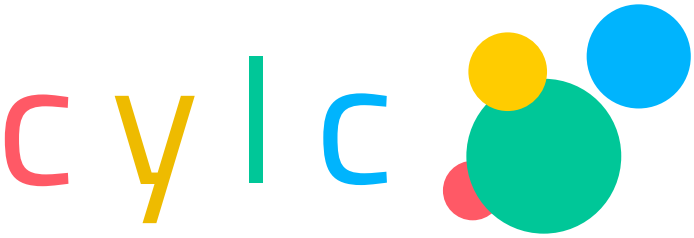


Cylc Lisbon Tutorial

Oliver Sanders, Hilary Oliver, and Ben Fitzpatrick

2016-08-03

Contents

1 Tutorial Scope

This is intended to be a one-day hands-on tutorial for new cylc users. Cylc, Rose, this document, and the tutorial example suites are installed on a Linux VM that can be downloaded from:

- http://https://github.com/metomi/metomi-vms

Automating large distributed workflows in operational environments can be complicated, and the full cylc user guide may look intimidating to new users as a result. Simplicity and ease of use for smaller workflows has always been a primary goal of the cylc project, however, and we have endeavored to convey this in the tutorial. Those who want to know more can refer to supplementary material provided in the Appendices, and other cylc documentation:

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

Cylc terminology will be defined and highlighted like this.

Note: *Pointers to additional information will be highlighted like this.*

Embedded tutorials are in shaded boxes like this.

1.1 A Note On Rose

Rose is an open source system for suite (workflow) management, designed for use with cylc.

- http://metomi.github.io/rose

It is installed on your VM if want to try it. Rose functionality includes:

- Suite storage, discovery, and revision control. This enables sharing and collaborative development of suites.
- Generic configuration, and metadata-driven configuration editing, of suites and tasks, including large scientific models.
- Various utilities for use with cylc suites.

The last bullet point includes some programs that will soon be migrated into the cylc project. We will be using two of these, pre-migration, in this tutorial: `rose suite-run` - a better way of running cylc suites; and `rose bush` - a sophisticated web-based suite log viewer.

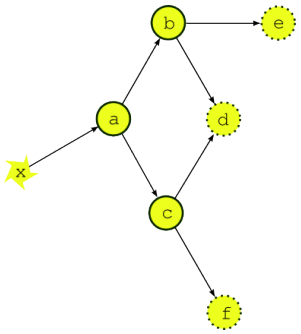
2 Cylc Overview

Note: *For more information on the topics of this section, see Appendix A and the Cylc User Guide.*

Cylc ("silk") is a workflow engine that can manage continuous workflows of cycling tasks for applications such as weather and environmental forecasting, and climate simulation.

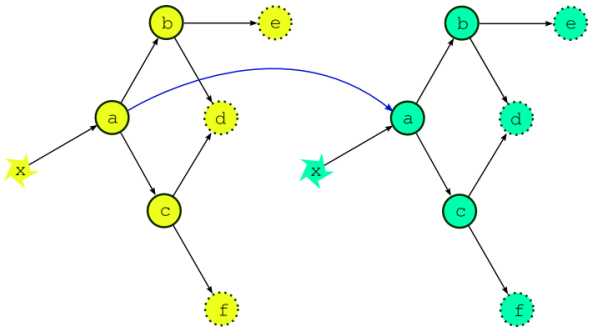
- *A task represents a job (a script or program) that runs on a computer.*
- *A workflow is a suite of interdependent tasks.*
- *A cycle point is a point on a date-time (or integer) sequence.*
- *A cycling task runs the same job repeatedly, for some sequence of cycle points.*
- *A continuous workflow of cycling tasks is a single workflow composed of cycling tasks, not just a series of separate single-cycle workflows, and it may extend indefinitely into the future.*

A small workflow of six tasks (with no cycling as yet) can be shown as a *dependency graph*:



where the graph *nodes* represent tasks, and *edges* (arrows) represent dependence. Task *a* might write out data files that are read in by tasks *b* and *c*, for instance. The node *x* here represents an external event such as receipt of new real-time data (weather observations, perhaps).

Before cycling this workflow we should be aware of *inter-cycle dependence*. For example, task *a* may depend on restart files generated by its own previous instance, like this:



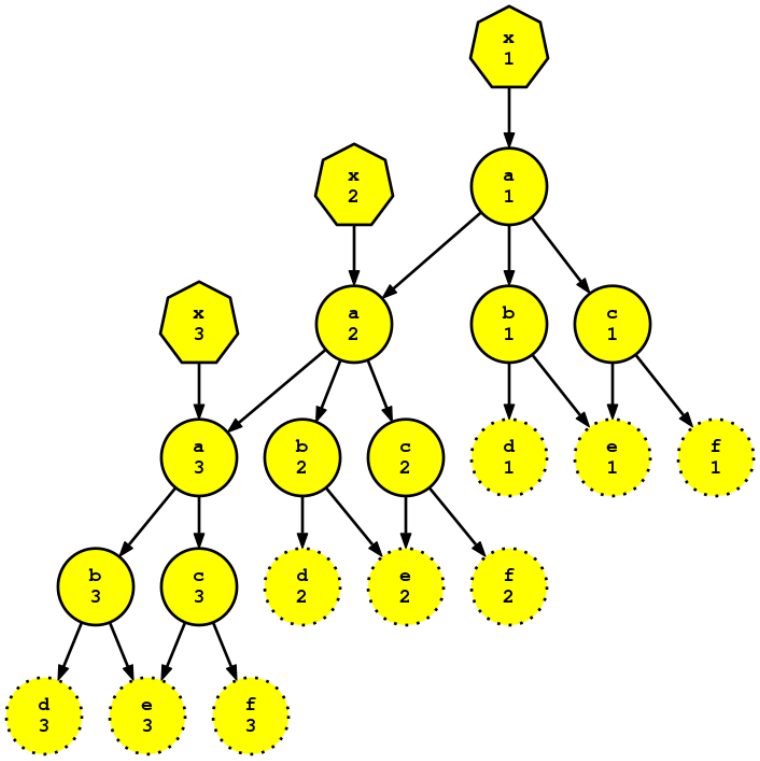
This inter-cycle dependence can be ignored *if* we agree always to finish one complete cycle before starting the next. This is usually the case in operational systems where there's a wait between cycles for new real time data (such as the latest weather observations). But if the suite has to catch up from a delay, or if we run it over archived historical data, we should be able to run tasks from multiple cycles at once for more efficient scheduling. Here, according to the dependencies shown, if green *x* is already done then green *a* should be able to start at the same time as yellow *b* and *c*, as soon as yellow *a* is finished.

Traditional fixed-cycling schedulers can't do this because they ignore inter-cycle dependence and must therefore run complete cycles sequentially.[1]

Cylc knows about inter-cycle dependence and manages the suite as *a single continuous workflow with no cycle boundaries*, not just a series of separate single-cycle-point workflows. In cylc, cycle points are merely

[1] Although some mitigate this problem by statically defining multiple steps or "chunks" within fixed cycles.

task labels, not global loop indices. Each task can run when its own inputs are satisfied, regardless of cycle point. Here's how cylc sees three cycles of our example suite (and note that this can go on indefinitely):



example

(The integers attached to each task are the cycle points). Consequently, cylc can run workflows faster, recover from delays faster, and handle failures and delays better than fixed-cycle schedulers. Appendix A shows actual job schedules for this example suite, and compares the result for a fixed-cycle scheduler.

3 Cylc Introduction

Note: *For more information on the topics of this section, see Appendix B and the Cylc User Guide.*

3.1 The suite.rc File

A cylc suite is a collection of files in a *suite directory* configured by a single *suite.rc* file, which is written in a nested INI format with section and sub-section (etc.) headings denoted by square brackets.

```
[section]
    option = value
    [[sub-section]]
        option = value
```

The most important top level sections in a suite.rc file are:

`[cylc]`	various suite-level settings
`[scheduling]`	determines *when* tasks are ready to run (e.g. dependencies)
`[runtime]`	defines *what* to run when a task is ready, and *where* and *how* to run it

3.2 Hello World in cylc

This suite runs a single task named `hello` that prints "Hello World!", sleeps for a few seconds, then exits:

```
[scheduling]
    [[dependencies]]
        graph = hello
[runtime]
    [[hello]]
        script = echo "Hello World!"; sleep 30
```

The `[scheduling]` section says to run the hello task immediately on start-up, because it doesn't depend anything else; and the `[runtime]` section says that the task should run the given inlined shell scripting.

You can run a new cylc suite like this (but don't do it just yet):

```
$ cylc register hello_world /path/to/hello_world/  # register name "hello_world"
$ cylc validate hello_world  # check for configuration errors
$ cylc run hello_world  # run the suite
$ cylc gui hello_world &  # open a suite control GUI
```

In this tutorial however, we will be using the `rose suite-run`, command to run suites. It automatically installs, registers, validates, starts the suite with `cylc run`, and opens the cylc GUI.

The `cylc run` command starts a new *suite daemon* to run your suite. It will stay alive even if you log out, until your suite runs to completion or you tell it to shut down.

A cylc suite daemon is a light-weight server program dedicated to managing a single workflow.

3.3 Hello World Tutorial

If you are running this tutorial on the cylc VM, the raw directory structure of the suite is set up for you. Change directory to *$HOME/tutorial/suites/hello_world/*.

Inside that directory is a cylc *suite.rc* file that looks like this:

```
[scheduling]
    [[dependencies]]
        graph = hello
[runtime]
    [[hello]]
        script = echo "Hello World!"; sleep 30
```

Now run the following commands:

```
$ rose suite-run  # install, register, validate, run the suite, and open the GUI
```

If no errors are found in the suite.rc file your suite will start up and a GUI window will appear showing the `hello` task with a coloured square representing its state. For example, green means 'running' and gray 'succeeded'. Once the task has succeeded, the suite has no more tasks to run and will shut down.

Note that "Hello World!" is not printed to the terminal. It is printed by the `hello` task, which is launched by the suite daemon as a separate process (potentially on another machine, although not in this case).

A task job script is a shell script generated by cylc to run a task as defined in the suite.rc file.

The job script and its output are written to a standard job log directory:

```
$ ls $HOME/cylc-run/hello_world/log/job/1/hello/01/
job  # task job script
job.out  # task standard out
job.err  # task standard error
job.status
job-activity.log
```

The task job logs are automatically retrieved to the suite host, if the task runs on another machine.

While a task is visible in the GUI, right-click on it to view its log files. After that, look in its log directory:

```
$ cd $HOME/cylc-run/hello_world/log/job/1/hello/01/
$ cat job.out
...
Hello World!
...
```

or use the `cylc cat-log` command:

```
$ cylc cat-log --help  # see "cylc --help" for top level command help
$ cylc cat-log --stdout hello_world hello.1
...
Hello World!
...
```

or use the `rose suite-log` web-based suite log file viewer (also known as *Rose Bush*):

```
$ cd ~/tutorial/suites/hello_world
$ rose suite-log
# (now view suite and job logs in your web browser)
```

3.4 Defining Tasks

A task can run an external program or script, or shell scripting inlined in the suite.rc file, or any combination of the two. A suite `bin` directory is automatically added to your shell `$PATH` so that scripts residing in it can be called by name,

```
[runtime]
    [[model]]
        script = run-model  # in <suite-dir>/bin/ (or elsewhere in $PATH)
```

or use a full file path,

```
[runtime]
    [[model]]
        script = /path/to/my-scripts/run-model
```

or set `$PATH` in the task environment (this makes more sense if the environment is inherited by multiple tasks - see Section 5.3),

```
[runtime]
    [[model]]
        script = run-model
        [[[environment]]]
            PATH = /path/to/my-scripts:$PATH
```

You can pass information to a script via its command line or environment,

```
[runtime]
    [[model]]
        script = run-model --color=green
        [[[environment]]]
            START_TIME = $CYLC_TASK_CYCLE_POINT
```

or use custom multi-line scripting, in triple quotes, to do anything you like:

```
[runtime]
    [[model]]
        script = """
cat > model-input.txt <<__EOF__
COLOR=green
START_TIME=$CYLC_TASK_CYCLE_POINT
__EOF__
run-model model-input.txt"""
```

An optional `[[[remote]]]` section determines *where* a task will run (defaults to localhost),

```
[runtime]
    [[model]]
        script = run-model
        [[[remote]]]
            host = supercomputer
```

(In this case the `run-model` script and any files that it needs - such as the model program itself - must be installed on host "supercomputer").

An optional `[[[job]]]` sub-section determines *how* the task should be submitted to run (defaults to "background", an ordinary shell subprocess).

```
[runtime]
    [[model]]
        script = run-model
        [[[job]]]
            batch system = pbs
```

Note: *So far we have only configured individual tasks. In fact the `[runtime]` section is a* multiple inheritance hierarchy *that allows all shared configuration to be factored out into* families *that are inherited by multiple tasks - see Section 5.3.*

3.5 Dependency Graphs

The hello_world suite contained the *graph string* `graph = hello_world`. Graph strings specify the scheduling logic that determines when tasks can run: which tasks depend on, or *trigger off*, which other tasks, if any. For instance if we have two tasks `foo` and `bar`, and `foo` depends on `bar` succeeding, we could write:

```
graph = "foo:succeed => bar"
```

If `foo` fails here, `bar` will not run. Tasks can also trigger off failure and other conditions, but success triggers are the default so `:succeed` can optionally be omitted:

```
graph = "foo => bar"
```

A graph string can contain many triggers, and the default success triggers can be chained together. This,

```
graph = "foo => bar => baz"
```

is equivalent to this,

```
graph = """foo => bar
           bar => baz"""
```

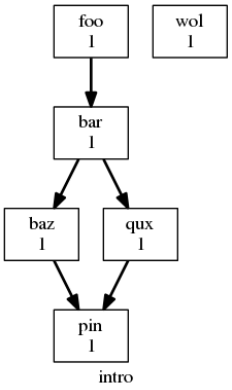
You can also write conditional triggering logic with `&` (AND) and `|` (OR) operators. For example,

```
graph = "foo => bar & baz"  # foo => bar, AND foo => baz
```

The `cylc graph` command generates nice suite graph visualizations. For this graph string,

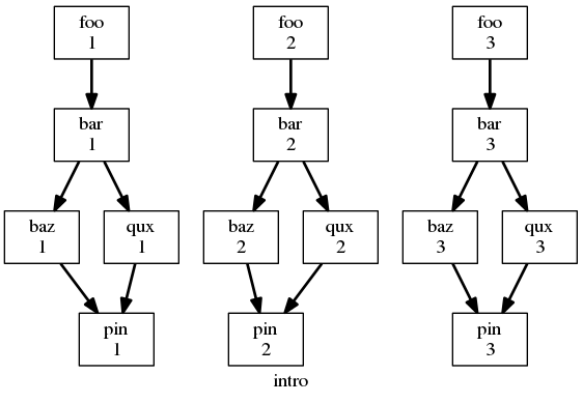
```
graph = """foo => bar => baz & qux => pin
           wol"""
```

it produces,



3.6 Cycling Workflows

The concept of a workflow of cycling tasks was introduced in Section 2. The following diagram shows the previous workflow (minus the lone 'wol' task) repeated for three integer cycle points 1, 2 and 3.



In cylc these are three distinct workflows that can run concurrently (which is how it should be if there is no dependence between them!). In fact many more than three cycle points could run concurrently, but we deliberately limit the amount of "runahead". The default is:

```
[scheduling]
    max active cycle points = 3
```

A cycling suite needs an *initial cycle point* and at least one cycling sequence with associated dependencies defined in a graph string. An optional *final cycle point* can also be given.

```
[cylc]
    cycle point format = %Y-%m
[scheduling]
    initial cycle point = 2000-01
    final cycle point = 2000-05
    [[dependencies]]
```

```
        [[[P1M]]]
            graph = model
```

This suite definition says to run task `model` for each cycle point on a sequence of date-times with a 1-month interval between 2000-01 and 2000-05 (see 5.1 for other kinds of cycling sequence). With no clock-triggers defined (see 5.10) these date-times have no connection to the real-time clock. Each instance of `model` is merely labelled with its cycle point value (which the task job can use, e.g. as a model run start date). In this case there is no dependence between successive `model` instances, so they can all run concurrently, out to `max active cycle points`.

3.7 Inter-Cycle Dependence

If each instance of `model` in the previous example really depends on its own previous instance (for restart files, say), running multiple models concurrently would result in failure. Here's how to express this inter-cycle dependence correctly,

```
[cylc]
    cycle point format = %Y-%m
[scheduling]
    initial cycle point = 2000-01
    final cycle point = 2000-05
    [[dependencies]]
        [[[P1M]]]
            graph = model[-P1M] => model
```

`P1D` is an ISO8601 duration - see http://wikipedia.org/wiki/ISO_8601#Durations. `P` denotes a duration and `1M` means one month. Other examples of ISO8601 durations are:

- `-PT12H` (12 hours ago).
- `-PT6H30M` (6 hours 30 minutes ago).
- `P1W` (1 week in the future).

3.8 Cycling Suite Tutorial

This demo is an example of a cycling workflow. On the cylc VM, the suite is located at *$HOME/tutorial/suites/rocket_cycling/*. `cd` to that directory.

The directory structure should look like this:

```
|-- bin/
|   `-- count-down
|-- rose-suite.conf
`-- suite.rc
```

The *suite.rc* file should look like this:

suite.rc

```
[scheduling]
    initial cycle point = 2000-01-01T00
    final cycle point = 2000-01-05T00
    [[dependencies]]
        [[[T00]]]
            graph = """

            blast_off[-P1D] => point_upwards
            point_upwards => load_astronauts & fill_fuel_tank & set_coordinates
            load_astronauts & set_coordinates => count_down
            fill_fuel_tank => light_fuse => count_down
            count_down => blast_off

            """
```

```
[runtime]
    [[point_upwards]]
        script = sleep 2; echo "spikey end pointing at sky, flamey end \
                    pointing at ground"
    [[load_astronauts]]
        script = sleep 1; echo "loaded astronauts"
    [[fill_fuel_tank]]
        script = sleep 5; echo "tank brimmed"
    [[set_coordinates]]
        script = echo "coordinates set for west wallaby st"
    [[light_fuse]]
        script = echo "stand well back"
    [[count_down]]
        script = count-down
    [[blast_off]]
        script = echo "blast off"
```

bin/count-down

```
#!/bin/bash
for count in {5..1}; do
    echo $count
    sleep 1
done
```

```
$ rose suite-run
```

```
View > 1 - Graph View
```

4 In-Depth Suite Writing Tutorial

4.1 Introduction

This tutorial walks you through creating a non-trivial suite based around the requirements of a piece of code that needs to be compiled, run, and supplied with inputs. Often, people won't create suites from scratch like this, but will base new suites on copies of existing ones.

4.2 Starting Out

This example supposes:

- We're on a sailing ship, making a passage
- We're navigating using a Fortran program

Our Fortran program reads in some position data and then pretends to calculate a new one based on a compass direction and a 5 knot speed for a given period of time.

If you are reading this tutorial via the cylc VM then the Fortran code is located at *$HOME/tutorial/-suites/navigation/src/dead_reckoning.f90*. Otherwise, the code is given in Appendix C.

We need to analyse our program to figure out what the dependencies are.

Have a quick look through the code and look for:

- what files or environment it needs to run
- what it produces
- when it might need to run

4.3 Inputs

The inputs to this Fortran code are:

- Two environment variables, *TIME_INTERVAL_HRS* and *POSITION_FILEPATH*
- An input (and output) file located at *$POSITION_FILEPATH* that stores the latitude and longitude.

When we run the compiled program, we'll need all these inputs to be present.

We want to run this program every 3 hours, on the hour - so there is a repeated dependency on the time or cycle.

We also want to build the program to begin with, so we need a compilation task that runs at the start.

4.4 Suite Creation

If you are running this tutorial on the cylc VM [2] , the raw directory structure of the suite is set up for you. Change directory to *$HOME/tutorial/suites/navigation/*. This is our suite top-level directory.

4.4.1 suite.rc file

An initial suite.rc file is already present in the *$HOME/tutorial/suites/navigation/* directory. It looks like this:

```
[cylc]
   UTC mode = True # Ignore DST
[scheduling]
   initial cycle point = 20160601T00Z # 1 June 2016
   final cycle point = 20160603T00Z # 3 June 2016
   [[dependencies]]
      [[[R1]]] # Run once at the start of the suite.
         graph = compile_navigate
[runtime]
   [[root]]
      pre-script = sleep 5 # Slow down tasks for visualization.
   [[compile_navigate]]
      script = """
         gfortran $CYLC_SUITE_DEF_PATH/src/dead_reckoning.f90 \
            -o $CYLC_SUITE_SHARE_PATH/dead_reckoning.exe
      """
```

We've set the suite to run from midnight, 1 June 2016 to midnight, 3 June 2016. This is just a model time or label. It isn't synchronised with real clock time.

We've set the `compile_navigate` task to run once at the initial cycle point (i.e. midnight, 1 June 2016). This will compile the source code and produce an executable in a directory *$HOME/cylc-run/navigation/share/*. This directory, *$CYLC_SUITE_SHARE_PATH* is provided by cylc as a place for inter-task 'sharing' of files.

All our tasks inherit from `root` and will run `sleep 5` from the `root` *pre-script* setting.

4.4.2 Initial Run

We can test that this works as it is.

Run `rose suite-run`[3].

The `compile_navigate` task should succeed, and the suite will shut down automatically. After that, there should be a newly created *dead_reckoning.exe* file under *$HOME/cylc-run/navigation/share/*. Have a look in that directory.

4.4.3 Adding a task

Let's now add our navigation task to run our executable. There are two steps involved in adding a task in the *suite.rc*:

[2] If and only if you are *not* running on the VM:

- create a new directory called *navigation*) somewhere in your homespace, which will be the top-level suite directory.
- inside that directory, create a subdirectory called *src*
- copy the Fortran code into a file called *src/dead_reckoning.f90*
- create a suite.rc file under *navigation* with the contents below in Section 4.4.1.

[3] Section 3.2 explained why we're running this instead of cylc commands.

- Add the task to the dependencies so cylc knows when to run it.
- Configure what the task actually does, in the runtime section!

Modify the *suite.rc* to have a dependencies section that looks like this:

```
[[dependencies]]
   [[[R1]]] # Run once at the start of the suite.
      graph = compile_navigate => navigate
   [[[PT3H]]] # Run every 3 hours (ISO 8601 date-time syntax).
      graph = navigate[-PT3H] => navigate
```

Here, we've made `navigate` run every 3 hours, with each instance waiting for the previous one to finish.

We've configured the dependency between `compile_navigate` and `navigate`.

We also need to tell the `navigate` task what to run. Add a new entry under the runtime section:

```
[[navigate]]
   script = $CYLC_SUITE_SHARE_PATH/dead_reckoning.exe
   [[[environment]]]
      POSITION_FILEPATH = $CYLC_SUITE_SHARE_PATH/position
      TIME_INTERVAL_HRS = 3
```

This instructs cylc to run the executable made by `compile_navigate` and sets up the necessary environment variables.

Our navigation executable will read and write the *$POSITION_FILEPATH* position file. Each run of the executable will read the previous run's output. However, at the first cycle point, the file will not currently exist. We'll need to make it and include start coordinates.

4.4.4 Adding another initial-only task

We'll add a task called `write_start_position` to create this initial file. Add it as a run-once (*R1*) task by replacing the graph for the `[[[R1]]]` section with:

```
   [[[R1]]] # Run once at the start of the suite.
      graph = """
         compile_navigate => navigate
         write_start_position => navigate
      """
```

Finally, add these lines under the `runtime` section, at the end of the suite.rc file:

```
[[write_start_position]]
   script = echo 50.0 -3.0 >$CYLC_SUITE_SHARE_PATH/position
```

This initialises our location for the `navigate` task via a file (pointed to via *$POSITION_FILEPATH*). Our start coordinates are 50.0 north, 3.0 west, which is in the English Channel (a.k.a. La Manche, Canal da Mancha, etc). You can change these to another location if you like.

4.5 Checking the suite

Your suite should now look something like this:

```
[cylc]
   UTC mode = True # Ignore DST
[scheduling]
   initial cycle point = 20160601T00Z # 1 June 2016
   final cycle point = 20160603T00Z # 3 June 2016
   [[dependencies]]
      [[[R1]]] # Run once at the start of the suite.
         graph = """
            compile_navigate => navigate
            write_start_position => navigate
         """
      [[[PT3H]]] # Run every 3 hours (ISO 8601 date-time syntax).
```

```
         graph = navigate[-PT3H] => navigate
[runtime]
   [[root]]
      pre-script = sleep 5 # Slow down tasks for visualization.
   [[compile_navigate]]
      script = """
         gfortran $CYLC_SUITE_DEF_PATH/src/dead_reckoning.f90 \
            -o $CYLC_SUITE_SHARE_PATH/dead_reckoning.exe
      """
   [[navigate]]
      script = $CYLC_SUITE_SHARE_PATH/dead_reckoning.exe
      [[[environment]]]
         POSITION_FILEPATH = $CYLC_SUITE_SHARE_PATH/position
         TIME_INTERVAL_HRS = 3
   [[write_start_position]]
      script = echo 50.0 -3.0 >$CYLC_SUITE_SHARE_PATH/position
```

4.6 Running our Suite

Run:

```
$ rose suite-run
```

If everything has been set up successfully, after running that command, `cylc gui` will launch with your running suite.

4.7 Finished Output

You can look at the finished output by running:

```
$ rose suite-log
```

The position will be written in the out file for each navigate task.

4.7.1 Nicer Output

If you want a quick and easy way of visualising the output, try replacing the line:

```
PRINT*, "New position, me hearties:",new_lat," ",new_long
```

in *dead_reckoning.f90* with

```
lat = (180.0/pi) * lat
long = (180.0/pi) * long
CALL get_environment_variable("CYLC_TASK_LOG_ROOT",value=task_log_root,status=code)
OPEN(1,file=TRIM(task_log_root) // '-map.html',action='write')
WRITE(1,'(A)') &
"<html><head><link rel='stylesheet'",&
"href='https://unpkg.com/leaflet@1.0.0-rc.3/dist/leaflet.css' /></head>",&
"<body><div id='map' style='width: 1000px; height: 600px'></div>",&
"<script src='https://unpkg.com/leaflet@1.0.0-rc.3/dist/leaflet.js'></script>"
WRITE(1,'(A, A, A, A F7.4, A, F7.4, A, F7.4, A, F7.4, A)') &
"<script>var map = L.map('map');",&
"L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png',",&
"{attribution: '&copy; <a href=""http://osm.org/copyright"">OpenStreetMap</a>",&
"contributors'}).addTo(map);var polyline = L.polyline([[",&
lat,", ",long,"],[",new_lat,", ",new_long,&
"]], {color: 'red'}).addTo(map);map.fitBounds(polyline.getBounds());"
WRITE(1,'(A F7.4, A, F7.4, A)') &
"var start = L.marker([",lat,", ",long,"]).addTo(map);"
WRITE(1,'(A F7.4, A, F7.4, A)') &
"var end = L.marker([",new_lat,", ",new_long,"]).addTo(map);"
WRITE(1,'(A)') &
"start.bindPopup('Start');",&
"end.bindPopup('End').openPopup();",&
"map.zoomOut(2);</script></body></html>"
CLOSE(1)
```

Note: *Beware of pdf copy-and-paste introducing line breaks before and after the asterisks (*) in the code above.*

This will produce a *job-map.html* file per job which you can click through to in `Rose Bush` (via running `rose suite-log`).

4.8 More Cycling

Our suite has a start cycle point and a single cycling period of 3 hours, but we could have other cycle definitions in the same suite. Let's add a task called `take_sun_sight` that runs at 12:00 each day. This will correct our latitude.

Add these lines below `[[dependencies]]` in your suite.rc file:

```
[[[T12]]] # Run at 12:00Z each day.
   graph = navigate => take_sun_sight
```

This will run after the `navigate` task from that cycle point finishes.

In order to make the `navigate` task wait for our new `take_sun_sight` task, we'll need to add some extra configuration for the 15 hour cycle - add the following lines in the same way as you did for `[[[T12]]]`:

```
[[[T15]]] # Run at 15:00Z each day.
   graph = take_sun_sight[-PT3H] => navigate
```

4.9 Adding a Script to the Suite

You can put scripts in the *bin/* directory of a suite, and they will be added to cylc's path.

Change directory to the root of your suite, and create a *bin/* directory.

Create an empty file in the *bin/* directory called *bin/sun_sight*.

Open this file with a text editor. Paste the following text into the *sun_sight* file:

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

import random
import sys


if __name__ == "__main__":
   random.seed()
   with open(sys.argv[1], "r") as f:
      (lat, long) = f.read().split()
   lat = float(lat) + random.uniform(-0.05, 0.05)
   print "Yarr! Our corrected position be {0}, {1}".format(lat, long)
   with open(sys.argv[1], "w") as f:
      f.write("{0} {1}\n".format(lat, long))
```

Save the file and make sure the indentation is correct. Make it executable by running:

```
$ chmod +x bin/sun_sight
```

We need to reference this script explicitly in the *suite.rc* for our `take_sun_sight` task - append these lines to the *suite.rc* file:

```
[[take_sun_sight]]
   script = sun_sight $CYLC_SUITE_SHARE_PATH/position
```

4.10 Results

Run the suite by invoking:

```
$ rose suite-run
```

Our extra task should run at 20160601T1200Z and 20160602T1200Z. If it does, congratulations!

If you run into any problems, you can refer to a completed suite at *$HOME/tutorial/suites/navigation-complete*.

You can check the graphing by running:

```
$ cylc graph navigation 20160601T0000Z 20160603T0000Z
```

4.11 Rose (optional - feel free to skip)

4.11.1 Introduction

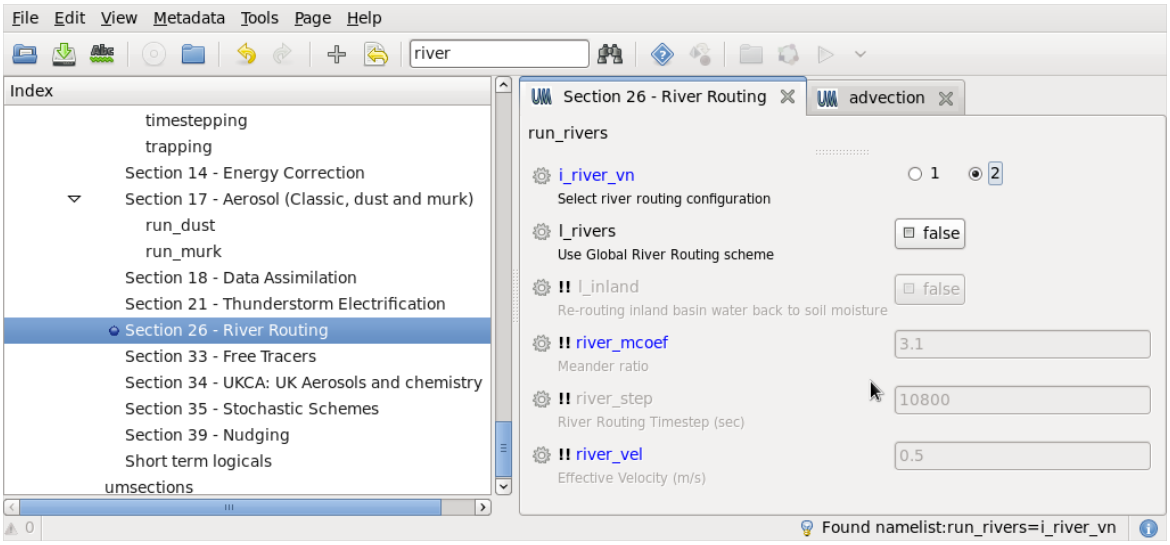


Figure 1: rose edit, a generic configuration GUI showing a dynamically generated panel for a section in the inputs for the Met Office Unified Model. The contents of the panel are generated from the relevant part of a Rose app file plus some simple metadata, both specified in flat INI-based text files.

rose-app.conf snippet used to drive Figure 1

```
[namelist:run_rivers]
i_river_vn=1
!!l_inland=.false.
l_rivers=.false.
!!river_mcoef=3.1
!!river_step=10800
!!river_vel=0.5
```

Rose has a powerful set of utilities for the configuration and running of executables, suite storage and version control, and also includes some top-level suite running functionality that will be migrated to cylc.

In this part of the tutorial, we'll demonstrate a couple of pieces of Rose functionality that can enhance suites and tasks:

- Top-level suite.rc configuration - *rose-suite.conf*
- Metadata for the top-level inputs in *rose-suite.conf*, which is used for checking and presentation via `rose edit`

- *app* configuration to configure inputs for particular executables in tasks
- Metadata for the *app*, as for the suite, and `rose edit` presentation of it

4.11.2 rose-suite.conf

Let's pull out our initial starting longitude and latitude into top-level *rose-suite.conf* Jinja2 template inputs. Jinja2 allows us to effectively script the *suite.rc* file when we need to.

Add this line to the top of the suite.rc file:

```
#!jinja2
```

and replace:

```
   [[write_start_position]]
      script = echo 50.0 -3.0 > $CYLC_SUITE_SHARE_PATH/position
```

with:

```
   [[write_start_position]]
      script = """
         echo {{START_LATITUDE}} {{START_LONGITUDE}} > \
            $CYLC_SUITE_SHARE_PATH/position
      """
```

We can now pull out `START_LATITUDE` and `START_LONGITUDE` into top-level *rose-suite.conf* inputs. Open the (currently empty) *rose-suite.conf* file and change it to read:

```
[jinja2:suite.rc]
START_LATITUDE=50.0
START_LONGITUDE=-3.0
```

When the suite is run with `rose suite-run`, these variables will be inserted into the `write_start_position` task's script and it will run as before.

4.11.3 rose-suite.conf Metadata

Rose has a concept of metadata for inputs like these, which help check their validity and improve the presentation in the `rose edit` configuration editor GUI. We can take a shortcut to fill out some of that metadata.

In the top-level directory of your suite, run:

```
$ rose metadata-gen --auto-type
```

It will create a subdirectory called *meta* with a file called *rose-meta.conf* inside. Open that file and edit it to add some description and allowed value ranges:

```
[jinja2:suite.rc]
title=Starting Position

[jinja2:suite.rc=START_LATITUDE]
compulsory=true
description=Starting latitude in degrees
range=-90:90
type=real

[jinja2:suite.rc=START_LONGITUDE]
compulsory=true
description=Starting longitude in degrees
range=-180:180
type=real
```

Now run the command `rose edit` - this launches a nice GUI interface to our inputs which will check for errors. Try e.g altering the longitude to 300 degrees. An error should display.

You can run the suite directly through `rose edit` via the 'Play' button in the toolbar (although for safety, this button will be disabled if there are unsaved changes).

4.11.4 Rose Apps (rose-app.conf)

We can also add configuration for our Fortran program via a Rose app configuration. This allows us to pull out internal task-specific configuration into a separate location and add metadata and GUI support for it. Inter-task configuration, such as the `$POSITION_FILEPATH` variable, would normally belong in the suite.rc file where it helps to define workflow.

Alter the `navigate` task `script` in the `suite.rc` file to read:

```
[[navigate]]
   script = rose task-run -v
```

and delete the `[[[environment]]]` settings for the task there too.

Create an *app/navigate/* subdirectory in your suite:

```
$ mkdir -p app/navigate/
```

Create a file underneath that directory called *rose-app.conf* (*app/navigate/rose-app.conf*). Alter that file to read:

```
[command]
default=$CYLC_SUITE_SHARE_PATH/dead_reckoning.exe

[env]
POSITION_FILEPATH=$CYLC_SUITE_SHARE_PATH/position
TIME_INTERVAL_HRS=3
```

The new `[env]` section in the app replaces the `[[[environment]]]` settings for `[[navigate]]` in the *suite.rc* file.

4.11.5 rose-app.conf Metadata

Create a further *meta* subdirectory for some app metadata:

```
$ mkdir -p app/navigate/meta
```

Inside that new directory, create a *rose-meta.conf* file (*app/navigate/meta/rose-meta.conf*) that looks like this:

```
[env=TIME_INTERVAL_HRS]
range=1:24
type=integer
```

If you save these files and then launch `rose edit`, you should see configuration appear for the `navigate` app. As we have declared `TIME_INTERVAL_HRS` to be an integer, it should have a numeric widget. It should display an error if the value is less than 1 or greater than 24.

Note: *This tutorial covered only a limited subset of the functionality that Rose apps and metadata can provide - there is also inter-variable triggering, a Pythonic mini-language, dynamic page assignment, deep Fortran namelist integration, and much more. See the Rose documentation for more details.*

5 Advanced Tutorials

This section outlines some of the more advanced features of cylc along with links to tutorials on these features on the rose documentation website.

The final example from each tutorial has been provided as a pre-built suite on the Metomi VM for convenience. These suites are referenced here in the "Demo Suite" sections. If you don't want to go work though the full tutorials these suites will give you working examples of the tutorials topic.

5.1 Advanced Cycling

So far we have defined cycling using graph section headings of the form `[[[TXX]]]` as in the following example:

```
[scheduling]
    initial cycle point = 2000-01-01T00
    [[dependencies]]
        [[[T00]]]  # Run every day at 00:00
            graph = foo => bar
```

This is the simplest form of graph section heading. More powerful forms exist to help define complex recurrences.

They all follow abbreviations of a basic format - `Rn/DATE-TIME/REPEAT_INTERVAL`, an ISO 8601[4] recurrence format, where `n` is an optional limit on repetitions, `DATE-TIME` is a starting date-time, and `REPEAT_INTERVAL` is the duration between successive recurrences.

A date-time without a repeat interval must have a missing unit of date or time to extrapolate a repeat interval from. `T00` above is short for `2000-01-01T00`, and the absence of higher-level day information is taken to mean 'daily'. It is therefore ultimately short for `R/2000-01-01T00/P1D`. This means: starting at `2000-01-01T00:00:00`, repeat every day (`P1D`) indefinitely (`R` with no number limit suffix).

`T12` would mean the same thing but starting at `2000-01-01T12:00:00`. Another example - `01T00` - has missing month information, so it is taken to mean 'monthly' or `R/2000-01-01T00/P1M`.

A repeat interval without a date-time is taken to use the initial cycle point as the start date-time. `P2D` is taken to mean `R/2000-01-01T00/P2D`.

A single `R1` is taken to mean 'run once at the initial cycle point'.

Examples

`[[[ T06, T1845 ]]]`	Run every day at 06:00 and 18:45
`[[[ 01T00 ]]]`	Run every month on the first of the month
`[[[ PT15M ]]]`	Run every 5 minutes
`[[[ +P5D/P1M ]]]`	Run every month, starting 5 days after the initial cycle point
`[[[ R3/T06 ]]]`	Run three times, once every day at 06:00

Further Reading: http://cylc.github.io/cylc/html/single/cug-html.html#9.3.4

5.2 Advanced Dependencies

The & symbol can be used to condense multiple graph lines, for instance the following example:

```
graph = """
    foo => bar
    foo => baz
"""
```

[4] https://en.wikipedia.org/wiki/ISO_8601

... can be condensed to `graph = foo => bar & baz`. Graph lines can also be written with the | symbol meaning or, for example in the following example `baz` will run as soon as either `foo` or `bar` succeed.

```
graph = foo | bar => baz
```

Up until now we have written dependencies in the form `foo => bar` which means `bar` will trigger (run) as soon as `foo` succeeds. It is possible to trigger tasks off of other states e.g:

`foo:fail => bar`	`bar` triggers if `foo` fails
`foo:submit => bar`	`bar` triggers once `foo` has submitted
`foo:start => bar`	`bar` triggers once `foo` starts executing
`foo:finish => bar`	`bar` triggers once `foo` succeeds or fails

5.3 Families

Often a suite will contain a collection of similar tasks. With cylc these tasks can be grouped together for convenience. This can be used to factor out a lot of repeated configuration to make suites more readable and maintainable.

Example In the following example the task `hello_eris` inherits the `script` setting and `IS_WORLD_A_PLANET` environment variable from the family `HELLO_FAIMILY`. The task `hello_pluto` on the other hand inherits the `script` setting but overrides the `IS_WORLD_A_PLANET` environment variable.

```
[scheduling]
    [[dependencies]]
        graph = HELLO_FAMILY  # Runs all tasks that inherit from HELLO_FAMILY.
[runtime]
    [[HELLO_FAMILY]]  # A family.
        script = echo $IS_WORLD_A_PLANET
        [[[environment]]]
            IS_WORLD_A_PLANET = false # Environment variable shared with tasks
                                      # that inherit from this family.
    [[hello_eris]]
        inherit = HELLO_FAMILY
    [[hello_pluto]]
        inherit = HELLO_FAMILY
        [[[environment]]]
            IS_WORLD_A_PLANET = true  # Overrides the inherited environment
                                      # variable.
```

Tutorial: http://metomi.github.io/rose/doc/rose-rug-advanced-tutorials-family-trigs.html

Demo Suite: ~/tutorial/suites/family-triggers

5.4 Retries

Sometimes tasks fail, you can tell cylc to automatically retry failed tasks.

Example

```
[runtime]
    [[task]]
        retry delays = 5*PT10S  # Retry up to 5 times waiting 10 seconds before
                                # each retry.
```

Tutorial: http://metomi.github.io/rose/doc/rose-rug-advanced-tutorials-retries.html

Demo Suite: ~/tutorial/suites/retries

5.5 Jinja2

Jinja2 (http://jinja.pocoo.org/) is a *template processor* - essentially a programming language that can be embedded in arbitrary text, for the purpose of altering that text. In cylc, you can think of Jinja2 as providing programming constructs (variables, data structures, loops, etc.) for *generating your suite definition*.

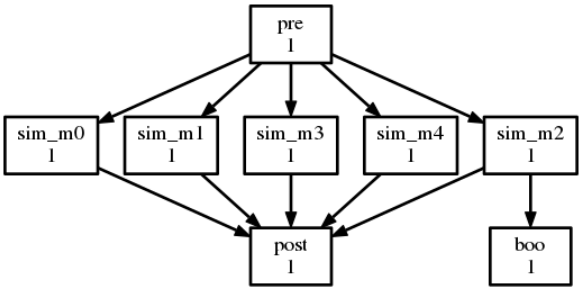
Example

```
#!jinja2
{% set N_MEMS = 5 -%}
[scheduling]
    [[dependencies]]
        graph = """
{% for MEM in range(N_MEMS) -%}
     pre => sim_m{{MEM}} => post
{% endfor -%}
        sim_m2 => boo"""
[runtime]
{% for MEM in range(N_MEMS) -%}
    [[sim_m{{MEM}}]]
        script = echo "I'm member {{MEM}}"
{% endfor -%}
    [[boo]]
        script = echo "BOO!"
```

When this file is parsed Jinja2 generates the final suite definition, which you can see with `cylc view -j`:

```
[scheduling]
    [[dependencies]]
        graph = """
    pre => sim_m0 => post
    pre => sim_m1 => post
    pre => sim_m2 => post
    pre => sim_m3 => post
    pre => sim_m4 => post
        sim_m2 => boo"""
[runtime]
    [[sim_m0]]
        script = echo "I'm member 0"
    [[sim_m1]]
        script = echo "I'm member 1"
    [[sim_m2]]
        script = echo "I'm member 2"
    [[sim_m3]]
        script = echo "I'm member 3"
    [[sim_m4]]
        script = echo "I'm member 4"
    [[boo]]
        script = echo "BOO!"
```

The `cylc graph` command shows the suite like this:



Tutorial: http://metomi.github.io/rose/doc/rose-rug-advanced-tutorials-jinja2.html

Demo Suite: ~/tutorial/suites/jinja2

5.6 Parameterized Tasks

Since 6.11.0, parameter expansion can be used instead of messy Jinja2 loops to generate tasks. Here is the parameterized task version of the small Jinja2 example just above:

```
[cylc]
    [[parameters]]
        m = 0..4
[scheduling]
    [[dependencies]]
        graph = """pre => sim<m> => post
                   sim<m=2> => boo"""
[runtime]
    [[sim<m>]]
        script = echo "I'm member $CYLC_TASK_PARAM_m"
    [[boo]]
        script = echo "BOO!"
```

This produces exactly the same result as the Jinja2 version, but it is much clearer. Multiple parameters can be used at once, instead of nested Jinja2 loops. For more on this topic see the Cylc User Guide.

5.7 Cylc Broadcast

`cylc broadcast` is a command line utility that can be used to change any setting contained within the `[runtime]` section of a suite.rc file whilst the suite is running.

Example The following line of bash script will set the remote host for the task `task_name`.

```
$ cylc broadcast <suite_name> -n <task_name> -s [remote]host=<host_name>
```

Tutorial: http://metomi.github.io/rose/doc/rose-rug-advanced-tutorials-broadcast.html

Demo Suite: ~/tutorial/suites/broadcast

5.8 Suicide Triggers

Suicide triggers can be used to remove tasks from a suite's graph during runtime.

Example The task `recover_from_failure` will be removed from the graph if the task `task` succeeds.

```
[scheduling]
    [[dependencies]]
        graph = """
            task:fail => recover_from_failure  # Run recover_from_failure if
                                               # task fails.
            task => !recover_from_failure      # Don't run if task succeeds.
        """
```

Tutorial: http://metomi.github.io/rose/doc/rose-rug-advanced-tutorials-suicide.html

Demo Suite: ~/tutorial/suites/suicide-triggers

5.9 Queues

Queues can be used to limit the number of certain tasks that are submitted or run at any given time.

Example In this example all `foo`, `bar` and `baz` tasks will go to the `task_queue` queue. This queue will only allow two tasks to run at a time.

```
[scheduling]
    [[queues]]
        [[[task_queue]]]
            limit = 2
            members = foo, bar, baz
```

Tutorial: http://metomi.github.io/rose/doc/rose-rug-advanced-tutorials-queues.html

Demo Suite: ~/tutorial/suites/queues

5.10 Clock Triggered Tasks

Sometimes tasks should wait until a certain time before running, in cylc this is possible using "clock triggering".

- see 5.10)

Example

```
[scheduling]
    [[dependencies]]
        [[[T00]]]
            graph = daily_task
    [[special tasks]]
        clock-triggered = daily_task(PT0H)   # Run daily_task with 0 hours
                                             # offset from the cycle point.
```

Tutorial: http://metomi.github.io/rose/doc/rose-rug-advanced-tutorials-clock-triggered.html.

Demo Suite: ~/tutorial/suites/clock-triggers

A Appendix: Cylc Overview

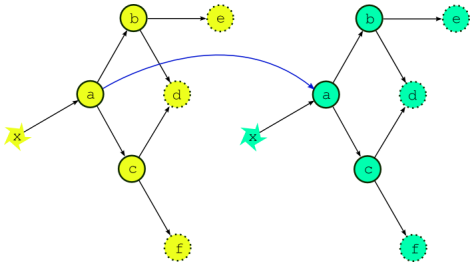
Note: *This Appendix contains extra explanatory material for Section 2.*

A task represents a job (a script or program) that runs on a computer. We make this distinction because a *job* exists only when it runs, but its representation in a cylc suite has a longer lifetime. For instance, a "waiting" task represents a job that will run sometime in the future because its inputs have not been satisfied yet.

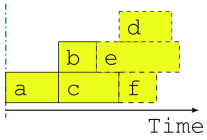
A cycle point is a point on an integer or date-time sequence. Note that a date-time cycle point has no connection to real time unless you attach a *clock trigger* to a task. Clock-triggers say that in addition to any dependence on other tasks, a task cannot trigger unless the wall-clock time is greater than or equal to its cycle point, or some offset from that point. See Section 5.10.

A continuous workflow of cycling tasks is a single workflow composed of cycling tasks, not just a series of separate single-cycle workflows, and it may extend indefinitely into the future. This describes cylc's classic *iterative cycling* mode in which the scheduler continually extends the workflow to future cycle points as the suite runs, and each instance of a repeated job is represented by the same logical task with a different cycle point. *Parameterized cycling* is an alternative to this for workflows that are *finite in extent and not too large*: each instance of a repeated job is represented by a different logical task with a different task name, and the entire workflow is mapped out at start-up rather than extended as the suite runs. Parameterized cycling is generally not as flexible or powerful as iterative cycling, and it has several significant disadvantages, but it can sometimes be useful - see the cylc user guide for more information.

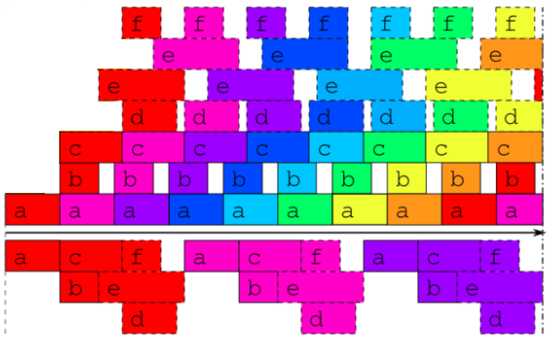
The Section 2 example workflow is repeated here:



Here's a job schedule for a single cycle of the workflow,



Bar width is proportional to job run time, and the vertical axis has no meaning. So *b* and *c* start running at the same time, immediately after *a* finishes, and so on. The job schedule for repeatedly cycling the same workflow is shown below, for cylc (top) and a traditional fixed-cycle scheduler (bottom)



The different colours represent different cycle points. Cylc automatically interleave cycles for faster scheduling throughput. In this case cylc is running tasks from four different cycle points at once, most of the

time. The optimal result is shown (the white time-gaps between tasks are are required by the dependency relationshionships). This assumes sufficient compute resource to run every task as soon as its inputs are satisified. But if a task is delayed (in a batch scheduler queue or otherwise) the system organically adapts, and the rest of the workflow will carry on as dependencies allow.

B Appendix: Cylc Introduction

Note: *This Appendix contains extra explanatory material for Section 3.*

B.1 The suite.rc File Format

Cylc suites are defined in a simple human-readable text format, by design, because complex workflow definitions are best managed like program source code, with dedicated revision control power tools such as git and subversion that work with text-based diffs. This allows proper branch-and-merge collaborative development of complex workflows.

B.2 The Minimal Cylc Suite, and Dummy Tasks

The Hello World suite is not quite the simplest working cylc suite: you can omit the runtime configuration for a task and cylc will automatically create a dummy job for it that just prints some information and exits:

```
[scheduling]
    [[dependencies]]
        graph = hello
[runtime]
    [[hello]]  # (empty)
```

In fact, if all tasks in a suite are dummy tasks, you can omit the entire runtime section:

```
[scheduling]
    [[dependencies]]
        graph = hello
```

This can be useful for mocking up suites, and temporarily "dummying out" real tasks. Note however that tasks without even an empty runtime section heading will fail strict validation: `cylc validate --strict`. This is to catch misspelled task names in the graph, which will create accidental "naked dummy tasks".

B.3 Suite Registration

`cylc-register` just associates a name with a suite definition. The reason for this is that most cylc commands interact with a running suite daemon, by name, rather than a suite.rc file. Commands that do parse a suite.rc file (e.g. `cylc validate`) can refer to the suite name or the file path, whereas commands that connect to a suite daemon must use the suite name.

B.4 Suite Daemons

Some workflow schedulers have a large central server program to run suites for all users. These have significant admin overheads (user accounts) and security requirements (elevated system privileges are required in order to submit jobs for many users).

`cylc run` starts a dedicated light-weight server program just for your workflow, running under your normal Unix user account, it submits jobs just as you would. By default it is a Unix *daemon* process that detaches from your terminal and stays alive when you log out. To stop a suite from daemonizing use `cylc run --no-detach`.

B.5 rose suite-run

`rose suite-run` will soon be migrated into cylc. It is not just a convenient short-cut command. More importantly it *installs* the suite into its run directory and registers it there. This separates the live suite

from its source so you can work on a suite without interfering with a running instance, and it provides an opportunity to install external files into the suite at start-up.

B.6 rose-suite.conf

`rose suite-run` aborts if no rose-suite.conf file is found in the suite directory. An empty file will suffice here, but note that you can use it to supply input variables to the suite definition. Together with similar Rose configuration files for tasks, and associated metadata, the generic Rose config editor can provide, with very little effort, a sophisticated configuration GUI for a suite and all of its tasks. See Rose documentation for more on this.

B.7 Suite Log Directories

```
$HOME/cylc-run/hello_world/log/job/1/hello/01/
```

The directory path contains the suite name (hello_world), log type (*job* or *suite*), task cycle point (1 for a non-cycling suite), task name (hello), and task submit number (01). This structure prevents re-submitted jobs from overwriting their previous logs, and it avoids unmanageable flat log directories in large or long-running suites.

B.8 Defining Tasks

You can run any existing program or script in a cylc task so long as it:

- returns standard exit status (zero for success, non-zero for error) to allow automatic error detection
- waits on any internal processes before exiting (e.g. jobs submitted internally to a batch scheduler)

Shell scripts unfortunately do not abort on error by default, but you can force them to do so by putting `set -e` at the top. It is also a good idea to use `set -u` to abort if an undefined variable is referenced.

Scripts that spawn detaching processes internally have to be modified, to make them wait before exiting - otherwise it is impossible for cylc to determine when the job is truly finished.

A task job script is a shell script generated by cylc to run a task as defined in the suite.rc file.

Task job scripts wrap the configured task scripting and environment in code to automatically trap errors and communicate progress back to the suite daemon.

Variables from suite.rc task environment section are exported in the job script, and several scripting items get inserted verbatim into it:

- `init-script` - runs at the top of job script
- `env-script` - runs just before task environment variables are exported
- `pre-script` - runs just before the main `script` item
- `script` - this is the main `script` item
- `post-script` - runs just after the main `script` item

Multiple script items are provided to allow families of tasks to inherent common blocks of code as well as having unique main script content.

B.9 Remote Task Hosts

Cylc doesn't install files to remote hosts for you because in general it is not possible to automatically determine what files might be needed. If a task host does not share a filesystem with the suite host, make sure you install any files needed by the tasks that run there, or define some initital tasks to do file installation for you. Rose can help to automate file installation at suite start-up.

B.10 Supported Batch Systems

As well as background jobs and the simple Unix `at` scheduler, cylc supports various batch systems such as PBS and SLURM, for job submission, queue interrogation, and job cancellation or kill. These generally require use of a `[[[directives]]]` section too, to select the right queue, set wall clock limits, and so on.

C Appendix: In-Depth Suite Writing Tutorial

Note: *This Appendix contains extra explanatory material for Section 4.*

C.1 Fortran navigation code

```
PROGRAM extract_compass_log
IMPLICIT NONE
CHARACTER(31) :: dt_hr_env ! No. of hours, from environment
CHARACTER(255) :: pos_fpath ! File path to a position file with lat/long
INTEGER:: code ! iostat code
INTEGER:: timevalues(8) ! time values
INTEGER, PARAMETER :: real64=SELECTED_REAL_KIND(15, 307)
REAL(KIND=real64) :: ang_distance ! Angular distance travelled across Earth
REAL(KIND=real64) :: dt_hr ! No. of hours elapsed, from dt_hr_env
REAL(KIND=real64) :: heading ! Compass heading in radians
REAL(KIND=real64) :: lat ! Initial latitude
REAL(KIND=real64) :: long ! Initial longitude
REAL(KIND=real64) :: new_lat ! Final latitude
REAL(KIND=real64) :: new_long ! Final longitude
REAL(KIND=real64) :: speed_kn=5.0 ! Speed in knots
REAL(KIND=real64), PARAMETER :: pi=3.141592654 ! pi
REAL(KIND=real64), PARAMETER :: radius_earth_nm=3443.89 ! Earth radius (nm)

! Get position file location via $POSITION_FILEPATH
CALL get_environment_variable("POSITION_FILEPATH",value=pos_fpath,status=code)
IF (code /= 0) THEN
  WRITE(0,*) "$POSITION_FILEPATH: not set."
  STOP 1
END IF

! Read in starting latitude and longitude
OPEN(1,file=pos_fpath,action="read",iostat=code)
IF (code /= 0) THEN
  WRITE(0,*) pos_fpath,": position file read failed."
  STOP 1
END IF
READ(1,*) lat,long
CLOSE(1)

! Convert to radians, where they belong
lat = (pi/180.0) * lat
long = (pi/180.0) * long

! Read in our duration input
CALL get_environment_variable("TIME_INTERVAL_HRS",value=dt_hr_env,status=code)
IF (code /= 0) THEN
  WRITE(0,*) "$TIME_INTERVAL_HRS: not set"
  STOP 1
END IF
READ(dt_hr_env,*) dt_hr

! Pretend to extract an average heading from the ship's compass
CALL date_and_time(VALUES=timevalues)
heading = mod(1000 * timevalues(7) + timevalues(8), 60) * 2 * pi / 60

! This is how far we went, in radians:
! (1 knot = 1 nautical mile / 1 hour)
ang_distance = (speed_kn*dt_hr) / radius_earth_nm
```

```
! Get the new latitude and longitude
new_lat = ASIN(SIN(lat) * COS(ang_distance) + &
               COS(lat) * SIN(ang_distance) * COS(heading))
new_long = long + &
           ATAN2(SIN(heading) * SIN(ang_distance) * COS(lat), &
                 COS(ang_distance) - SIN(lat) * SIN(new_lat))
new_lat = (180.0/pi) * new_lat
new_long = (180.0/pi) * new_long

PRINT*, "New position, me hearties:",new_lat," ",new_long

! Overwrite position file with new lat and long
OPEN(1,file=pos_fpath,action='write')
WRITE(1,*) new_lat,new_long
CLOSE(1)
END PROGRAM
```