK FOR EARTH SYSTEM MODELLING

IS-ENES2 DELIVERABLE (D-N°: 9.3)

Assessment report on Autosubmit, Cylc and ecFlow

{Original title: Assessment report on Autosubmit and Cylc}

File name: IS-ENES2_D9_*3*.pdf

Author(s): **Domingo Manubens-Gil, Javier Vegas-Regidor**

Reviewer(s): **Sébastien Denvil, Stéphane Senesi, Pavan Kumar Siligam**

Contributor(s): **David Matthews, Matthew Shin**

Reporting period: ***01/10/2014 – 31/03/2016***

Release date for review: **21/03/2016** Final date of issue: **31/03/2016**

Revision table			
Version	**Date**	**Name**	**Comments**
0.1	21/03/2016	Domingo Manubens-Gil	Original version for review
0.2	29/03/2016	Domingo Manubens-Gil	Including review comments from Pavan
0.3	30/03/2016	Domingo Manubens-Gil	Including review comments from Stéphane
0.4	30/03/2016	Domingo Manubens-Gil	Including review comments from Sébastien
1.0	31/03/2016	Domingo Manubens-Gil	Final version

Abstract

D-9.3, "Assessment report on Autosubmit, Cylc and ecFlow", is a deliverable due in month 36, ahead of the deliverable D-9.6 "Multi-model multi-member (M4) high resolution (HR) Earth System Model (ESM) ensemble performance analysis". In the context of work package 9, IC3/BSC in collaboration with the Met Office, have set up three tools to design workflows and monitor experiments: Autosubmit, Cylc and ecFlow. The three scheduling and submission systems have been tested and evaluated with regard to the suitability for M4 HR experiments. GloSea5 operational seasonal forecast system and EC-Earth3 decadal hindcast have been prototyped with several workflow configurations in order to compare the differences and assess the suitability of the three tools for M4 HR experiments.

Project co-funded by the European Commission's Seventh Framework Programme (FP7; 2007-2013) under the grant agreement n°312979		
Dissemination Level		
PU	Public	PU
PP	Restricted to other programme participants including the Commission Services	
RE	Restricted to a group specified by the partners of the **IS-ENES2** project	
CO	Confidential, only for partners of the **IS-ENES2** project	

CAPACITIES

Table of contents

1. Executive summary

The main objective of WP9/JRA1 is to define, set up, and run a multi-model multi-member high-resolution (M4 HR) earth system model (ESM) ensemble experiment.

Before adapting M4 HR workflow to the job submission tool under consideration, this deliverable (D9.3) aims at reporting on the development status of the suitability of submission tools for M4 HR runs and on their respective computational performance in an operational environment.

Given the strong dependence of job scheduling and submission systems on ESM and HPC particularities, three options are retained for assessment in this study: Autosubmit, Cylc and ecFlow. For multi-member experiments, Autosubmit has already been evaluated on two HPC systems: ECMWF IBM Power 7 and MareNostrum 3 (see M9.1 [6] and M9.2 [7]).

IC3/BSC in collaboration with the Met Office, have set up Autosubmit, Cylc and ecFlow. An environment with access to several remote HPC systems, with and without security constraints have been used for the evaluation. Thereafter, the support for remote platforms, workload managers and task communication methods, among others, has been assessed.

The three scheduling and submission systems have been tested and evaluated with regard to the suitability for M4 HR experiments. GloSea5 operational seasonal forecast system and EC-Earth3 decadal hindcast have been prototyped with several workflow configurations in order to compare the differences and assess the suitability of the three tools.

2. Background

2.1 Studied submission tools

Autosubmit is a solution created at IC3's Climate Forecasting Unit (CFU) to manage and run the research group's experiments. Lack of in house HPC facilities led to a software design with very minimal requirements on the HPC that will run the jobs. Autosubmit provides a simple workflow definition capacity that allows running weather, air quality and climate multi-member experiments in more than one supercomputing platform. Autosubmit is currently being developed at BSC Computational Earth Sciences group.

http://www.bsc.es/projects/earthscience/autosubmit/

The Cylc suite engine is a workflow engine and meta-scheduler for weather forecasting and climate modelling. It is designed to run operational suites with complex date-time cycling requirement. Cylc was created by Hilary Oliver at NIWA. Its core team now includes Hilary as well as members in the Modelling Infrastructure Support Systems Team at the Met Office. Cylc is used to run time critical operational weather forecasts at NIWA and Met Office, as well as for research. It is also installed and used by research partners of NIWA and Met Office, and beyond.

http://cylc.github.io/cylc/

ecFlow is a workflow package that enables users to run a large number of programs (with dependencies on each other and on time) in a controlled environment. It is used at ECMWF to manage around half their operational suites across a range of platforms.

https://software.ecmwf.int/wiki/display/ECFLOW/

Criteria	**Autosubmit**	**Cylc**	**ecFlow**
Seniority	2011	2010	2011
Original authors/sponsors	IC3, BSC	NIWA, Met Office	ECMWF
License	GNU GPL v3	GNU GPL v3	Apache License v2.0

3. Comparison

In this chapter we agreed a set of features which we intend to compare: portability (ease of installation), task communication (how the tool knows the state of tasks), support for remote platforms, support for different workload managers (SLURM, PBS, etc.), support for automated error recovery (e.g. on submission failure, task failure, etc.), support for date/time cycling, scalability (ability to cope with large, complicated workflows + ability to cope with large numbers of users), monitoring & intervention tools (ability to interact with & modify running suites), fault tolerance (ability to recover if server goes down whilst running a suite) and support for generated workflows (i.e. defined via some sort of programming language).

3.1 Portability

Autosubmit is a Python package available on the Python Package Index repository (PyPi), so it can be installed using the pip install instruction on a terminal where no administrator privileges are needed.

To start using Autosubmit, configure and install commands need to be run by following the Autosubmit user guide. It creates a self-contained SQLite database that allows registering experiments uniquely identified. After that, the database can be shared with other installations through NFS, for example.

No installation is needed on the machines that will run the jobs.

Cylc is an application implemented mainly in Python and Bash. Installation is as simple as downloading a release tarball from Github, and editing the environment to ensure that the "bin/" directory of the distribution is in "PATH". Cylc has a compulsory dependency on Pyro3 on hosts running suite daemons. Pyro will be included as part of Cylc's distribution in the next release, so there will no longer be a dependency.

On hosts running task jobs, a copy of Cylc should also be available, but it has no compulsory dependencies.

On hosts running suite daemons, optional dependencies are Jinja2 (if used with suites with Jinja2 in their configuration), Pygraphviz (if graphing and/or full validation of suite configurations are required), and PyGTK (if GUIs are required). All of these dependencies are readily available from PyPi and/or from standard repositories of popular Linux distributions.

Cylc has been installed on many sites around the world. For personal use, it can normally work without any global site configuration. For site installation, it is normally desirable to modify the global site configuration file to tailor for the site. The settings in the global configuration file are well documented in the Cylc user guide.

ecFlow is a C++ application and features a client-server model. Installation for server and clients is the same. ecFlow does not provide distribution packages; instead it is usually installed from the source code. Instructions are well documented and can be found on ecFlow

wiki page [4]. The required software dependencies need to be installed with administrator privileges, following the instructions (cmake, g++, Python, Xlib, X11, XMotif).

ecFlow functionality is provided by following executables and shared libraries:

- ecflow_client: This executable is a command line program: it is used for all communication with the server. It needs to be installed on the target platforms.
- ecflowview: This is a specialised GUI client that monitors and visualises a tree-like hierarchy corresponding to the tasks.
- ecflow_server: this executable is the server. It is responsible for scheduling the jobs and responding to the ecflow_client requests.
- ecflow.so, libboost_python.so: these shared libraries provide the Python API for creating the suite definition and communication with the server.

Submission of tasks to remote queueing systems from ecFlow is possible, although it is based on quite rudimentary features (see section 3.3). Site administrators are required to provide job submission scripts and install elements of ecFlow (mentioned above) to enable task communications.

3.2 Task communication

Autosubmit is built on top of Simple API for Grid Applications (SAGA) [10]. SAGA-Python is a light-weight Python package that implements the Open Grid Forum (OGF) GFD.90 SAGA [13] interface specification and provides the access layer for distributed computing infrastructure. Autosubmit uses this access layer to control the submission of available jobs when the dependencies are satisfied and to monitor the status of the active ones.

SAGA-Python provides several plugins (called adaptors) that interface with middleware that doesn't support remote submission. These plugins are used in conjunction with other type of adaptors that provide machine communication, e.g. tunnelling calls via SSH:

```
saga.job.Service('pbs+ssh://my.remote.cluster')
```

This functionality requires a working SSH set-up on both, the submit host (the machine that runs Autosubmit) and the machine that is specified as saga.job.Service (the machine that executes the jobs). In order to use plugins that allow SSH-tunneling (xyz_+ssh_://), it is necessary to set-up password-less SSH-keychain access to the remote hosts one wants to use.

SAGA performs different kinds of interactions with remote systems. Many of those systems are only accessible via shell-like tools, such as SSH, GSISSH, FTP, GSIFTP etc. 'Shell-like' means that those tools are mostly designed for interactive use: after connection setup they present a prompt and wait for commands on stdin, and then respond to those commands via stdout/stderr.

The PTY layer in SAGA [11] consists of several components which handle interaction with those tools: pty_process, pty_shell and pty_shell_factory. The pty_process is a fork/exec'ed

process which provides low level process management (is_alive, kill, wait, ...) and process I/O (read, write, find). The pty_shell_factory basically creates a suitable command line and hands it to pty_shell. The pty_shell applies various heuristics to ensure that a spawned shell is bootstrapping correctly, and to get the two-way communication channel initialized. Once a prompt is detected and a new prompt is set, the pty_shell performs reliably, fast and stably.

A job as returned by job.Service.create(jd) is in NEW state – it is not yet submitted to the job submission backend. Once it is submitted, via run(), it will enter the PENDING state, where it waits to get actually executed by the backend (e.g. waiting in a queue). Once the job is actually executed, it enters the RUNNING state – only in that state is the job actually consuming resources (CPU, memory, etc.). Jobs can leave the RUNNING state in three different ways: they finish successfully on their own (DONE), they finish unsuccessfully on their own, or get cancelled by the job management backend (FAILED), or they get actively cancelled by the user or the application (CANCELLED).

Autosubmit checks regularly the status of the jobs by reading the SAGA job state.

```
saga_status = self.service.get_job(jobid).state
```

Cylc can submit jobs on the suite host and/or on remote job hosts via SSH. A task submits a job when all prerequisites are satisfied. On job submission, Cylc generates a job script for each task, which it then submits to the relevant queueing system on the job host.

Cylc has two ways to track progress of submitted jobs. This can be configured per job host in the site/user global configuration. These methods can be used on different job hosts within the same suite if necessary.

1. Job-to-suite messaging.
    - Job-to-suite messaging via Pyro (default). This is the most direct and efficient communication method. The job script calls a Cylc command to send a message back to the suite via Pyro on given events, e.g. on job start, success and failure. (Custom events can also be defined with custom messages.) The job script also writes to a status file to ensure that the event is recorded in case of network outage.
    - Job-to-suite messaging via SSH+Pyro. This is similar to the above, but the job host will connect to the suite host via non-interactive SSH before connecting to the suite via Pyro. This method is useful if the job host does not have access to the Pyro network port.
2. Polling. This is the least efficient method for monitoring the progress of a job, but can be used on job hosts that cannot route back to the suite host via Pyro or SSH. Cylc polls for the progress of the job at regular intervals by inspecting the status file written by the job script and/or by querying the queueing system. Users can also manually tell the suite to poll its jobs at any time while the jobs are in the submitted or running state.

ecFlow submits jobs and receives acknowledgements from jobs when they change status and when they send events. It does this using child commands embedded in the ecFlow scripts.

The ecFlow script refers to an '.ecf' file. The script file is transformed into the job file by the job creation process. The creation is initiated by the ecflow_server during scheduling when a task (and all of its parent nodes) is free of its dependencies. The script must include calls to the init and complete child commands so that the ecflow_server is aware when the job starts (i.e. changes state to active) and finishes (i.e. changes state to complete).

An ecf script is converted to a job file that can be submitted by performing variable substitution on the ECF_JOB_CMD variable and invoking the command (typically):

```
/bin/sh %ECF_JOB% &
```

The running jobs will communicate back to the ecflow_server by calling child commands:

- ecflow_client –init Sets the task to the active status
- ecflow_client –event Raises an event
- ecflow_client –meter Change a meter
- ecflow_client –label Change a label
- ecflow_client –wait wait for an expression to evaluate.
- ecflow_client –abort Sets the task to the aborted status
- ecflow_client –complete Sets the task to the complete status

The ecflow_server is responsible for scheduling the jobs and responding to ecflow_client requests. ecFlow stores the relationship between tasks and is able to submit tasks depending on triggers. For any communication with the server, the client needs to know the machine where the server is running and the port on the server. Communication is based on TCP/IP. Multiple servers can be run on the same machine/host provided they are assigned a unique port number. The server records all requests in the log file. The server will periodically write out a check point file.

3.3 Support for remote platforms

Autosubmit is capable to run experiments on remote clusters or supercomputers and on any GNU/Linux or Unix host. The interaction with the machines is done through SAGA-Python adaptor, allowing the user to add adaptors if needed.

The Autosubmit default method for accessing remote platforms through SAGA requires a working interactive SSH set-up on both the machine that runs Autosubmit and the machine that executes the jobs. For transferring files it uses SFTP.

The SAGA adaptors mechanism can add support for other communication methods, such as ECaccess Tools [5]. The ECaccess Tools gives ECMWF users batch access to the ECMWF computing and archiving facilities for the management of files, file transfers and jobs. Access is available via the Internet as well as via RMDCN. A SAGA ECaccess adaptor has been developed for the latest version of Autosubmit.

Due to the fact that developers of Autosubmit had no in house HPC facilities, the software was designed to work with very minimal requirements on the HPCs that will run the jobs: only a bash console with the usual commands is required. To run Python or R jobs, no additional packages are needed.

Cylc can run jobs on remote Unix/Linux job hosts that are able to receive SSH (and SCP) connections. For fully automated job submission, non-interactive SSH is required. On the job host, Cylc requires bash, a small subset of GNU coreutils and Python (for running a small set of Cylc commands). Optionally, job hosts can be configured to take advantage of efficient task messaging via Pyro.

ecFlow can run tasks on remote systems**.** To start a job, the ecflow_server uses the content of the ECF_JOB_CMD variable. By modifying this variable, it is possible to control where and how a job file will run. Having a variable called HOST defined as the name of the host and assuming that all the files are visible on all the hosts, e.g. using NFS, a remote job can be submitted:

```
edit ECF_JOB_CMD "ssh %HOST% '%ECF_JOB% > %ECF_JOBOUT% 2>&1 &'"
```

When using SSH, requires ones public key to be available on the destination machine.

However, the communication back from running jobs on diverse supercomputing platforms to ecflow_server may not be possible due to restricted traffic and firewalls in the network. ecflow_client must have access to the ecflow_server network port.

Since ecflow_server cannot poll for job status (the alternate method implemented in the other two submission tools assessed in this report) support for certain remote platforms is not possible in ecFlow.

3.4 Support for different workload managers

Within **Autosubmit**, SAGA provides a homogeneous programming interface to the majority of production HPC queuing systems:

- Fork (run job as a background process)
- Condor and Condor-G
- LoadLeveler
- LSF
- PBS and Torque
- Sun/Oracle Grid Engine
- SLURM

All queuing system adaptors can also access clusters remotely by tunnelling commands through SSH or GSISSH.

Behind the API facade, SAGA implements flexible adaptor architecture. Adaptors are plugins that binds API calls to the respective queuing system. Most application developers use the

adaptors that are already part of SAGA but implementation of adaptors not yet supported is possible by following the 'Writing SAGA-Python Adaptors' guide [14].

Cylc supports a number of commonly used job submission methods:

- Background (run job as a background process with "nohup")
- at
- LoadLeveler
- LSF
- PBS and Torque
- MOAB
- Sun/Oracle Grid Engine
- SLURM

Users can also provide their own methods in case they need to interact with a scheduler that is not supported out of the box for Cylc. The process to implement and use a new submission method is well-documented on Cylc documentation.

ecFlow can submit tasks directly to the relevant queuing system on the target machine. ECMWF provides a submission script (ecf_submit) that allows submission to multiple systems and multiple queuing systems:

- LoadLeveler
- PBS
- Sun/Oracle Grid Engine
- SLURM

An example ecf_submit is included in the ecFlow release. The ECF_JOB_CMD can be defined as:

```
edit ECF_JOB_CMD "ecf_submit %USER% %HOST% %ECF_JOB% %ECF_JOBOUT%"
```

A generic script header is included alongside the ecf_submit script. It contains typical queuing commands, such as wall clock time and priority. The ecf_submit script can replace the generic queuing commands with the relevant commands for the host to which the task is submitted and submit the task the relevant way. For example, for a PBS system it replaces the QSUB commands with the equivalent PBS commands.

Similarly to running a task remotely, to kill a task remotely one need to either send a signal to the task or issue the relevant queueing system command. Latest releases of ecFlow include example scripts for including other information: ecf_kill to issue the correct command depending on the host, ecf_status to show status of tasks and ecf_url to open a web link for a task. ecFlow variables can be defined as:

```
edit ECF_KILL_CMD   'ecf_kill %USER% %HOST% %ECF_RID% %ECF_JOB%
edit ECF_STATUS_CMD 'ecf_status %USER% %HOST% %ECF_RID% %ECF_JOB%
```

3.5 Fault tolerance

Autosubmit is able to deal with faults at different levels. It creates a COMPLETED file to deal with inconsistencies when the queue scheduler does not respond properly. It also automatically saves the job list each time that it is updated. This way, if Autosubmit process is killed, it can restart the experiment at the same point with no data loss.

Autosubmit also has a mechanism that can be used in case of a critical failure that makes the job list file unreadable. In this case, the user can run the create command to recreate the job list at the initial state and then run the recovery command to look for the COMPLETED files and update the job's status accordingly.

In the case that the fault is due to the jobs templates, Autosubmit has two features that allow the user to easily continue with the experiment. The first one is that the scripts for the jobs are prepared immediately before sending them to the platform. This allows the users to modify the templates after the experiment has started. Autosubmit also reads the project parameters at this time to allow the users to change them in the middle of the experiment in case that the failure is due to a configuration error.

A Cylc suite dumps out state files and writes information to SQLite databases on state changes. If a suite is terminated due to the process being terminated, e.g. a power failure, the user can easily recover by restarting the suite when the machine is back up again. On restart, the suite will automatically poll all its submitted and running jobs for their latest states, by looking at their status files and by querying the queueing systems.

ecFlow could stop working for a number of reasons such as the server crashes, the computer ecFlow is running on crashes, etc. The ecFlow checkpoint file allows ecFlow to restart at the point of the last checkpoint before a failure. This gives reasonable tolerance against failures.

When the server starts, if the checkpoint file exists and is readable and is complete, ecFlow server recovers from that file. Once recovered the status of server may not exactly reflect the real status of the suite, it could be up to a few minutes old. Tasks that were running may have now completed so the task status should be checked for consistency.

The checkpoint files can be read by any ecFlow running on any operating system. There are two separate checkpoint files.

```
ECF_CHECK      ecf.check
ECF_CHECKOLD  ecf.check.b
```

When ecFlow needs to write a checkpoint file it first moves (renames) the previous file ECF_CHECK to ECF_CHECKOLD and then creates a new file with the name ECF_CHECK. This means that one should always have a file that is good. In the event of a crash, while writing ECF_CHECK, one can still recover from ECF_CHECKOLD (by copying its contents to ECF_CHECK), although that version is not quite as up to date. One can copy the checkpoint files between systems. Another ecFlow server can be started with the original

server's checkpoint file and take over from the original ecFlow server host in case of a catastrophic systems failure.

3.6 Support for automated error recovery

Autosubmit has a RETRIAL variable defined for each experiment. Additionally each job type can have a different RETRIAL number. If a job gets the FAILED status from the queue scheduler it automatically retries that number of times.

Sometimes the queue scheduler does not respond properly and it could easily lead to wrong status. For this, Autosubmit adds a final operation to the end of the job which creates a completed file. When Autosubmit gets an answer from the scheduler, it will put the job on a COMPLETED, FAILED or UNKNOWN state, it checks the existence of the completed file and updates the status if needed, i.e., if the scheduler returns an UNKNOWN status and Autosubmit finds the completed file, the final status of the job will be COMPLETED.

Autosubmit also has a recovery command that can be used when the experiment jobs list has not an accurate representation of the current state. In this case, this command can search for the completed files of all the jobs at every platform and update job status accordingly.

Cylc tasks can be configured to retry a number of times on runtime failure as well as on submission failure. An environment variable $CYLC_TASK_TRY_NUMBER increments from 1 on each successive try, and is passed to the task to allow different behaviour on the retry. When a task with configured retries fails, its Cylc task proxy goes into the retrying state until the next retry delay is up, then it resubmits. It only enters the FAILED state on a final failure.

Cylc also has the capability to add triggers on failure, which allows implementing recovery tasks. It also has support for suicide triggers that take tasks out of the suite. This can be used for automated failure recovery, defining a chain of failure recovery tasks that trigger if they’re needed but otherwise remove themselves from the suite.

ecFlow allows setting ECF_TRIES to a number greater than one in the definition file of a suite, the task will automatically rerun on an abort. Then the ecFlow variable ECF_TRYNO can be used to modify the behaviour of the task depending on the try number.

Additionally, ecFlow allows error checking and handling of zombies. A zombie is a running job that fails authentication when communicating with the ecflow_server. The default behaviour of ecFlow server is to block the job. The child command continues attempting to contact the ecFlow server. This is done during 24 hours. This duration is configurable on ecflow_client through ECF_TIMEOUT variable. The jobs can also be configured, so that if the server denies the communication, then the child command can be set to fail immediately (ECF_DENIED). ECF_TIMEOUT is the maximum time in seconds for the client to try to deliver a message. ECF_DENIED, if is set to 1 and ecFlow denies access, the client will exit with failure.

3.7 Support for date/time cycling

Autosubmit does not have built-in triggers that run at a given real date/time as it was no designed for running operational suites.

The extent of an experiment can be defined in three aggregations or families: number of start-dates, number of members within a start date and number of chunks within a member. Chunk length can be defined in years, months, days or hours. The experiment can cycle as many chunks, members and start-dates as needed. Autosubmit supports standard and no-leap calendar modes.

Cylc can work in two modes:

1. Integer cycling. Each cycle point corresponds to an integer in a sequence. Non-cycling suite is a special case, where we have a single cycle point == 1.
2. Date-time cycling: Each cycle point corresponds to a given date-time. Cylc supports various calendar modes including Gregorian, 360day, 365day and 366day. Date time cycling syntax can be expressed using the full grammar of ISO8601 notations for date-time, duration and recurrence.

The cycling period is not fixed: one can have jobs cycling at different frequencies without any problem. Cylc has also support for clock-triggered jobs, which is important for operational suites.

The user can also use Jinja2 script language on suite definition, allowing the creation of cycling-like suites even when using Cylc in non-cycling mode.

ecFlow has two ways for defining workflows: a text-based custom format and a Python API. In this document we will only refer to the Python API as it's the most powerful method.

ecFlow does not have an internal cycle, but allows repeating the same task or family several times, looping on a specific value. It can iterate over sequences of strings, integers or dates (standard calendar). It has the possibility to define clock-triggered task, allowing ecFlow to be used on operational suites.

3.8 Monitoring and intervention tools

Autosubmit has a very basic monitoring tool that shows the experiment graph. Autosubmit monitor command creates a directed acyclic graph with jobs as nodes and its dependencies as arrows. Each node is shown with the job identifier and it is coloured with the current status (e.g. WAITING, RUNNING, COMPLETED, FAILED, etc.).

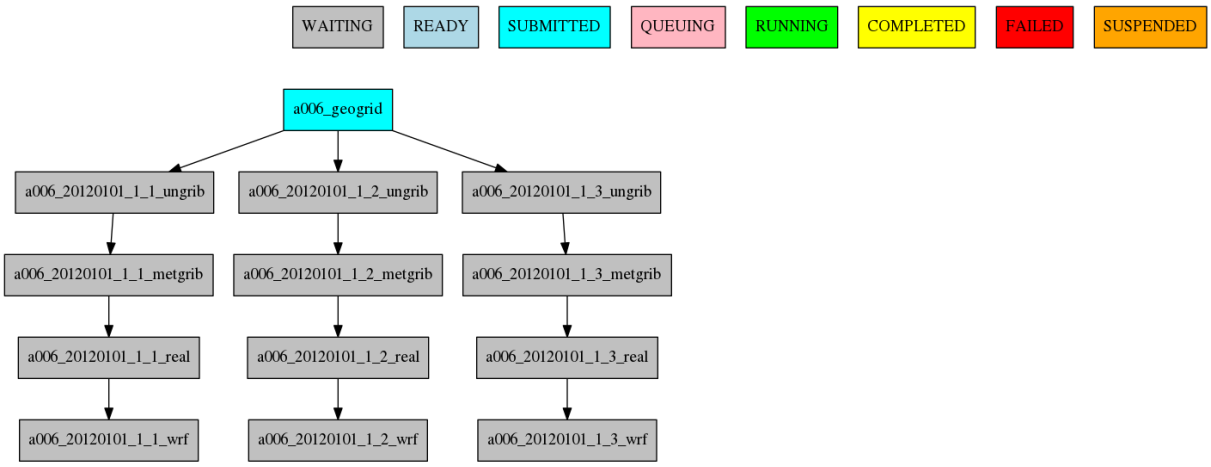

Figure 1: Autosubmit monitor screenshot

A typical user monitors his/her Autosubmit experiment by launching the monitoring tool regularly. Autosubmit has the capability to export the plot on vectorial format (SVG). This allows zooming in and out larger plots without losing clarity, which is a problem with the default pdf format. However, monitoring large experiments with only static images can be a burden.

Autosubmit allows the user to manually set the status of jobs using the setstatus command. This command is normally used to suspend some parts of the experiment while keeping others running or to relaunch failed jobs that have reached maximum retrials.

Autosubmit also collects statistics during experiment's execution (number of retrials, queueing and execution time, etc.) and allows the user to generate plots to monitor the resources consumption.

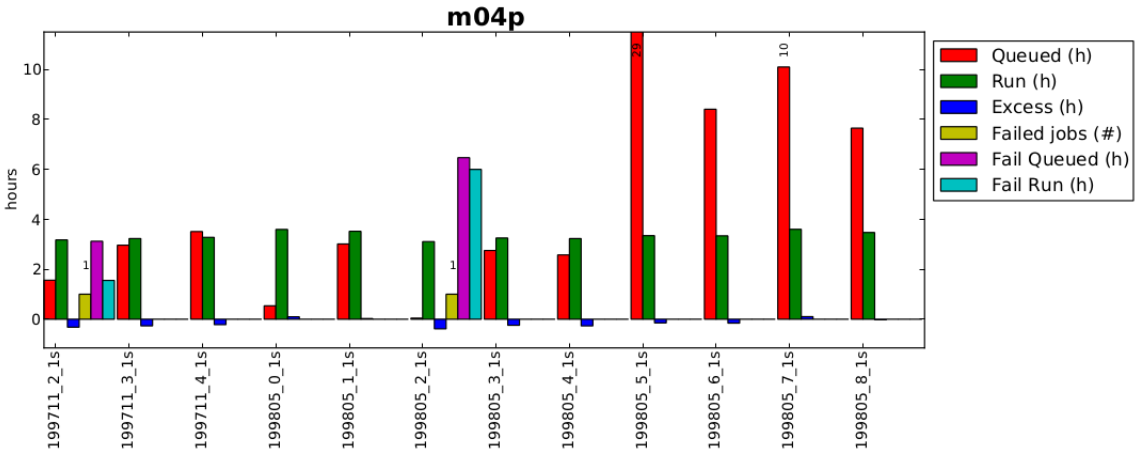

Figure 2: Detail of a statistics plot from Autosubmit

Cylc has many facilities to interact with a running suite, including the following:

- Hold/release whole suite and/or individual tasks.
- Poll/kill any submitted or running tasks.
- Insert/remove tasks at a range of cycles.
- Manually reset state of tasks.
- Trigger (manually submit) tasks.
    - Edit the task's job file, and trigger it.
- Broadcast (i.e. override) configuration settings to a running suite to tasks matching a cycle/name pattern.
- Reload the suite definition to a running suite.
- Restart a suite.
- Tail-follow STDOUT/STDERR of job logs while the job is running.
- View dependency graph and/or satisfied/unsatisfied task prerequisites.
- Scan and display states of all running suites.

All these can be controlled using the command line interface (CLI). Most functionality is also available via graphical user interfaces (GUI). Cylc has two GUI applications used to monitor and control the runs: a very minimal interface that shows the basic state of all the Cylc suites within a system (gscan) and a more detailed interface (gcylc) to monitor and control one suite at a time.

gscan (Figure 3) is the most convenient interface to keep track of the status of all existing simulations. It provides an interface that shows all the registered suites on the system along with a summary state. This way the user can detect easily if a suite has failed jobs or has stopped running and launch gcylc to act accordingly.

gcylc can be used to monitor and control one suite at a time. It has different views: a graph view that shows tasks and dependencies, a dot view that shows a summary state for the tasks (Figure 4) and a text-tree view that shows more details than the other two. In all these views, the user can control if all tasks are shown or if they are grouped by families (a family on Cylc is a group of tasks that derive from a common ancestor).

This GUI can also be used to start, pause and stop suites; to change task’s status, to relaunch failed tasks, to see logs from executed tasks and many other control functionalities. The interface is intuitive and easy to use.

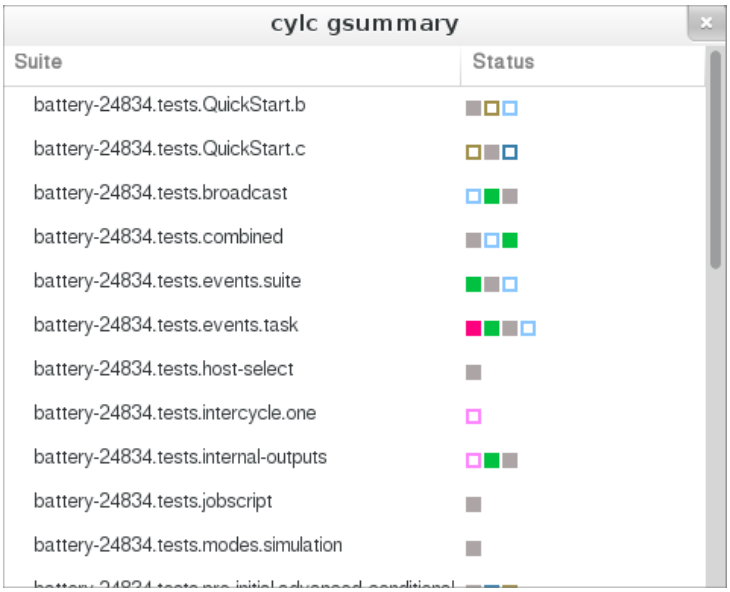

Figure 3: Cylc gsummary screenshot

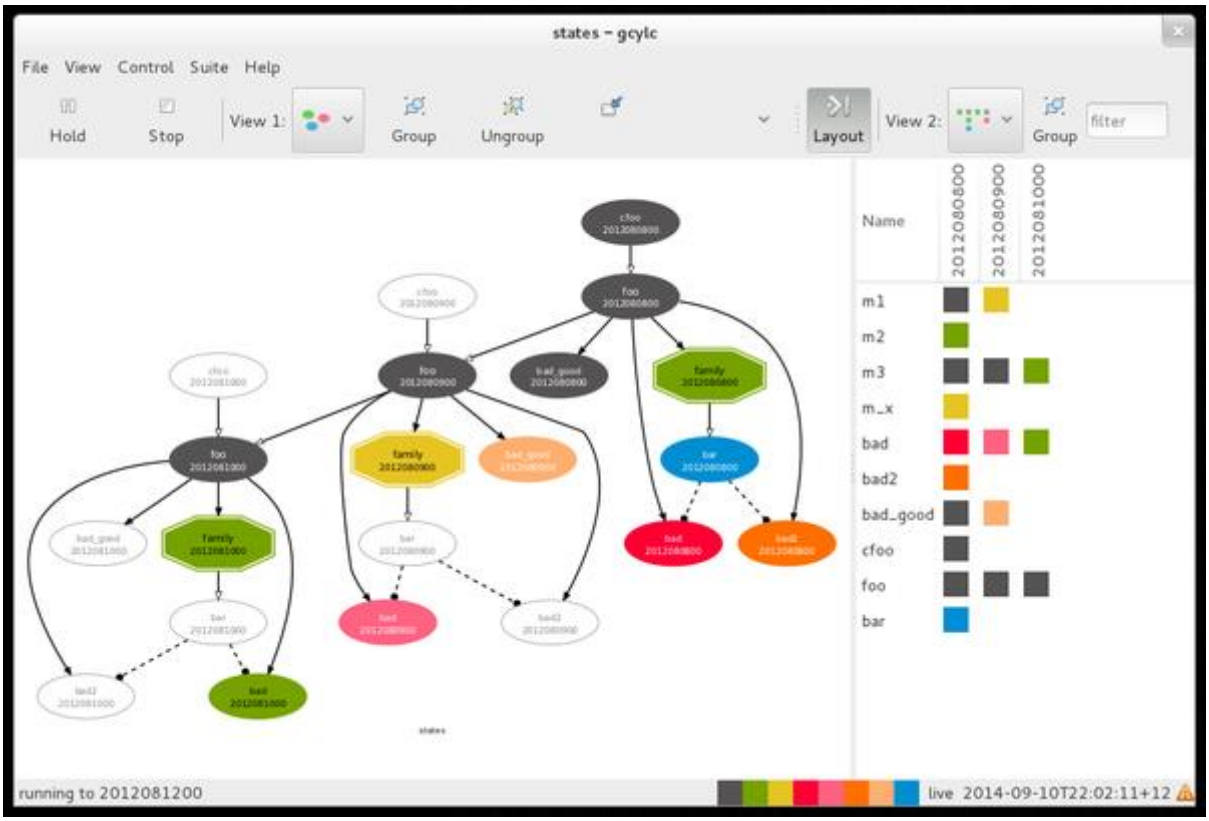

Figure 4: gcylc graph and dots view screenshot

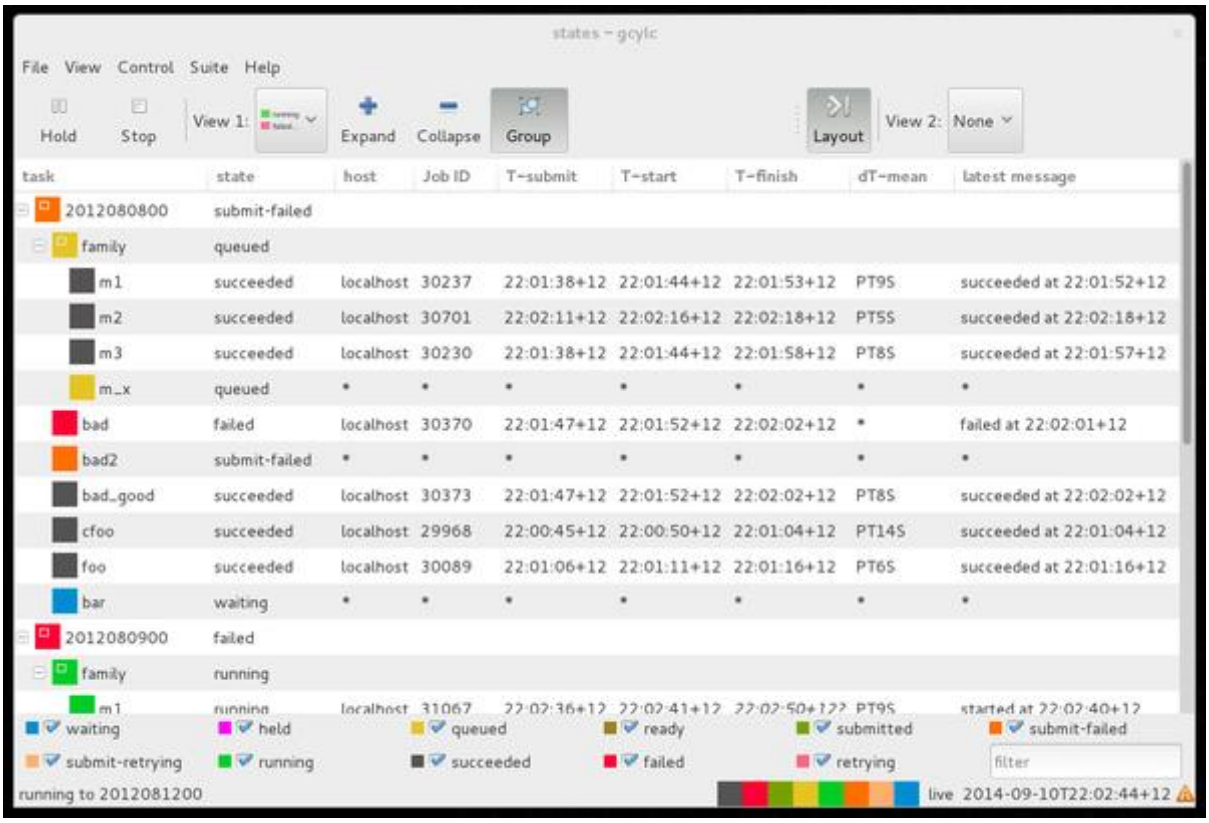

Figure 5: gcylc text view screenshot

ecFlow offers a Command Line Interface (CLI) and a client Graphical User Interface (GUI) (ecflowview) that can be used to monitor and control suites. ecflowview can control suites running on different ecFlow servers. It shows all the registered suites collapsed at start-up and the user can expand them as needed. It uses a tree-like display where the root nodes correspond to the suite, the intermediate to families (a group of tasks or other families that share parts of the configuration) and leafs correspond to the tasks.

The user can use the tool to change task status, launch, pause or stop suites, relaunch failed tasks, see logs and many other control functionalities. The interface is intuitive and easy to use.

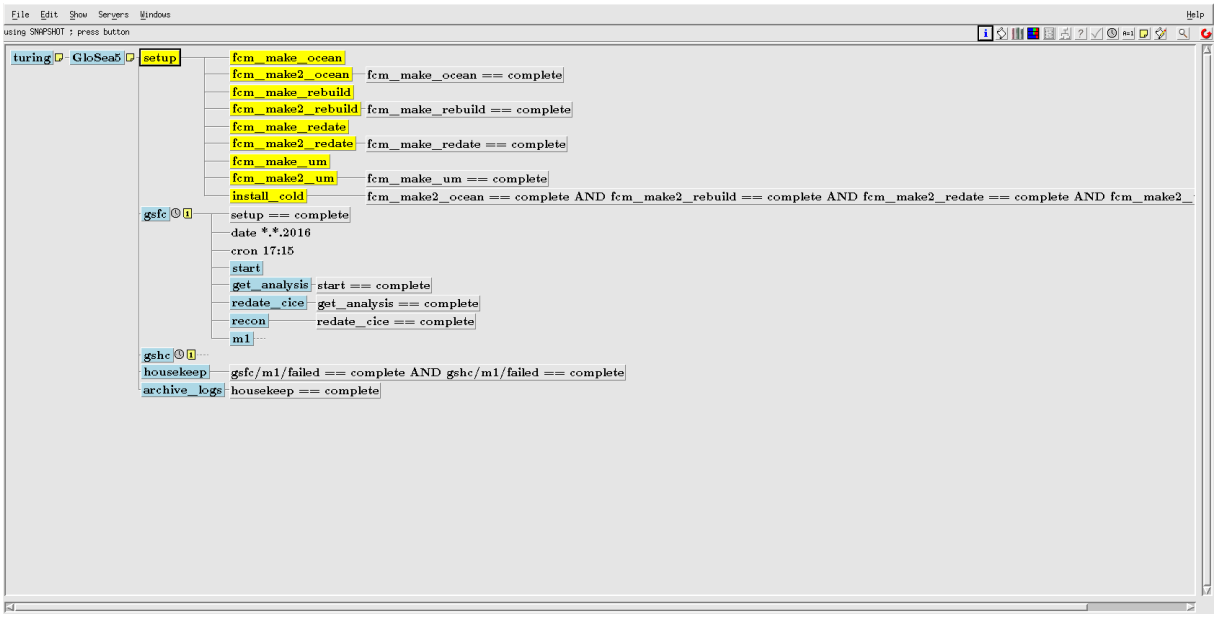

Figure 6: ecFlow Graphical User Interface (ecflowview)

3.9 Support for generated workflows

Autosubmit experiment's workflow is defined in INI style configuration files. Very simple workflows can be defined as sequences of jobs with one triggering at the end of the previous one. References are evaluated within a job-defined DEPENDENCIES attribute. It is important to note that DEPENDENCIES on Autosubmit always refer to jobs that must be finished before launching the job that has the DEPENDENCIES attribute. Further, a job defined RUNNING attribute is used to set the level where job runs. It has four possible values: once, date, member and chunk corresponding to running once, once per startdate, once per member or once per chunk respectively. Thereafter, Autosubmit can manage dependencies between jobs that are part of different chunks, members or startdates. The higher level job will wait for ALL the lower level jobs to be finished.

Cylc suites are defined in INI style configuration files. For complex workflow, Cylc supports the use of the Jinja2 template language to generate suite configuration files with loops, conditional statements, data structure, etc.

ecFlow suites are typically written in Python, so complex workflow can be defined using normal Python. This method has increased functionality over the text based format. The Python API allows complete specification of the suite definition, including trigger and time dependencies. Since the full power of Python is available to specify the suite definition, there is considerable flexibility. The API is documented using the Python __doc__ facility.

3.10 Scalability

Autosubmit was designed with multi-startdate multi-member experiments in mind. It has a built-in option to avoid overflowing platforms by limiting the number of jobs queueing and running at a time at any given platform. These limits are set at 3 and 6 jobs by default respectively, but can be tweaked by the user.

Autosubmit is used to run large experiments of a total of around 1000 jobs. The consumption of resources of the Python-based command line monitor and control tool is in the order of a few MB per experiment. The SAGA performance is evaluated in this study [12]. The saturation of the pty_shell and pty_process infrastructure (mentioned in the Task communication section) for a remote backend is reached above 1000 jobs per second. In the study, three options for further scaling are explored: (a) concurrent job service instances, (b) asynchronous operations, and (c) bulk operations. Normal Autosubmit usage within the limits mentioned before, does not lead to reach such a situation. However it would be worth implementing bulk operations in Autosubmit to reduce roundtrip overhead.

Cylc is used to run large (>1000 tasks per cycle point) time critical operational weather forecast suites at Met Office, so performance and scalability are key design elements.

It has efficient logic for job submission:

- Submissions of jobs are typically grouped together by job hosts.
    - Similar logic for job monitoring commands, e.g. poll and kill.
- Process pool limits the number of child processes the suite uses on its own host.

To avoid overflowing a job host, users can:

- Design the suite with explicit dependencies, including inter-cycle dependencies.
- Set the cycle runahead limit.
- Define internal queues to limit the number of jobs that can be submitted for any groups of tasks.

The GUI performance could be affected when having a high number of active cycle points. A limit on active cycle point could be a problem when used on an experiment with multiple startdates, although it can be overcome by using a sub-suite for each startdate.

ecFlow provide methods to check the status of an ecflow_server. Invoking “ecflow_client –stats” will display some standard information regarding the ecflow_server including the version number, node information, status, security information, usage, load, setup and up time. The load on the ecflow_server can be checked invoking “ecflow_client --server_load”.

The fact that ecflow_server was built from the ground up in C++, and that provides inter-server cooperation leads to good scalability. It is even possible to maintain work load during server and network outages and to share load between ecflow_server(s).

4. Complex workflow evaluation

In this chapter we analyse two real-life workflows:

- The GloSea5 operational suite.
- A hindcast EC-Earth experiment.

We implemented a simplified version of those two workflows with Autosubmit, Cylc and ecFlow to exemplify real-life applications. The simplification only affects job scripting and parameters definition.

4.1 Operational seasonal forecasting system GloSea5

a) Autosubmit

Autosubmit cannot reproduce the workflow used on GloSea5 due to two missing features. The first one is that Autosubmit does not have real time dependencies, as it is not designed to support operational runs. It also does not have support to trigger tasks on failed jobs, so the model_failed and housekeep jobs that the original workflow triggers on failure cannot be added.

The configuration of the *experiment* section on the *expdef.conf* file that defines the workflow can be seen in APPENDIX A - GloSea5 workflow code in Autosubmit.

The *jobs.conf* file for this workflow is available in APPENDIX A - GloSea5 workflow code in Autosubmit section Jobs (options for wallclock, processors, queues and other machine-related configurations have been removed).

Figure 7: Example of GloSea5 workflow generated with Autosubmit

b) Cylc

The GloSea5 operational suite is managed by Cylc using Rose [9]. Since it is a fully operational suite, the original suite definition file is quite complex. Hence, in this document, we present a simplified version of the suite definition file: APPENDIX B - GloSea5 workflow code in Cylc. It only includes the graph definition for the suite, the most important part. Thereafter the comparison of different methods to generate complex workflows is easier to understand.

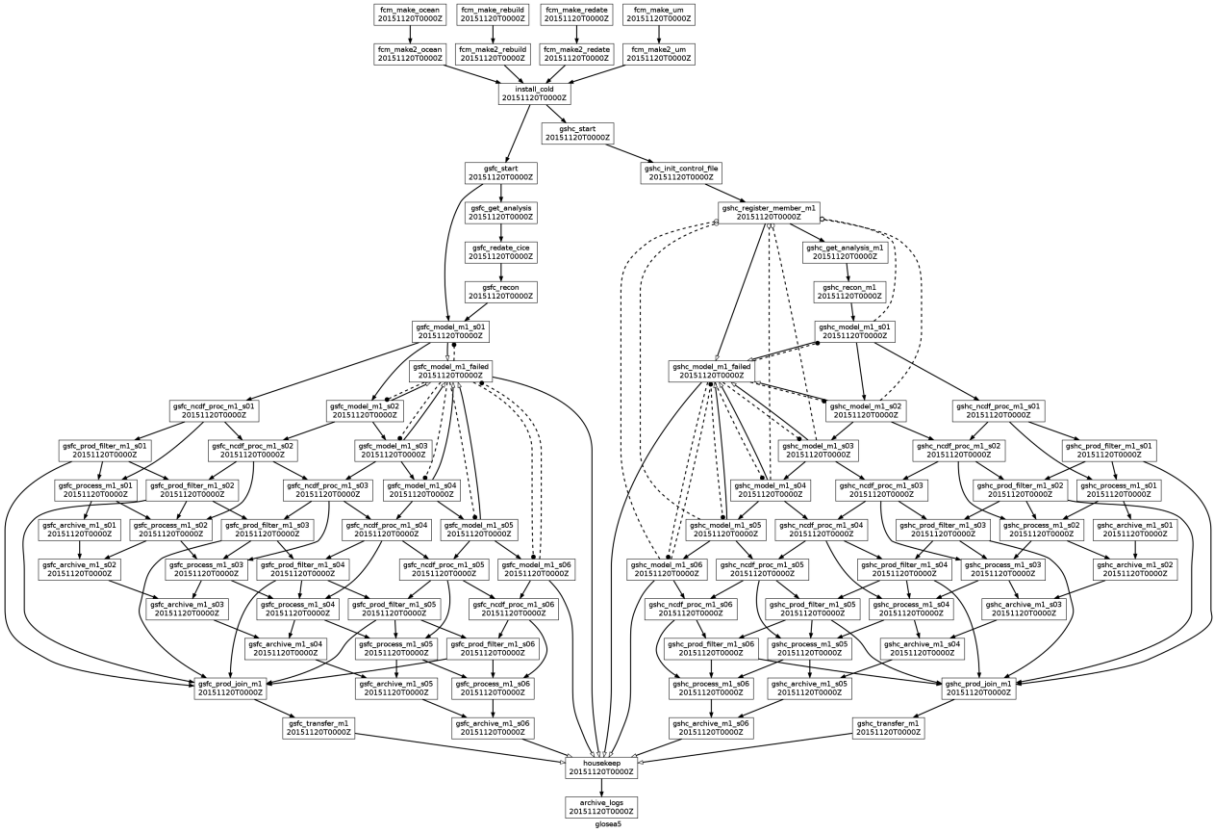

Figure 8: Example of GloSea5 workflow generated with Cylc

c) ecFlow

To define the GloSea5 workflow in ecFlow we have used the Python API. In this case we have been able to reproduce the behavior of the original suite created using Cylc. The Python code used to generate the workflow can be seen in APPENDIX C - GloSea5 workflow code in ecFlow.

ecflowview uses a tree view that can be collapsed to any level. Figure 9 shows the suite collapsed to the first level and successive pictures show the families expanded one at each time (see Figure 10 - Figure 11 - Figure 12).

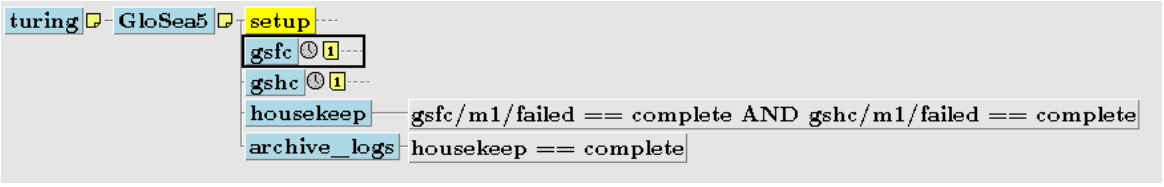

Figure 9: GloSea5 ecFlow suite with all families collapsed

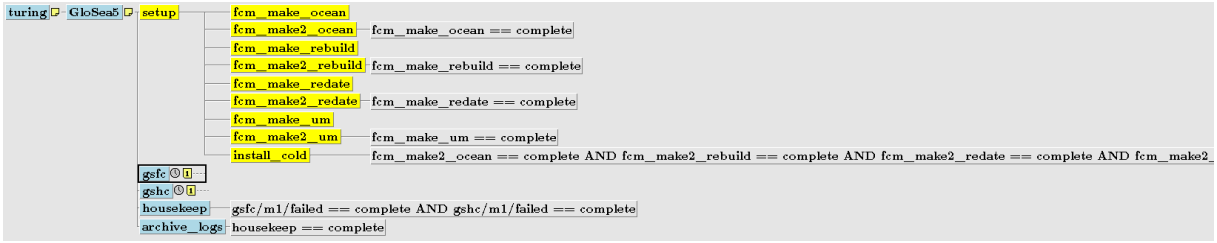

Figure 10: GloSea5 ecFlow suite with the setup family expanded

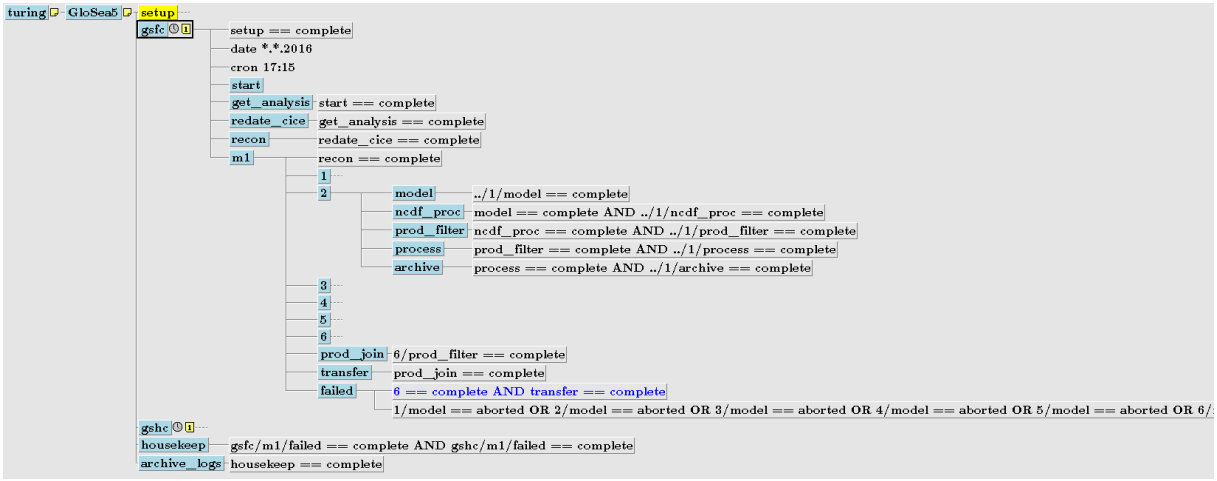

Figure 11: GloSea5 ecFlow suite with the gsfc family expanded

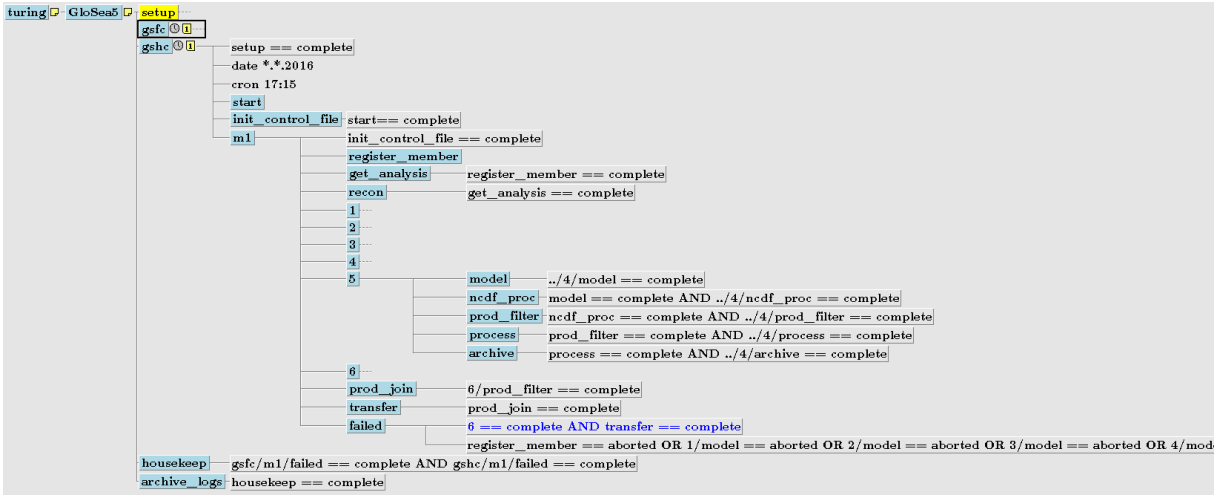

Figure 12: GloSea5 ecFlow suite with the gshc family expanded

4.2 Decadal hindcast with EC-Earth

A decadal hindcast experiment with EC-Earth usually consists of several model runs for different startdates and members. The workflow that will be used as a model is the one used at BSC-Earth sciences department, which has seven types of jobs.

As the complexity of this workflow for the managers is due to the high number of model runs required and the long time that this models need to run, we use a configuration with three startdates of five members each that will run for ten years in chunks of three months.

a) Autosubmit

In the case of the hindcast workflow, the *experiment* section on the *expdef.conf* file is configured below in APPENDIX D - EC-Earth workflow code in Autosubmit.

The *jobs.conf* file for this workflow can be seen in APPENDIX D - EC-Earth workflow code in Autosubmit, section Jobs.

Figure 13: Example of EC-Earth experiment generated with Autosubmit

b) Cylc

To define a hindcast experiment with Cylc we choose to use a multiple suite approach. In this case, the experiment has two suite definitions available in the APPENDIX. Note that these are simplified versions with a lot of parameters definition removed for clarity.

The main suite definition is shown in Control suite section. The control suite contains common tasks relative to model deployment and data transfer and also contains a task that will register, run and unregister a suite for each member that will run the proper simulation.

The simulation suite definition is shown in Sub-suites section. This definition is quite simple, and keeps the simulation of each member separated from the main ensemble workflow.

c) ecFlow

To define a hindcast experiment with ecFlow we choose to create a family composed by families for each member. Each member family is also composed by chunk families. This kind of definition makes the experiment easier to monitor using the tree view that ecflowview provides.

The use of Python functions while keeping as much as possible the dependencies relative to families, allows this workflow to be easily extended with new tasks at any point. The Python code used to create the workflow is available in the APPENDIX F - EC-Earth workflow code in ecFlow.

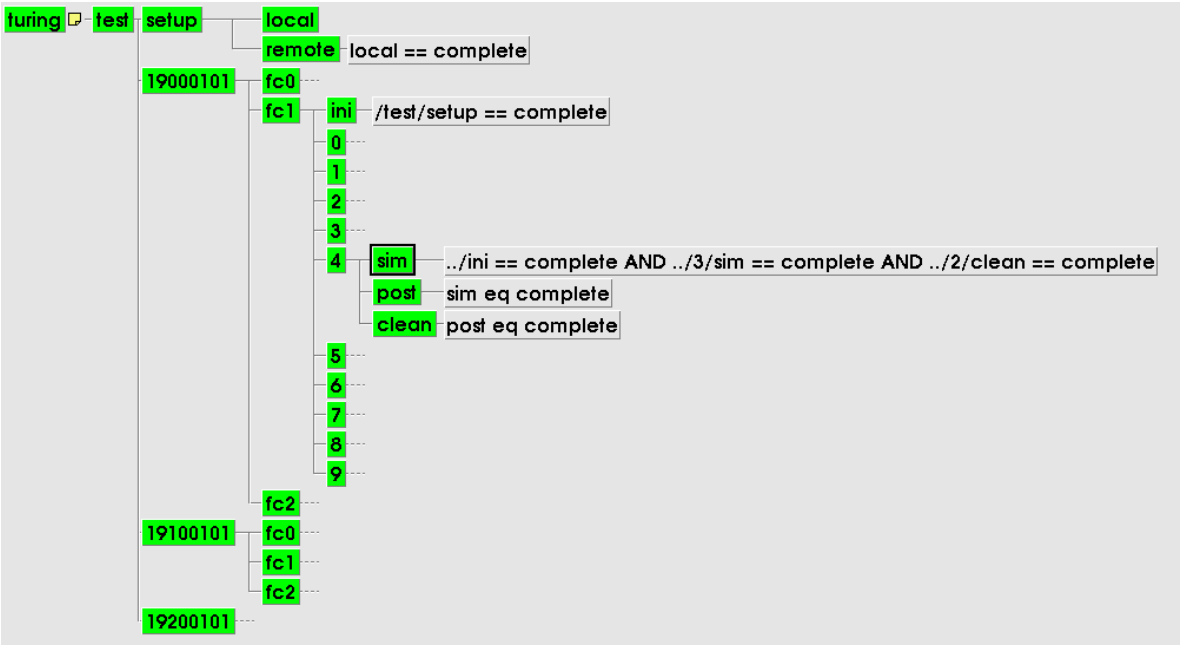

Figure 14: EC-Earth ecFlow suite

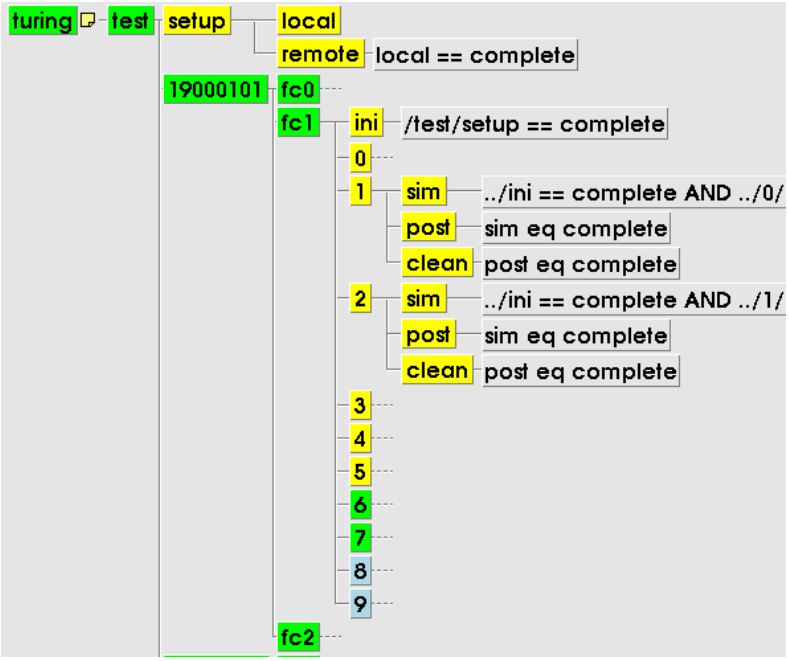

Figure 15: EC-Earth ecFlow suite with some chunk families expanded

4.3 Synthesis

The first use case evaluated, the operational seasonal forecasting system GloSea5, has been prototyped with Autosubmit, Cylc and ecFlow. The original version created with Cylc and Rose includes the full potential of an operational suite. The ecFlow equivalent includes all the potential, similarly. However, the Autosubmit equivalent has been simplified since launching triggers on failed tasks and defining real time dependencies is not supported.

The second use case evaluated, a decadal hindcast with EC-Earth has been successfully prototyped with Autosubmit, Cylc and ecFlow.

5. Perspectives

5.1 Assessment report relevance

The present assessment report and in particular the case evaluated in the previous chapter Operational seasonal forecasting system GloSea5, is relevant to understand how a multi-model multi-member ensemble experiment can be defined. The assessment demonstrates that Autosubmit, Cylc and/or ecFlow are suitable options to define, set-up and run such an experiment.

5.2 M4 HR ESM ensemble performance analysis

Contributing groups to WP9/JRA1 have been testing and evaluating common multi-member HR simulations, running an ensemble of HR ESM simulations (more than five members), in parallel on a given machine. A new set of computational performance metrics for ESMs, and the results of an initial analysis of the participating ESMs using these metrics are reported in D9.1 [8].

The final deliverable in the context of WP9/JRA1 is testing and evaluating a Multi-model multi-member (M4) high resolution (HR) Earth System Model (ESM) ensemble on a single HPC system: several groups will bring together three HR ESMs, with a minimum of 10 ensemble members per ESM, to be run in parallel as a single M4 HR ensemble, on a single system through a single submission step. Based on the computational costs of this initial test set, this task may be extended to multiple start-dates. The computational performance will be analysed (utilizing improved methods and/or tools described in D9.1 [8]) and D9.6 report will be written: “Multi-model multi-member high resolution Earth System Model ensemble performance analysis”.

Helped by BSC, 4 groups (CERFACS, CMCC, Met.no, SMHI) decided to join their efforts, developing an integrated multi model, based on ocean-atmosphere (CAM-NEMO, ARPEGE-NEMO or IFS-NEMO) or atmosphere only (CAM) models. To address it by means of a demonstrator an Autosubmit demonstrator is being designed that could be easily ported to Cylc and/or ecFlow. The demonstrator aims at showing that it is technically feasible to run HR models side-by-side on a machine. Technically speaking, the 3x2+1 executables are launched together, in the same MPMD MPI command. In addition to the MPI parallelism of each component, a “model” level parallelism increases the number of computing resources used at the same time.

6. References

1. Autosubmit documentation, [online], (accessed March 2016), http://www.bsc.es/projects/earthscience/autosubmit/
2. Autosubmit and EC-Earth Configuration Management, [online], (accessed March 2016), https://is.enes.org/documents/na3-documents/cm-documents/autosubmit-and-ec-earth-configuration-management/view
3. Cylc documentation, [online], (accessed March 2016), http://cylc.github.io/cylc/
4. ecFlow documentation, [online], (accessed March 2016), https://software.ecmwf.int/wiki/display/ECFLOW/
5. ECaccess documentation, [online], (accessed March 2016), https://software.ecmwf.int/wiki/display/ECAC/ecaccess
6. IS-ENES2_MS9.1_Multi-member HR prediction experiment using Autosubmit on Tier-1, [online], (accessed March 2016), https://is.enes.org/documents/milestones/is-enes2_ms9-1_multi-member-hr-prediction-experiment-using-autosubmit-on-tier-1/view
7. IS-ENES2_MS9.2_Further developments of Autosubmit, [online], (accessed March 2016), https://is.enes.org/documents/milestones/is-enes2_ms9-2_further-developments-of-autosubmit/view
8. IS-ENES2_D9.1_HR ESM Initial performance analysis_Submitted Version, [online], (accessed March 2016), https://is.enes.org/documents/deliverables/is-enes2_d9-1_hr-esm-initial-performance-analysis/view
9. Rose documentation, [online], (accessed March 2016), https://github.com/metomi/rose/
10. Merzky, Andre; Weidner, Ole; Jha, Shantenu. SAGA: A standardized access layer to heterogeneous Distributed Computing Infrastructure, SoftwareX, 1:3-8, 2015.
11. SAGA wiki - PTY Layer, [online], (accessed March 2016), https://github.com/radical-cybertools/saga-python/wiki/PTY-Layer
12. SAGA wiki - Performance, [online], (accessed March 2016), https://github.com/radical-cybertools/saga-python/wiki/Performance-of-saga-python
13. SAGA GFD.90, [online], (accessed March 2016), https://www.ogf.org/documents/GFD.90.pdf
14. SAGA developer documentation - Writing SAGA-Python Adaptors, [online], (accessed March 2016), http://saga-python.readthedocs.org/en/latest/developers/adaptorwriting.html

7. APPENDIX

7.1 APPENDIX A - GloSea5 workflow code in Autosubmit

a) Experiment

```
[experiment]
DATELIST = 20000101
MEMBERS = fc0
CHUNKSIZEUNIT = month
CHUNKSIZE = 1
NUMCHUNKS = 6
CALENDAR = standard
```

b) Jobs

```
[fcm_make_ocean]
FILE = fcm_make_ocean.sh

[fcm_make2_ocean]
FILE = fcm_make2_ocean.sh
DEPENDENCIES = fcm_make_ocean

[fcm_make_rebuild]
FILE = fcm_make_rebuild.sh

[fcm_make2_rebuild]
FILE = fcm_make2_rebuild.sh
DEPENDENCIES = fcm_make_rebuild

[fcm_make_redate]
FILE = fcm_make_redate.sh

[fcm_make2_redate]
FILE = fcm_make2_redate.sh
DEPENDENCIES = fcm_make_redate

[fcm_make_um]
FILE = fcm_make_um.sh

[fcm_make2_um]
FILE = fcm_make2_um.sh
DEPENDENCIES = fcm_make_um

[install_cold]
FILE = fcm_make_ocean.sh
DEPENDENCIES = fcm_make2_ocean fcm_make2_rebuild fcm_make2_redate
fcm_make2_um

[gsfc_start]
FILE = gsfc_start.sh
DEPENDENCIES = install_cold

[gsfc_get_analysis]
FILE = gsfc_get_analysis.sh
DEPENDENCIES = gsfc_start
```

```
[gsfc_redate_cice]
FILE = gsfc_redate_cice.sh
DEPENDENCIES = gsfc_get_analysis

[gsfc_recon]
FILE = gsfc_recon.sh
DEPENDENCIES = gsfc_redate_cice

[gsfc_model]
FILE = gsfc_model.sh
RUNNING = chunk
DEPENDENCIES = gsfc_recon gsfc_model-1

[gsfc_ncdf_proc]
FILE = gsfc_ncdf_proc.sh
RUNNING = chunk
DEPENDENCIES = gsfc_model gsfc_ncdf_proc-1

[gsfc_prod_filter]
FILE = gsfc_prod_filter.sh
RUNNING = chunk
DEPENDENCIES = gsfc_ncdf_proc gsfc_prod_filter-1

[gsfc_process]
FILE = gsfc_process.sh
RUNNING = chunk
DEPENDENCIES = gsfc_prod_filter gsfc_process-1

[gsfc_archive]
FILE = gsfc_archive.sh
RUNNING = chunk
DEPENDENCIES = gsfc_process gsfc_archive-1

[gsfc_prod_join]
FILE = gsfc_prod_join.sh
RUNNING = member
DEPENDENCIES = gsfc_prod_filter

[gsfc_transfer]
FILE = gsfc_transfer.sh
RUNNING = member
DEPENDENCIES = gsfc_prod_join

[gshc_start]
FILE = gshc_start.sh
DEPENDENCIES = install_cold

[gshc_init_control_file]
FILE = gshc_init_control_file.sh
DEPENDENCIES = gshc_start

[gshc_register_member]
FILE = gshc_redate_cice.sh
DEPENDENCIES = gshc_init_control_file
RUNNING = member
```

```
[gshc_get_analysis]
FILE = gshc_get_analysis.sh
DEPENDENCIES = gshc_register_member
RUNNING = member

[gshc_recon]
FILE = gshc_recon.sh
DEPENDENCIES = gshc_get_analysis
RUNNING = member

[gshc_model]
FILE = gshc_model.sh
RUNNING = chunk
DEPENDENCIES = gshc_recon gshc_model-1

[gshc_ncdf_proc]
FILE = gshc_ncdf_proc.sh
RUNNING = chunk
DEPENDENCIES = gshc_model gshc_ncdf_proc-1

[gshc_prod_filter]
FILE = gshc_prod_filter.sh
RUNNING = chunk
DEPENDENCIES = gshc_ncdf_proc gshc_prod_filter-1

[gshc_process]
FILE = gshc_process.sh
RUNNING = chunk
DEPENDENCIES = gshc_prod_filter gshc_process-1

[gshc_archive]
FILE = gshc_archive.sh
RUNNING = chunk
DEPENDENCIES = gshc_process gshc_archive-1

[gshc_prod_join]
FILE = gshc_prod_join.sh
RUNNING = member
DEPENDENCIES = gshc_prod_filter

[gshc_transfer]
FILE = gshc_transfer.sh
RUNNING = member
DEPENDENCIES = gshc_prod_join

[archive_logs]
FILE = archive_logs.sh
DEPENDENCIES = gsfc_archive gsfc_transfer gshc_archive gshc_transfer
```

7.2 APPENDIX B - GloSea5 workflow code in Cylc

```
#!jinja2
{% set START_CYCLE="20151120T00" %}
{% set N_GSHC_MEMBERS=1 %}
{% set N_GSHC_STEPS=6 %}
{% set N_GSFC_MEMBERS=1 %}
{% set N_GSFC_STEPS=6 %}
```

```
title = "GloSea Suite"
description="Global monthly forecast, seasonal forecast and seasonal
hindcast suite"

[cylc]
    UTC mode = True

[scheduling]
    initial cycle point = {{ START_CYCLE }}
    max active cycle points = 1
    [[special tasks]]
        sequential = gshc_start, gsfc_start, archive_logs, housekeep
        clock-triggered = gshc_start(PT12H15M),gsfc_start(PT12H15M)
    [[dependencies]]
        [[[ R1 ]]]
            graph = """
                fcm_make_ocean   => fcm_make2_ocean
                fcm_make_um      => fcm_make2_um
                fcm_make_rebuild => fcm_make2_rebuild
                fcm_make_redate => fcm_make2_redate
                fcm_make2_ocean & fcm_make2_um & fcm_make2_rebuild &
fcm_make2_redate => install_cold

                install_cold => gshc_start
                install_cold => gsfc_start
                """

        [[[ T00 ]]]
            graph = """
        gshc_start => gshc_init_control_file => REGISTER_MEMBERS
        {% for MEMBER in range( 1, N_GSHC_MEMBERS + 1 ) %}
            {% if MEMBER > 1 %}
            gshc_register_member_m{{ MEMBER - 1 }}:finish => \
            {% endif %}
            gshc_register_member_m{{ MEMBER }} => gshc_get_analysis_m{{
MEMBER }} => gshc_recon_m{{ MEMBER }} => \
                        gshc_model_m{{ MEMBER }}_s01 => gshc_ncdf_proc_m{{
MEMBER }}_s01 => gshc_process_m{{ MEMBER }}_s01 => \
                        gshc_archive_m{{ MEMBER }}_s01

            gshc_ncdf_proc_m{{ MEMBER }}_s01 => gshc_prod_filter_m{{ MEMBER
}}_s01 => gshc_process_m{{ MEMBER }}_s01
            gshc_prod_filter_m{{ MEMBER }}_s01 => gshc_prod_join_m{{ MEMBER
}}

            {% for STEP in range( 2, N_GSHC_STEPS + 1 ) %}
                {% set MEM_PREV_STEP = "m%d_s%02d" % (MEMBER, STEP-1) %}
                {% set MEM_STEP = "m%d_s%02d" % (MEMBER, STEP) %}

                gshc_model_{{ MEM_PREV_STEP }} => gshc_model_{{ MEM_STEP }}
=> gshc_ncdf_proc_{{ MEM_STEP }} => gshc_process_{{ MEM_STEP }} =>
gshc_archive_{{ MEM_STEP }}
                gshc_ncdf_proc_{{ MEM_PREV_STEP }} => gshc_ncdf_proc_{{
MEM_STEP }}
                gshc_process_{{ MEM_PREV_STEP }} => gshc_process_{{
MEM_STEP }}
```

```
                gshc_archive_{{ MEM_PREV_STEP }} => gshc_archive_{{
MEM_STEP }}
                gshc_ncdf_proc_{{ MEM_STEP }} => gshc_prod_filter_{{
MEM_STEP }} => gshc_process_{{ MEM_STEP }}
                gshc_prod_filter_{{ MEM_PREV_STEP }} => gshc_prod_filter_{{
MEM_STEP }}
                gshc_prod_filter_{{ MEM_STEP }} => gshc_prod_join_m{{
MEMBER }}
            {% endfor %}

            {% set MEM_LAST_STEP = "m%d_s%02d" % ( MEMBER, N_GSHC_STEPS) %}
            {% for STEP in range( 1, N_GSHC_STEPS) %}
                {% set MEM_STEP = "m%d_s%02d" % ( MEMBER, STEP) %}
                gshc_model_{{ MEM_STEP }}:fail | \
            {% endfor %}
            gshc_model_{{ MEM_LAST_STEP }}:fail | \
            gshc_register_member_m{{ MEMBER }}:fail => \
            gshc_model_m{{ MEMBER }}_failed & !gshc_register_member_m{{
MEMBER }}
            gshc_model_{{ MEM_LAST_STEP }} => !gshc_model_m{{ MEMBER
}}_failed
            gshc_model_m{{ MEMBER }}_failed => !GSHC_M{{ MEMBER }}

            (gshc_model_{{ MEM_LAST_STEP }} & gshc_archive_{{ MEM_LAST_STEP
}}) | \
            gshc_model_m{{ MEMBER }}_failed => housekeep

            gshc_prod_filter_{{ MEM_LAST_STEP }} => \
              gshc_prod_join_m{{ MEMBER }} => gshc_transfer_m{{ MEMBER }}

            gshc_transfer_m{{ MEMBER }} | gshc_model_m{{ MEMBER }}_failed
=> \
              housekeep
        {% endfor %}

        gsfc_start => gsfc_get_analysis => gsfc_redate_cice => gsfc_recon

        {% for MEMBER in range( 1, N_GSFC_MEMBERS + 1 ) %}
            gsfc_start & gsfc_recon => gsfc_model_m{{ MEMBER }}_s01 =>
gsfc_ncdf_proc_m{{ MEMBER }}_s01 => \
                gsfc_process_m{{ MEMBER }}_s01 => gsfc_archive_m{{ MEMBER
}}_s01

            gsfc_ncdf_proc_m{{ MEMBER }}_s01 => gsfc_prod_filter_m{{ MEMBER
}}_s01 => gsfc_process_m{{ MEMBER }}_s01
            gsfc_prod_filter_m{{ MEMBER }}_s01 => gsfc_prod_join_m{{ MEMBER
}}

            {% for STEP in range( 2, N_GSFC_STEPS + 1 ) %}
                {% set MEM_PREV_STEP = "m%d_s%02d" % ( MEMBER, STEP - 1) %}
                {% set MEM_STEP = "m%d_s%02d" % ( MEMBER, STEP) %}

                gsfc_model_{{ MEM_PREV_STEP }} => gsfc_model_{{ MEM_STEP }}
=> gsfc_ncdf_proc_{{ MEM_STEP }} => \
                gsfc_process_{{ MEM_STEP }} => gsfc_archive_{{ MEM_STEP
}}
```

```
                gsfc_ncdf_proc_{{ MEM_PREV_STEP }} => gsfc_ncdf_proc_{{
MEM_STEP }}
                gsfc_process_{{ MEM_PREV_STEP }} => gsfc_process_{{
MEM_STEP }}
                gsfc_archive_{{ MEM_PREV_STEP }} => gsfc_archive_{{
MEM_STEP }}
                gsfc_ncdf_proc_{{ MEM_STEP }} => gsfc_prod_filter_{{
MEM_STEP }} => gsfc_process_{{ MEM_STEP }}
                gsfc_prod_filter_{{ MEM_PREV_STEP }} => gsfc_prod_filter_{{
MEM_STEP }}
                gsfc_prod_filter_{{ MEM_STEP }} => gsfc_prod_join_m{{
MEMBER }}
            {% endfor %}

            {% set MEM_LAST_STEP = "m%d_s%02d" % ( MEMBER, N_GSFC_STEPS) %}
            {% for STEP in range( 1, N_GSFC_STEPS) %}
                {% set MEM_STEP = "m%d_s%02d" % ( MEMBER, STEP) %}
                gsfc_model_{{ MEM_STEP }}:fail | \
            {% endfor %}
            gsfc_model_{{ MEM_LAST_STEP }}:fail => gsfc_model_m{{ MEMBER
}}_failed
            gsfc_model_{{ MEM_LAST_STEP }} => !gsfc_model_m{{ MEMBER
}}_failed
            gsfc_model_m{{ MEMBER }}_failed => !GSFC_M{{ MEMBER }}

            (gsfc_model_{{ MEM_LAST_STEP }} & gsfc_archive_{{ MEM_LAST_STEP
}}) | gsfc_model_m{{ MEMBER }}_failed => housekeep
            gsfc_prod_filter_{{ MEM_LAST_STEP }} => gsfc_prod_join_m{{
MEMBER }} => gsfc_transfer_m{{ MEMBER }}
            gsfc_transfer_m{{ MEMBER }} | gsfc_model_m{{ MEMBER }}_failed
=> housekeep
        {% endfor %}
    housekeep => archive_logs
    """
```

7.3 APPENDIX C - GloSea5 workflow code in ecFlow

```
import ecflow
MEMBERS = 1
CHUNKS = 6

def create_setup():
    f=ecflow.Family("setup")
    f.add_variable('ECF_FILES',
'/home/jvegas/ecflow/GloSea5/GloSea5/setup')
    f.add_task('fcm_make_ocean')
    f.add_task('fcm_make2_ocean').add_trigger('fcm_make_ocean == complete')

    f.add_task('fcm_make_rebuild')
    f.add_task('fcm_make2_rebuild').add_trigger('fcm_make_rebuild ==
complete')

    f.add_task('fcm_make_redate')
    f.add_task('fcm_make2_redate').add_trigger('fcm_make_redate ==
complete')

    f.add_task('fcm_make_um')
```

```
    f.add_task('fcm_make2_um').add_trigger('fcm_make_um == complete')

    install_cold = f.add_task('install_cold')
    install_cold.add_part_trigger('fcm_make2_ocean == complete')
    install_cold.add_part_trigger('fcm_make2_rebuild == complete', True)
    install_cold.add_part_trigger('fcm_make2_redate == complete', True)
    install_cold.add_part_trigger('fcm_make2_um == complete', True)
    return f

def create_gsfc(hk):
    f = ecflow.Family('gsfc')
    cron = ecflow.Cron()
    cron.set_time_series(17,15)
    f.add_cron(cron)
    f.add_date(0,0,2016)
    f.add_trigger('setup == complete')
    f.add_task('start')
    f.add_task('get_analysis').add_trigger('start == complete')
    f.add_task('redate_cice').add_trigger('get_analysis == complete')
    f.add_task('recon').add_trigger('redate_cice == complete')

    for member in range(1, MEMBERS+1):
        fm = f.add_family('m{0}'.format(member))
        fm.add_variable('ECF_FILES',
'/home/jvegas/ecflow/GloSea5/GloSea5/gsfc')
        fm.add_trigger('recon == complete')
        failed = ecflow.Task('failed')
        for chunk in range (1, CHUNKS+1):
            fm.add_family(create_chunk_family(chunk))
            if chunk == 1:
                failed.add_part_trigger('{0}/model ==
aborted'.format(chunk))
            else:
                failed.add_part_trigger('{0}/model ==
aborted'.format(chunk), False)
        failed.add_part_complete('{0} == complete'.format(CHUNKS))
        failed.add_part_complete('transfer == complete', True)

        hk.add_part_trigger('gsfc/m{0}/failed == complete'.format(member))
        fm.add_task('prod_join').add_trigger('{0}/prod_filter ==
complete'.format(CHUNKS))
        fm.add_task('transfer').add_trigger('prod_join == complete')
        fm.add_task(failed)
    return f


def create_chunk_family(chunk):
    fc=ecflow.Family('{0}'.format(chunk))

    sim=fc.add_task('model')
    if chunk > 1:
        sim.add_part_trigger('../{0}/model == complete'.format(chunk -1))

    temp=fc.add_task('ncdf_proc')
    temp.add_part_trigger('model == complete')
    if chunk > 1:
```

```
        temp.add_part_trigger('../{0}/ncdf_proc == complete'.format(chunk -
1),True)

    temp=fc.add_task('prod_filter')
    temp.add_part_trigger('ncdf_proc == complete')
    if chunk > 1:
        temp.add_part_trigger('../{0}/prod_filter == complete'.format(chunk
-1), True)

    temp=fc.add_task('process')
    temp.add_part_trigger('prod_filter == complete')
    if chunk > 1:
        temp.add_part_trigger('../{0}/process == complete'.format(chunk -
1), True)

    temp=fc.add_task('archive')
    temp.add_part_trigger('process == complete')
    if chunk > 1:
        temp.add_part_trigger('../{0}/archive == complete'.format(chunk -
1), True)
    return fc


def create_gshc(hk):
    f = ecflow.Family('gshc')
    cron = ecflow.Cron()
    cron.set_time_series(17,15)
    f.add_cron(cron)
    f.add_date(0,0,2016)
    f.add_trigger('setup == complete')
    f.add_task('start')
    f.add_task('init_control_file').add_trigger('start== complete')

    for member in range(1, MEMBERS+1):
        fm = f.add_family('m{0}'.format(member))
        fm.add_variable('ECF_FILES',
'/home/jvegas/ecflow/GloSea5/GloSea5/gshc')
        fm.add_trigger('init_control_file == complete')
        fm.add_task('register_member')
        fm.add_task('get_analysis').add_trigger('register_member ==
complete')
        fm.add_task('recon').add_trigger('get_analysis == complete')
        failed = ecflow.Task('failed')
        failed.add_part_trigger('register_member == aborted')
        for chunk in range (1, CHUNKS+1):
            fm.add_family(create_chunk_family(chunk))
            failed.add_part_trigger('{0}/model == aborted'.format(chunk),
False)

        fm.add_task('prod_join').add_trigger('{0}/prod_filter ==
complete'.format(CHUNKS))
        fm.add_task('transfer').add_trigger('prod_join == complete')
        failed.add_part_complete('{0} == complete'.format(CHUNKS))
        failed.add_part_complete('transfer == complete', True)
        fm.add_task(failed)
        hk.add_part_trigger('gshc/m{0}/failed == complete'.format(member),
True)
```

```
    return f

print "Creating suite definition"
defs = ecflow.Defs()
suite = defs.add_suite("GloSea5")
suite.add_variable('ECF_INCLUDE','/home/jvegas/ecflow/GloSea5')
suite.add_variable('ECF_HOME','/home/jvegas/ecflow/GloSea5')
suite.add_family(create_setup())
hk = ecflow.Task('housekeep')
suite.add_family(create_gsfc(hk))
suite.add_family(create_gshc(hk))
suite.add_task(hk)
suite.add_task('archive_logs').add_trigger('housekeep == complete')
```

7.4 APPENDIX D - EC-Earth workflow code in Autosubmit

a) Experiment

```
[experiment]
DATELIST = 2000 2001 2002
MEMBERS = fc0 fc1 fc2 fc3 fc4
CHUNKSIZEUNIT = month
CHUNKSIZE = 3
NUMCHUNKS = 20
CALENDAR = standard
```

b) Jobs

```
[LOCAL_SETUP]
FILE = LOCAL_SETUP.sh
PLATFORM = LOCAL

[REMOTE_SETUP]
FILE = REMOTE_SETUP.sh
DEPENDENCIES = LOCAL_SETUP

[INI]
FILE = INI.sh
DEPENDENCIES = REMOTE_SETUP
RUNNING = member

[SIM]
FILE = SIM.sh
DEPENDENCIES = INI SIM-1 CLEAN-2
RUNNING = chunk

[POST]
FILE = POST.sh
DEPENDENCIES = SIM
RUNNING = chunk

[CLEAN]
FILE = CLEAN.sh
DEPENDENCIES = POST

[TRANSFER]
FILE = TRANSFER.sh
```

```
PLATFORM = LOCAL
DEPENDENCIES = CLEAN
RUNNING = member
```

7.5 APPENDIX E - EC-Earth workflow code in Cylc

a) Control suite

```
#!Jinja2

{% set DATES = {
    "2000-01":"2004-10",
    "2001-01":"2005-10",
    "2002-01":"2006-10"} %}

{% set MEMBERS = ["fc0", "fc1", "fc2", "fc3", "fc4"] %}

[scheduling]
    [[dependencies]]
        graph="""local => remote
            {% for ICP, FCP in DATES.iteritems() %}
            {% for MEMBER in MEMBERS %}
                remote => {{ ICP }}_{{ MEMBER }}
            {% endfor %}
            {% endfor %}
        """

[runtime]
    {% for MEMBER in MEMBERS %}
    [[{{ ICP }}_{{ MEMBER }}]]
        inherit={{ ICP }}
        command scripting = """
        set -xuve
        cylc register suite_{{ ICP }}_{{ MEMBER }} /home/jvegas/mem_suite
        cylc run --no-detach --set ICP={{ ICP }} --set FCP={{ FCP }} --set
MEMBER={{ MEMBER }} suite_{{ ICP }}_{{ MEMBER }}
        cylc unregister suite_{{ ICP }}_{{ MEMBER}}"""
    {% endfor %}
    {% endfor %}
```

b) Sub-suites

```
#!Jinja2

[cylc]
      cycle point format = %Y-%m
      [[environment]]
            MEMBER = {{ MEMBER }}

[scheduling]
      initial cycle point = {{ICP}}
      final cycle point = {{FCP}}
      max active cycle points = 2
      [[dependencies]]
                  [[[R1]]]
                        graph="ini => sim"
                  [[[P3M]]]
```

```
                        graph="""
                        sim[-P3M] => sim => post => clean
                        clean[-P6M] => sim
                        """
                  [[[R1//+P0D]]]
                        graph="""
                        CHUNK:succeed-all => transfer
                        """

[runtime]
      [[CHUNK]]

      [[ini, post, clean]]
            inherit = CHUNK
            command scripting = "sleep 2"
      [[sim]]
            inherit = CHUNK
            command scripting = "sleep 6"
      [[transfer]]
            command scripting = "sleep 4"
```

7.6 APPENDIX F - EC-Earth workflow code in ecFlow

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

import os
from dateutil.relativedelta import relativedelta
import ecflow

from datetime import date

START_DATE = date(2000,1,1)
END_DATE = date(2002,1,1)
SDATE_SEPARATION = 1
MEMBERS = 5
SIM_LENGTH= 5
CHUNK_SIZE = 3

def format_date(date):
    return date.strftime("%Y%m%d")

def create_family(name, path="" ):
    f=ecflow.Family(name)
    f.add_variable('ECF_FILES', ROOT_PATH + path)
    return f

def create_setup():
    f=create_family("setup","common" )
    f.add_task('localsetup')
    f.add_task('remotesetup').add_trigger('localsetup == complete')
    return f

def create_chunk_family(chunk, chunkdate, end_date):
    chunk_end = chunkdate + relativedelta(months=CHUNK_SIZE)
    fc=create_family(str(chunk),'ecearth3')

    sim=fc.add_task('sim')
```

```
    if chunk == 1:
        ini=fc.add_task('ini').add_trigger('/ecearth/setup == complete')
        sim.add_part_trigger('ini == complete')
    else:
        sim.add_part_trigger('../'+str(chunk-1)+'/sim == complete')

    if chunk > 2:
        sim.add_part_trigger('../'+str(chunk-2)+' == complete', True)

    fc.add_task('post').add_trigger('sim == complete')
    fc.add_task('clean').add_trigger('post == complete')
    return fc

def create_simulation(suite):
    startdate = START_DATE

    while startdate <= END_DATE:
        sdate = format_date(startdate)
        fsd=create_family(sdate)
        fsd.add_variable("SDATE", sdate)
        end_date=startdate + relativedelta(years=SIM_LENGTH)
        for nummember in range(0, MEMBERS):
            member = 'fc'+str(nummember)
            f=create_family(member, 'ecearth3')
            f.add_variable('MEMBER', member)
            date = startdate
            chunk=1
            while date < end_date:
                f.add_family(create_chunk_family(chunk, date, end_date))
                chunk += 1
                date = date + relativedelta(months=CHUNK_SIZE )

            transfer = fsd.add_task('transfer_'+member)
            transfer.add_trigger(member +' == complete')
            transfer.add_variable('ECF_FILES', ROOT_PATH + 'common')

            fsd.add_family(f)

        suite.add_family(fsd)

        startdate = startdate + relativedelta(years=SDATE_SEPARATION)

print "Creating suite definition"
defs = ecflow.Defs()
suite = defs.add_suite("ecearth")
suite.add_variable('ECF_INCLUDE', '/home/jvegas/ecflow/ecearth')
suite.add_variable("ECF_HOME", "/home/jvegas/ecflow/ecearth")

suite.add_family(create_setup())
create_simulation(suite)
```