The Cylc Forecast Suite Metascheduler User Guide

Cylc Version 4.0.0-rc3

Released Under the GNU GPL v3.0 Software License

Hilary Oliver

November 3, 2011

NIWA
Taihoro Nukurangi

Abstract

Cylc ("silk") is a metascheduler[1] for cycling environmental forecasting suites containing many forecast models and associated processing tasks. Cylc has a novel self-organising scheduling algorithm: a pool of task proxy objects, that each know just their own inputs and outputs, negotiate dependencies so that correct scheduling emerges naturally at run time. Cylc does not group tasks artificially by forecast cycle[2] (each task has a private cycle time and is self-spawning - there is no suite-wide cycle time) and handles dependencies within and between cycles equally so that tasks from multiple cycles can run at once to the maximum possible extent. This matters in particular whenever the external driving data[3] for upcoming cycles are available in advance: cylc suites can catch up from delays very quickly, parallel test suites can be started behind the main operation to catch up quickly, and one can likewise achieve greater throughput in historical case studies; the usual sequence of distinct forecast cycles emerges naturally if a suite catches up to real time operation. Cylc can easily use existing tasks and can run suites distributed across a heterogenous network. Suites can be stopped and restarted in any state of operation, and they dynamically adapt to insertion and removal of tasks, and to delays or failures in particular tasks or in the external environment: tasks not directly affected will carry on cycling as normal while the problem is addressed, and then the affected tasks will catch up as quickly as possible. Cylc has comprehensive command line and graphical interfaces, including a dependency graph based suite control GUI. Other notable features include suite databases; a fast simulation mode; a structured, validated suite definition file format; dependency graph plotting; task event hooks for centralized alerting; and cryptographic suite security.

[1] A metascheduler determines when dependent jobs are *ready to run* and then submits them to run by other means, usually a batch queue scheduler. The term can also refer to an aggregate view of multiple distributed resource managers, but that is not the topic of this document. We drop the "meta" prefix from here on because a metascheduler is also a type of scheduler.

[2] A *forecast cycle* comprises all tasks with a common *cycle time*, i.e. the analysis time or nominal start time of a forecast model, or that of the associated forecast model(s) for other tasks.

[3] Forecast suites are typically driven by real time observational data or timely model fields from an external forecasting system.

Contents

List of Figures

1 Introduction: How Cylc Works

1.1 Scheduling Forecast Suites

Environmental forecasting suites generate forecast products from a potentially large group of interdependent scientific models and associated data processing tasks. They are constrained by availability of external driving data: typically one or more tasks will wait on real time observations and/or model data from an external system, and these will drive other downstream tasks, and so on. The dependency diagram for a single forecast cycle in such a system is a *Directed Acyclic Graph* as shown in Figure 1 (in our terminology, a *forecast cycle* is comprised of all tasks with a common *cycle time*, which is the nominal analysis time or start time of the forecast models in the group). In real time operation processing will consist of a series of distinct forecast cycles that are each initiated, after a gap, by arrival of the new cycle's external driving data.

From a job scheduling perspective task execution order in such a system must be carefully controlled in order to avoid dependency violations. Ideally, each task should be queued for execution at the instant its last prerequisite is satisfied; this is the best that can be done even if queued tasks are not able to execute immediately because of resource contention.

1.2 EcoConnect

Cylc was developed for the EcoConnect Forecasting System at NIWA (National Institute of Water and Atmospheric Research, New Zealand). EcoConnect takes real time atmospheric and stream flow observations, and operational global weather forecasts from the Met Office (UK), and uses these to drive global sea state and regional data assimilating weather models, which in turn drive regional sea state, storm surge, and catchment river models, plus tide prediction, and a large number of associated data collection, quality control, preprocessing, postprocessing, product generation, and archiving tasks.[4] The global sea state forecast runs once daily. The regional weather forecast runs four times daily but it supplies surface winds and pressure to several downstream models that run only twice daily, and precipitation accumulations to catchment river models that run on an hourly cycle assimilating real time stream flow observations and using the most recently available regional weather forecast. EcoConnect runs on heterogenous distributed hardware, including a massively parallel supercomputer and several Linux servers.

1.3 Dependence Between Tasks

1.3.1 Intracycle Dependence

Most inter-task dependence exist within a single forecast cycle. Figure 1 shows the dependency diagram for a single forecast cycle of a simple example suite of three forecast models (*a, b,* and *c*) and three post processing or product generation tasks (*d, e* and *f*). A scheduler capable of handling this must manage, within a single forecast cycle, multiple parallel streams of execution that branch when one task generates output for several downstream tasks, and merge when one task takes input from several upstream tasks.

Figure 2 shows the optimal job schedule for two consecutive cycles of the example suite in real time operation, given execution times represented by the horizontal extent of the task bars. There is a time gap between cycles as the suite waits on new external driving data. Each

[4] Future plans for EcoConnect include additional deterministic regional weather forecasts and a statistical ensemble.

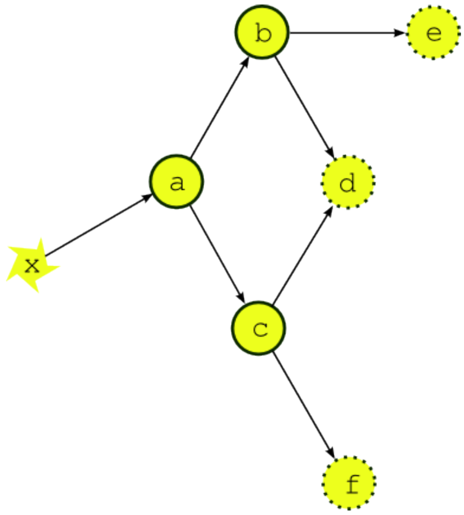

Figure 1: The dependency graph for a single forecast cycle of a simple example suite. Tasks *a, b,* and *c* represent forecast models, *d, e* and *f* are post processing or product generation tasks, and *x* represents external data that the upstream forecast model depends on.

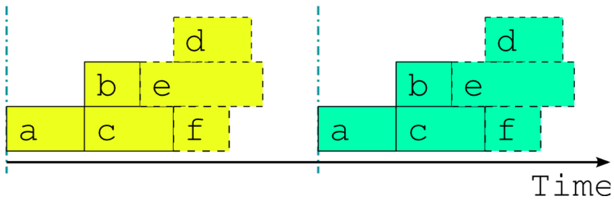

Figure 2: The optimal job schedule for two consecutive cycles of our example suite during real time operation, assuming that all tasks trigger off upstream tasks finishing completely. The horizontal extent of a task bar represents its execution time, and the vertical blue lines show when the external driving data becomes available.

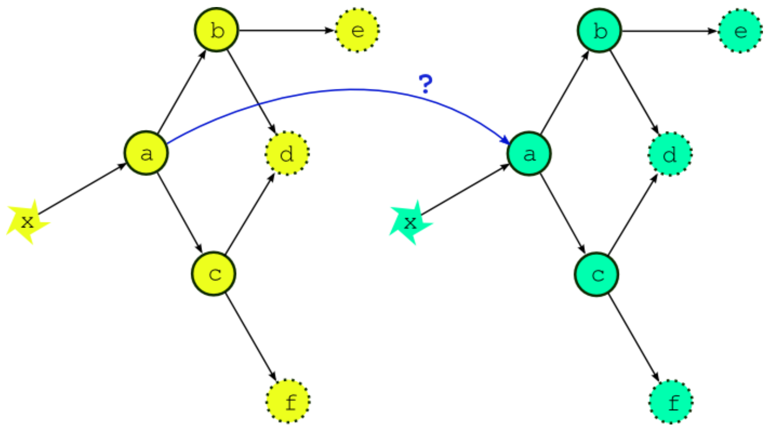

Figure 3: If the external driving data is available in advance, can we start running the next cycle early?

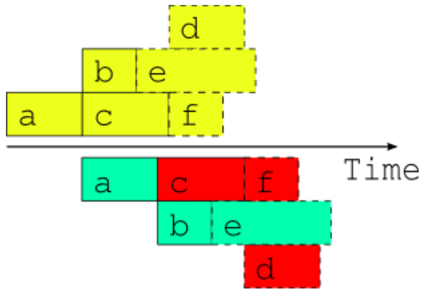

Figure 4: A naive attempt to overlap two consecutive cycles using the single-cycle dependency graph. The red shaded tasks will fail because of dependency violations (or will not be able to run because of upstream dependency violations).

task in the example suite happens to trigger off upstream tasks *finishing*, rather than off any intermediate output or event; this is merely a simplification that makes for clearer diagrams.

Now the question arises, what happens if the external driving data for upcoming cycles is available in advance, as it would be after a significant delay in operations, or when running a historical case study? While the forecast model *a* appears to depend only on the external data *x* at this stage of the discussion, in fact it would typically also depend on its own previous instance for the model *background state* used in initializing the new forecast. Thus, as alluded to in Figure 3, task *a* could in principle start as soon as its predecessor has finished. Figure 4 shows, however, that starting a whole new cycle at this point is dangerous - it results in dependency violations in half of the tasks in the example suite. In fact the situation is even worse than this - imagine that task *b* in the first cycle is delayed for any reason *after* the second cycle has been

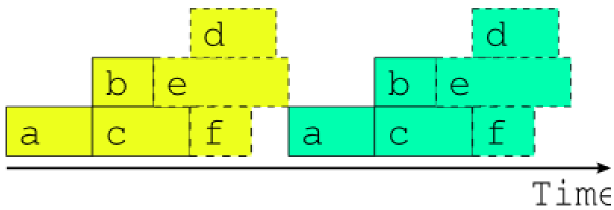

Figure 5: The best that can be done *in general* when intercycle dependence is ignored.

launched? Clearly we must consider handling intercycle dependence explicitly or else agree not to start the next cycle early, as is illustrated in Figure 5.

1.3.2 Intercycle Dependence

In most suites dependence between tasks in different cycles exist. Forecast models, as above, typically depend on their own most recent previous forecast for a background state, and different types of tasks in different forecast cycles can also be linked (in an atmospheric forecast analysis suite, for instance, the weather model may also generate background states for use by the observation processing and data-assimilation systems in the next cycle). In real time operation this intercycle dependence can be ignored because it is automatically satisfied when each cycle finishes before the next one begins. This is just as well because they dramatically increase the complexity of the dependency graph of even the simplest suites, by destroying the clean boundary between forecast cycles. Figure 6 illustrates the problem for our simple example suite assuming the minimal likely intercycle dependence: the forecast models (a, b, and c) each depend on their own previous instances.

For this reason, and because we tend to imagine that forecasting suites always run in distinct cycles, existing metaschedulers (as far as the author is aware!) ignore intercycle dependence and therefore *require* a series of distinct cycles at all times. While this does not affect normal real time operation it can be a serious impediment when advance availability of external driving data makes it possible, in principle, to run some tasks from upcoming cycles before the current cycle is finished - as suggested at the end of the previous section. This occurs after delays (late arrival of external data, system maintenance, etc.) and, to an even greater extent, in historical case studies, and parallel test suites that are delayed with respect to the main operation. It is a serious problem, in particular, for suites that have little downtime between forecast cycles and therefore take many cycles to catch up after a delay. Without taking account of intercycle dependence, the best that can be done, in general, is to reduce the gap between cycles to zero as shown in Figure 5. A limited crude overlap of the single cycle job schedule may be possible for specific task sets but the allowable overlap may change if new tasks are added, and it is still dangerous: it amounts to running different parts of a dependent system as if they were not dependent and as such it cannot be guaranteed that some unforeseen delay in one cycle, after the next cycle has begun, (e.g. due to resource contention or task failures) won't result in dependency violations.

Figure 7 shows, in contrast to Figure 4, the optimal two cycle job schedule obtained by respecting all intercycle dependence. This assumes no delays due to resource contention or otherwise - i.e. every task runs as soon as it is ready to run. The scheduler running this suite must be able to adapt dynamically to external conditions that impact on multicycle scheduling in the presence of intercycle dependence or else, again, risk bringing the system down with dependency violations.

To further illustrate the potential benefits of proper intercycle dependency handling, Figure 8 shows an operational delay of almost one whole cycle in a suite with little downtime between cycles. Above the time axis is the optimal schedule that is possible, in principle, when intercycle dependence is taken into account, and below is the only safe schedule possible *in general* when they are ignored. In the former case, even the cycle immediately after the delay is hardly affected, and subsequent cycles are all on time, whilst in the latter case it takes five full cycles to catch up to normal real time operation.

Similarly, Figure 9 shows example suite job schedules for an historical case study, or when catching up after a very long delay; i.e. when the external driving data are available many cycles in advance. Task a, which as the most upstream forecast model is likely to be a resource

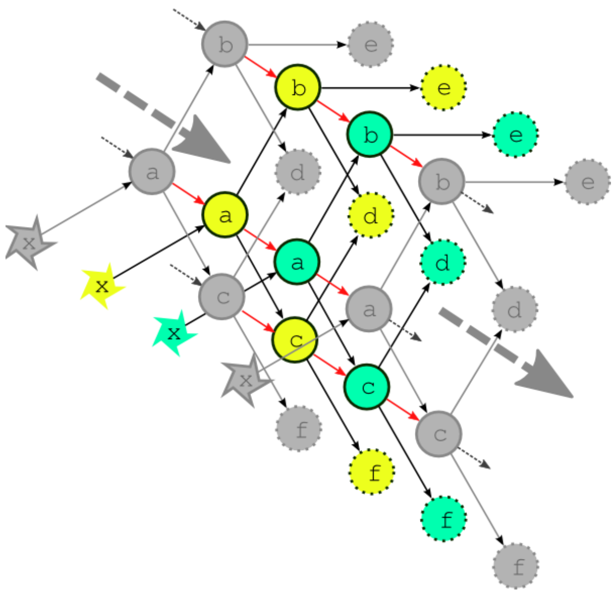

Figure 6: The complete dependency graph for the example suite, assuming the least possible intercycle dependence: the forecast models (a, b, and c) depend on their own previous instances. The dashed arrows show connections to previous and subsequent forecast cycles.

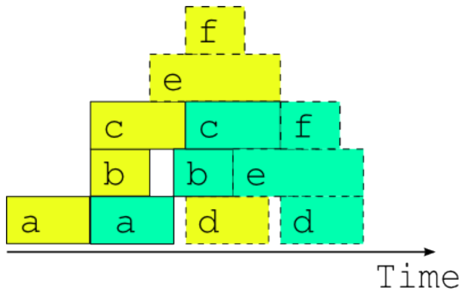

Figure 7: The optimal two cycle job schedule when the next cycle's driving data is available in advance, possible in principle when intercycle dependence is handled explicitly.

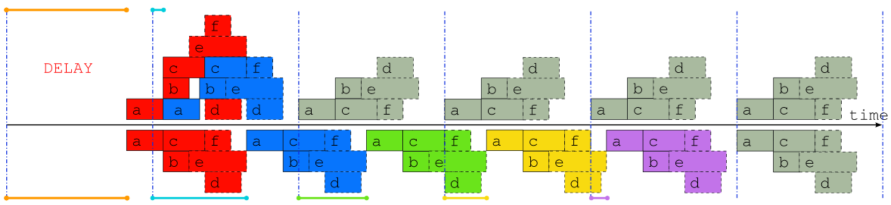

Figure 8: Job schedules for the example suite after a delay of almost one whole forecast cycle, when intercycle dependence is taken into account (above the time axis), and when it is not (below the time axis). The colored lines indicate the time that each cycle is delayed, and normal "caught up" cycles are shaded gray.

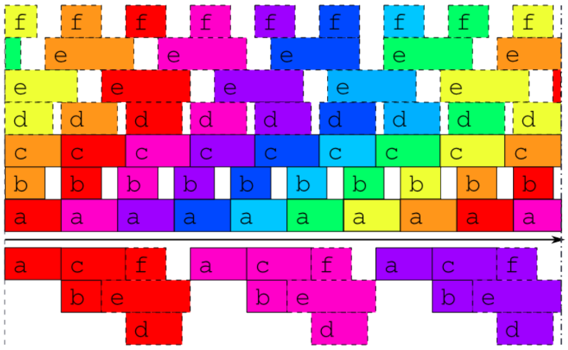

Figure 9: Job schedules for the example suite in case study mode, or after a long delay, when the external driving data are available many cycles in advance. Above the time axis is the optimal schedule obtained when the suite is constrained only by its true dependencies, as in Figure 3, and underneath is the best that can be done, in general, when intercycle dependence is ignored.

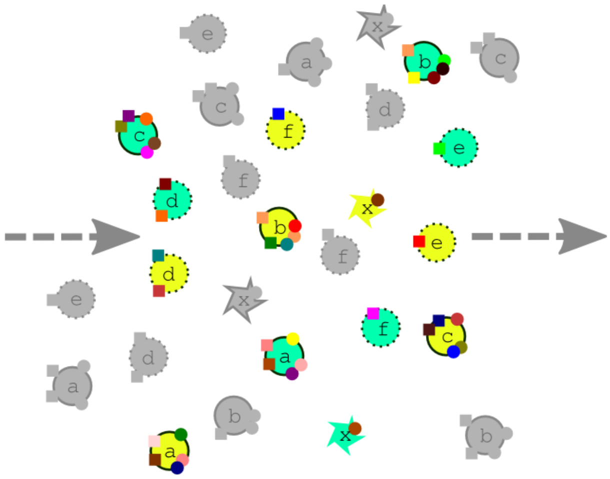

Figure 10: How cylc sees a suite, in contrast to the multicycle dependency graph of Figure 6. Task colors represent different cycle times, and the small squares and circles represent different prerequisites and outputs. A task can run when its prerequisites are satisfied by the outputs of other tasks in the pool.

intensive atmosphere or ocean model, has no upstream dependence on cotemporal tasks and can therefore run continuously, regardless of how much downstream processing is yet to be completed in its own, or any previous, forecast cycle (actually, task a does depend on cotemporal task x which waits on the external driving data, but that returns immediately when the external data is available in advance, so the result stands). The other forecast models can also cycle continuously or with short gap between, and some post processing tasks, which have no previous-instance dependence, can run continuously or even overlap (e.g. e in this case). Thus, even for this very simple example suite, tasks from three or four different cycles can in principle run simultaneously at any given time. In fact, if our tasks are able to trigger off internal outputs of upstream tasks, rather than waiting on full completion, successive instances of the forecast models could overlap as well (because model restart outputs are generally completed early in the forecast) for an even more efficient job schedule.

1.4 The Cylc Scheduling Algorithm

Cylc manages a pool of proxy objects that represent real tasks in the forecasting suite. A task proxy can run the real task that it represents when its prerequisites are satisfied, and can receive reports of completed outputs from the real task as it runs. There is no global cycling mechanism to advance the suite in time; instead each individual task proxy has a private cycle time and spawns its own successor. Task proxies are self-contained - they just know their own prerequisites and outputs and are not aware of the wider suite context. Intercycle dependencies are not treated as special, and the task pool can be populated with tasks from many different cycle times. The cylc task pool is illustrated in Figure 10. Now, *whenever any task changes state due to completion of an output, every task checks to see if its own prerequisites are now satisfied.*[5] Moreover, this matching of prerequisites and outputs involves the entire task pool, regardless of individual cycle times, so that inter- and intra-cycle dependence is handled with ease.

Thus without using global cycling mechanisms, and treating all inter-task dependence equally, cylc in effect gets a pool of tasks to self-organise by negotiating their own dependencies so that optimal scheduling, as described in the previous section, emerges naturally at run time.

2 Installation

2.1 Requirements

- Operating System: Linux or Unix [6]
- Python Version: 2.4 or later, but not Python 3.x as yet.[7]
- PyGTK, a Python wrapper for the GTK+ graphical user interface toolkit. PyGTK is included in most Linux Distributions.
  http://www.pygtk.org
- Pyro 3 (latest version tested 3.14.)[8]
  http://irmen.home.xs4all.nl/pyro3
- The graphviz graph layout engine (latest version tested: 2.28.0).
  http://www.graphviz.org
- Pygraphviz, a python interface to graphviz (latest version tested: 1.1).
  http://networkx.lanl.gov/pygraphviz

Python and Pyro are essential. PyGTK is required by the gcylc GUI (but you can control and monitor cylc suites from the command line). Graphviz and pygraphviz are required for dependency graphing and the graph-based suite control GUI (but you can also run cylc without them).

Cylc has absorbed the following software in modified form (no need to install them):

- xdot, a graph viewer (http://code.google.com/p/jrfonseca/wiki/XDot, LGPL license)
- ConfigObj and Validate (http://www.voidspace.org.uk/python, BSD license)

[5] In fact this dependency negotiation goes through a broker object (rather than every task literally checking every other task) which scales as n (rather than n^2) where n is the number of task proxies in the pool.

[6] The cylc codebase assumes Unix-style file paths in places, but it could easily made more portable if necessary.

[7] Python 2.4 was released in November 2004. Python 3 is the future of Python, but it is not backward compatible with 2.x and consequently still has significantly less library and third party support. As of mid 2011, Python 2.7 is still the standard for new Linux distributions.

[8] As of April 2011, Pyro 4, which is compatible with Python 3, is in development but it is still not recommended for production use.

2.2 Unpack The Cylc Tarball

Cylc installs into a normal user account (although it can be installed at system-level); just unpack the release tarball in the desired location, which we shall henceforth call `$CYLC_DIR`.

2.3 Configure Your Environment

The file `$CYLC_DIR/cylc.profile` documents the minimal user environment configuration required for interactive cylc usage:

```
#!/bin/bash

# CYLC USER LOGIN SCRIPT EXAMPLE. Copy the following to your .profile
# and adapt according to your preferences and local cylc installation.

# These environment variables are required for interactive cylc usage.
# Access to cylc by running tasks is automatically configured by cylc.

#______________________________________________________________________
# Add the cylc bin directory to $PATH. The 'cylc' and 'gcylc' commands
# configure access to cylc sub-commands and python modules at run time.
export PATH=/path/to/cylc/bin:$PATH

#______________________________________________________________________
# For 'cylc edit' or gcylc -> Edit, set terminal and GUI editors:
export EDITOR=vim
export GEDITOR='gvim -f'
# (See 'cylc edit --help' for examples of other editors).

#______________________________________________________________________
# To access the Cylc User Guide via the gcylc GUI menus:
export PDF_READER=evince
export HTML_READER=firefox
# (The HTML guide is opened via file path, not http URL).

#______________________________________________________________________
# Some cylc commands require $TMPDIR to be set and writeable.
export TMPDIR=/path/to/my/temporary/directory

#______________________________________________________________________
# For a local user install of Pyro, Graphviz, and Pygraphviz (if you
# can't easily get them installed at system level on the cylc host):
### export PYTHONPATH=$HOME/external/installed/lib64/python2.6/site-packages:$PYTHONPATH
### export PATH=$HOME/external/installed/bin:$PATH # (graphviz/bin not required by cylc)
# (See the Cylc User Guide "Installation" Section for instructions).
```

Copy this into your `.profile` login script and adapt appropriately for your environment. After sourcing your modified login script, or logging in again, you should be able to run cylc:

```
% cylc --version
x.y.z
```

2.4 Create The Central Suite Database

Cylc has a central suite database, visible to all users on the cylc host, to facilitate sharing of suites. Run the following command immediately after installation to create the central database and export cylc's example suites to it:

```
% cylc admin create-cdb
```

To view the content of the resulting central database, run gcylc and switch to the central database using the Database menu, or use `cylc db print` on the command line:

```
% cylc db print --central --tree -x 'Auto|Quick'
<admin>
 `-cylc-x:y:z
```

```
  |-AutoCleanup
  | |-FamilyFailHook  family failure hook script example
  | `-FamilyFailTask  family failure cleanup task example
  |-AutoRecover
  | |-async           asynchronous automated failure recovery example
  | `-cycling         cycling automated failure recovery example
  `-QuickStart
    |-a         Quick Start Example A
    |-b         Quick Start Example B
    |-c         Quick Start Example C
    `-z         Quick Start Example Z
```

where `<admin>` should be replaced with the username of the cylc installation account on your system. Type `cylc db print --help` to see what the command options mean.

2.5 Automated Database Test

The command `cylc admin test-db` gives suite registration database functionality a work out - it copies one of the cylc example suites, registers it under a new name and then manipulates it by recopying the suite in various ways, exporting it to the central database, and so on, before finally deleting the test registrations. This process should complete without error in a few seconds.

2.6 Automated Scheduler Test

The command `cylc admin test-suite` tests the cylc scheduler itself by running a suite temporarily registered under `testsuite`, configuring it to fail out a specific task, and then doing some advanced failure recovery intervention on it (recursive purge plus insertion of cold-start tasks). This process should complete in 2-3 minutes and can be watched in real time by right-clicking on the temporary test suite when it appears in the gcylc GUI and opening up a suite control GUI.

2.7 Complete Non-System-Level Installation

If you do not have root access to your host machine and cannot easily get Pyro, Graphviz, and Pygraphviz installed at system level, here's how to install everything under your home directory.

Cylc is already designed to be installed into a normal user account - just unpack the cylc release tarball into `$CYLC_DIR`. Now, if you try to use cylc commands now you will get a warning that Pyro is not installed.

Create a new sub-directory in the cylc source tree, `$CYLC_DIR/external`, and download the Pyro, Graphviz, and Pygraphviz source distributions to it (from the URLs given at the beginning of Section 2.1).

2.7.1 Pyro

Install Pyro under `$HOME/external/installed` as follows:

```
% cd $HOME/external
% tar xzf Pyro-3.14.tar.gz
% cd Pyro-3.14
% python setup.py install --prefix=$HOME/external/installed
```

Take note of the resulting Python `site-packages` directory under `external/installed/`, e.g.:

```
$HOME/external/installed/lib64/python2.6/site-packages/
```

The exact path will depend on your local Python environment. Configure your login script for the cylc environment as described in Section 2.3 above and add in this new installation path; for example:

```
# .profile
PYTHONPATH=$HOME/external/installed/lib64/python2.6/site-packages:$PYTHONPATH
PATH=$HOME/external/installed/bin:$PATH
```

Now you should be able to get cylc to print its release version:

```
% . $HOME/.profile   # (or log in again)
% cylc -v
x.y.z
```

If this command aborts and says that Pyro is not installed or is not available, then you have either not installed Pyro (check the output of the installation command carefully) *or* you have not pointed to the installed Pyro modules in your PYTHONPATH, *or* you have not sourced the cylc environment since updating PYTHONPATH.

2.7.2 Create The Central Suite Database

Now create the suite database and upload the example suites as described in Section 2.4. At this point you should have access to all cylc functionality except for suite graphing and the graph based suite control GUI.

2.7.3 Graphviz

Install Graphviz under `$CYLC_DIR/external/installed` as follows:

```
% cd $CYLC_DIR/external
% tar xzf graphviz-2.28.0.tar.gz
% cd graphviz-2.28.0
% ./configure --prefix=$CYLC_DIR/external/installed
% make
% make install
```

This installs graphviz files into the bin, include, and lib sub-directories of your local installation directory. The local installation section of `cylc.profile` (above) provides access to the graphviz executables in this bin directory, although you will probably not need to use them. The graphviz lib and include locations are required when installing Pygraphviz (next).

Note that graphviz may fail to build on a system that does not have QT installed. Lack of QT is not important for our purposes and you can disable it with `./configure --with-qt=no`.

2.7.4 Pygraphviz

Install Pygraphviz under `$CYLC_DIR/external/installed` as follows:

```
% cd $CYLC_DIR/external
% tar xzf pygraphviz-1.1.tar.gz
% cd pygraphviz-1.1
```

Now edit setup.py lines 31 and 32 to specify the graphviz lib and include directories:

```
library_path=os.environ['CYLC_DIR'] + '/external/installed/lib'
include_path=os.environ['CYLC_DIR'] + '/external/installed/include/graphviz'
```

Or you can just specify the absolute paths if you like, instead of using the `$CYLC_DIR` environment variable. Check that these are the correct library and include paths by inspecting the contents of the specified directories, and adjust them if necessary. Finally, install pygraphviz:

```
% export CYLC_DIR=/path/to/cylc
% python setup.py install --prefix=$CYLC_DIR/external/installed
```

This may or may not, depending on your local Python setup, install the Pygraphviz modules into the same place as the Pyro modules, e.g.:

```
% ls $CYLC_DIR/external/installed/lib64/python2.6/site-packages/
 pygraphviz  pygraphviz-1.1-py2.6.egg-info  Pyro  Pyro-3.14-py2.6.egg-info
```

If not, add the correct Pyraphviz installation path to your PYTHONPATH.

The easiest way to check that pygraphviz has been installed properly is to start an interactive Python session (type *python* after sourcing the cylc environment to configure your PYTHONPATH) then type *import pygraphviz* at the python interpreter prompt. If this results in the error message *ImportError: No module named pygraphviz* then either you have not installed pygraphviz properly, *or* you have not configured your PYTHONPATH to point to the installed pygraphviz modules, *or* you have not sourced the cylc environment since updating PYTHONPATH. Finally, if you have installed pygraphviz and configured your PYTHONPATH but graphviz itself has not been installed properly (or if the graphviz libraries have been deleted since you installed pygraphviz) then the initial pygraphiz import will be successful but a lower level import will fail when the pygraphviz modules cannot load the underlying graphviz libraries - in that case, reinstall graphviz.

2.7.5 What Next?

You should now have access to all cylc functionality. Create the central suite database and upload the example suites, if you have not done so already (Section 2.4), then test your cylc installation by running the automated suite database test (Section 2.5) and the automated scheduler test (Section 2.6), and go on to the *Quick Start Guide* (Section 7).

2.8 Upgrading To A New Cylc Version

Upgrading is just a matter of unpacking the new cylc release, copying the old central database into the new installation,[9] and running the new `cylc admin create-cdb` to load the new example suites. Finally, change the newly registered central suite group name to something sensible with `cylc db reregister` - it will have an ungainly string of numbers attached to ensure uniqueness.

3 Cylc Screenshots

- Figure 11 - the cylc command line interface.
- Figure 12 - viewing a private suite database in gcylc.
- Figure 13 - viewing the central suite database in gcylc.
- Figure 14 - a cylc suite definition file (suite.rc) in the vim editor.
- Figure 15 - the suite definition of Figure 14 graphed by cylc.
- Figure 16 - suite control GUI, treeview interface.
- Figure 17 - suite control GUI, graph-based interface.
- Figure 18 - a large operational suite graphed by cylc.
- Figure 19 - another view of the large operational suite of Figure 18.

[9] The central database is currently associated with a particular cylc installation - it is held in `$CYLC_DIR/CDB`.

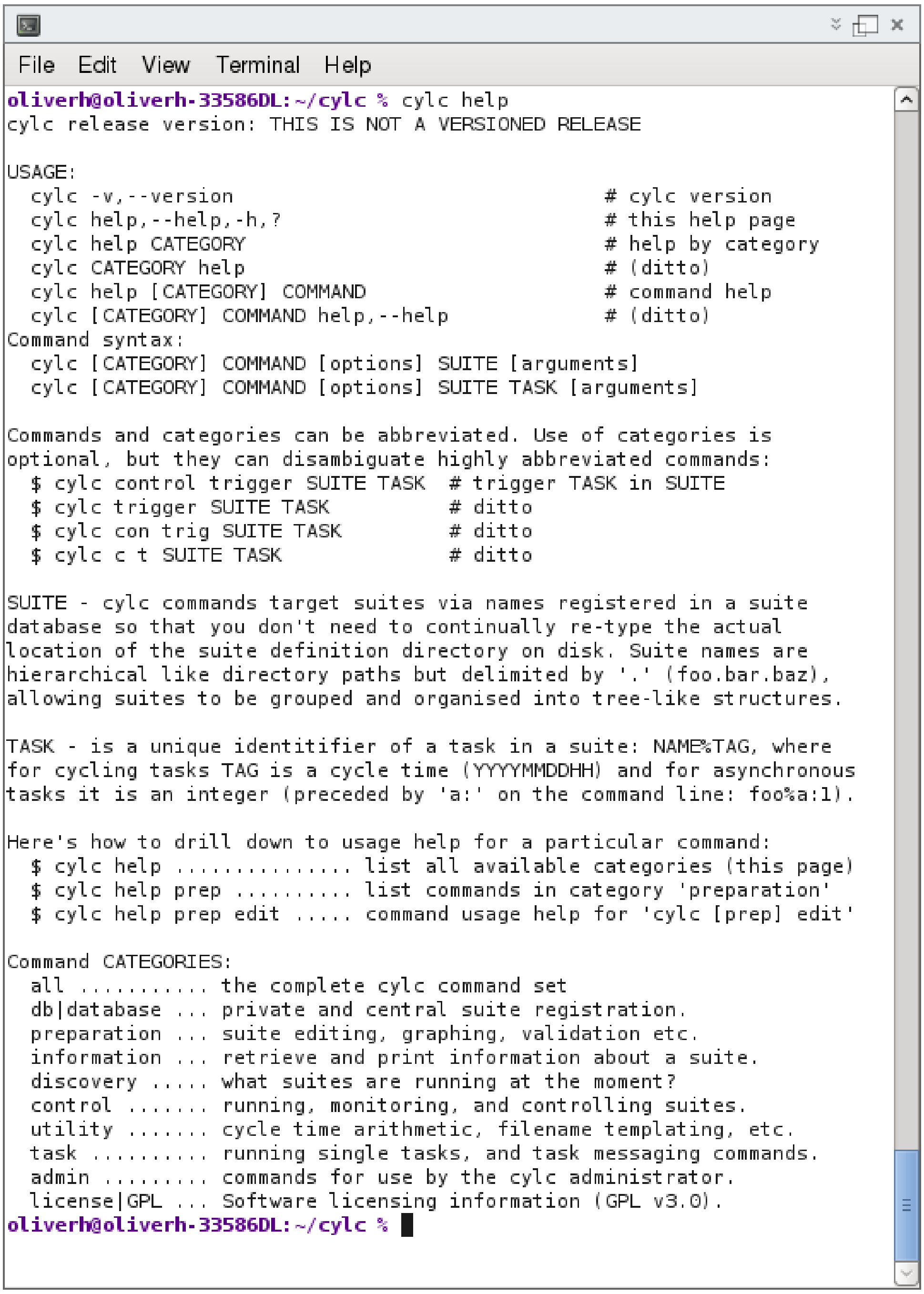

Figure 11: The cylc command line interface

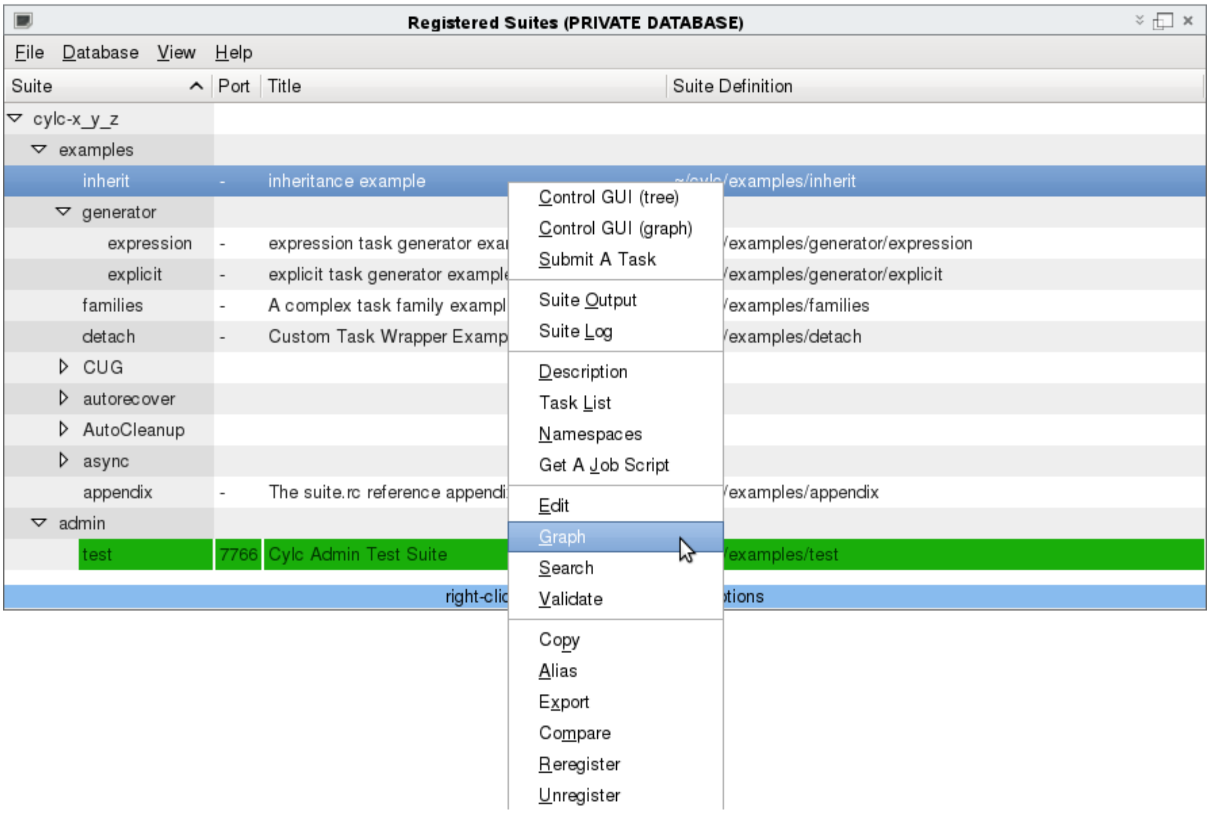

Figure 12: gcylc showing a private suite database, with one suite running on port 7766.

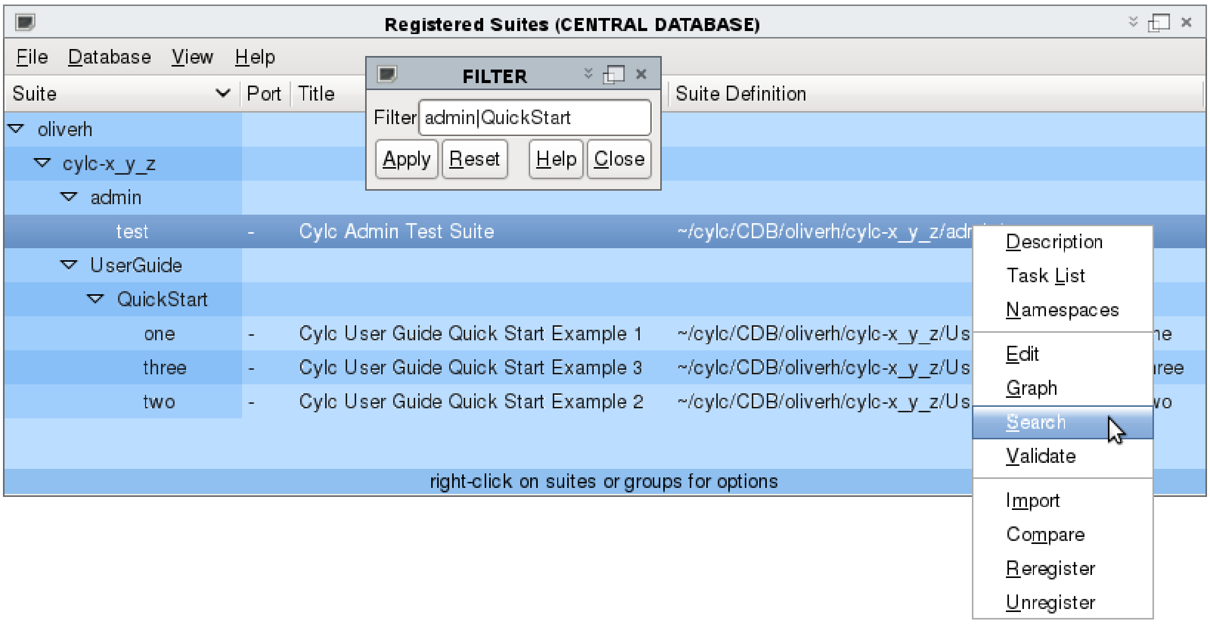

Figure 13: gcylc showing the central suite database.

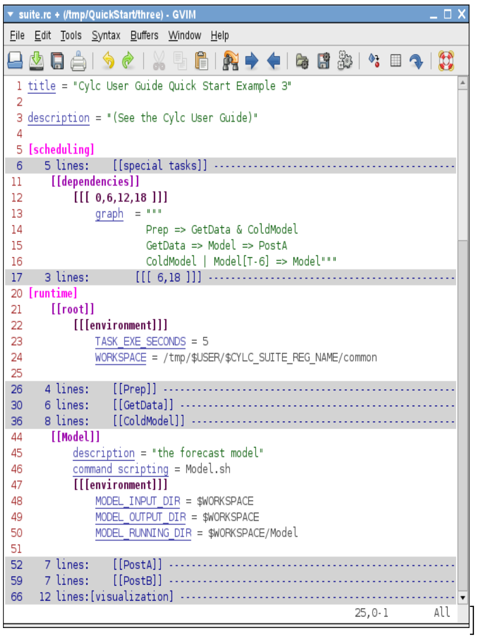

Figure 14: A simple cylc suite definition edited in *vim*

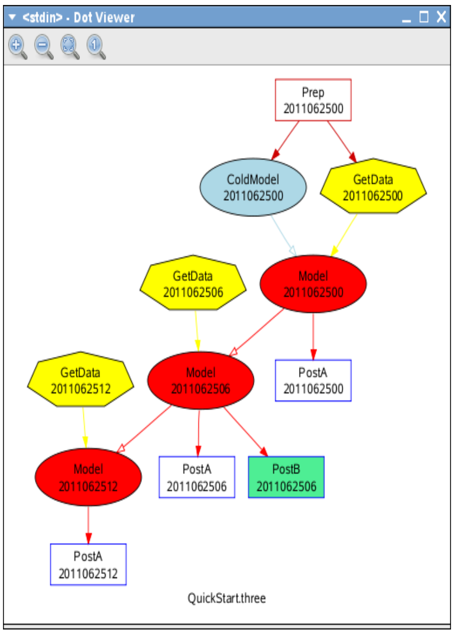

Figure 15: The suite definition of Figure 14 graphed by cylc.

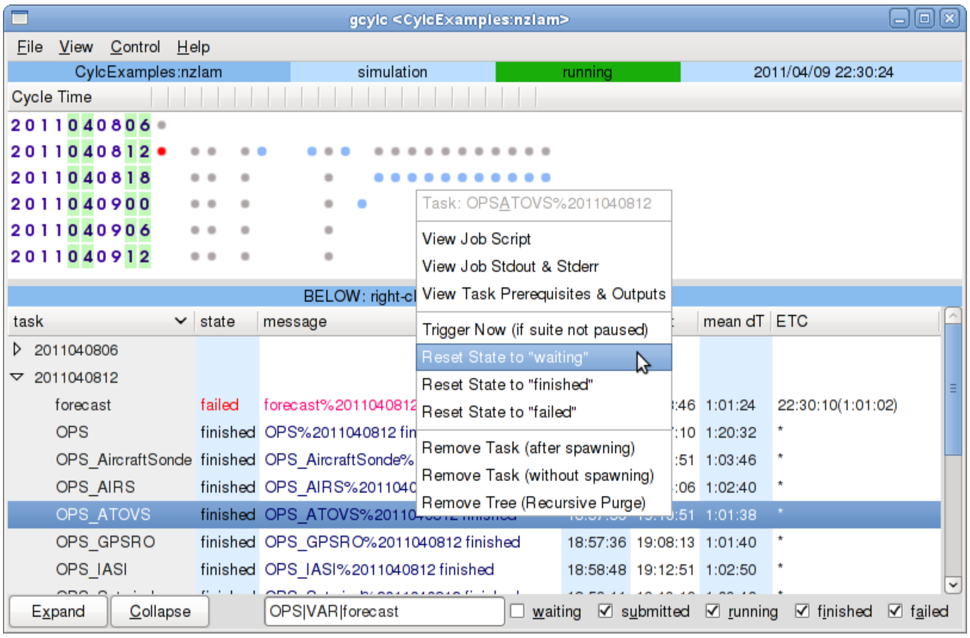

Figure 16: The suite control GUI, treeview interface.

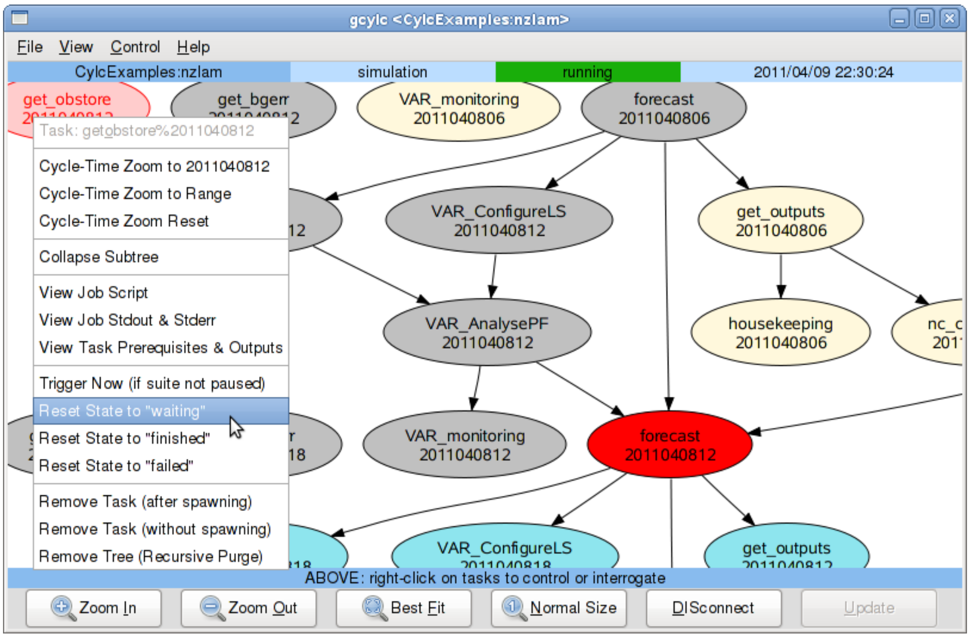

Figure 17: The suite control GUI, graph interface.

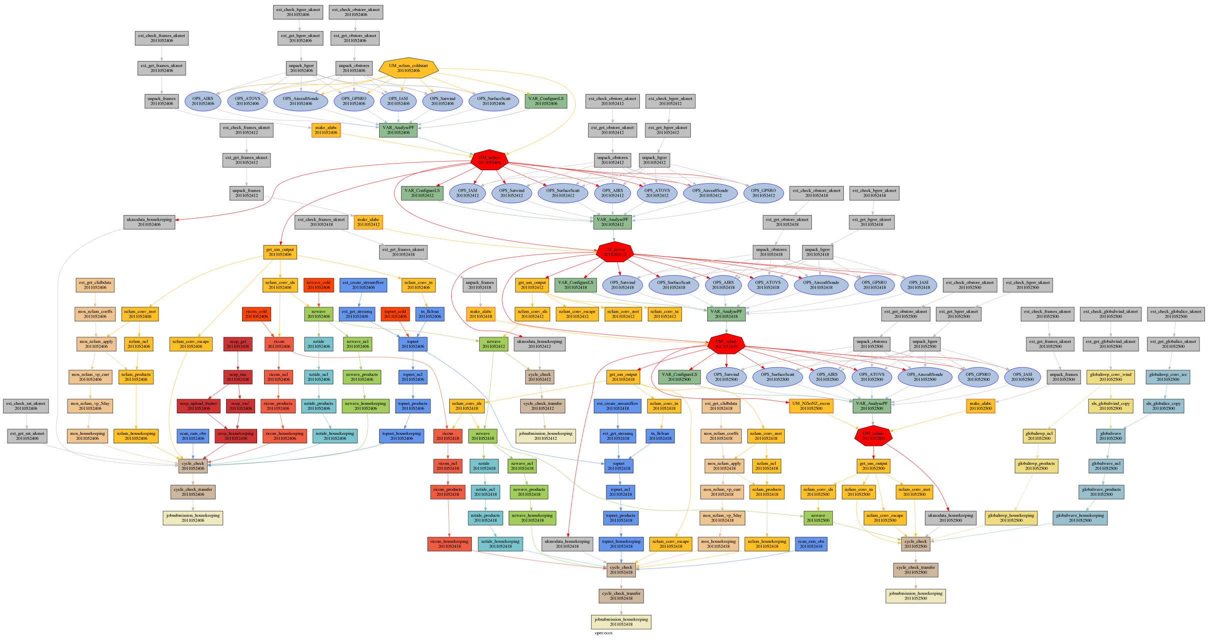

Figure 18: A large operational suite graphed by cylc.

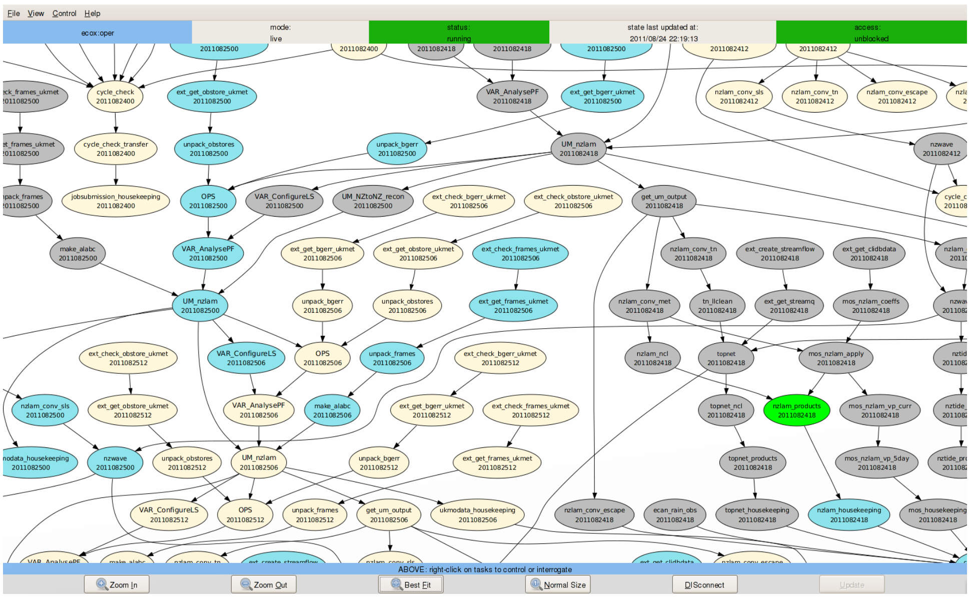

Figure 19: The large suite of Figure 18 running.

4 On The Meaning Of *Cycle Time* In Cylc

Cylc suites advance by means of individual tasks independently spawning successors at the next valid cycle time for the task, not by incrementing a suite-wide forecast cycle. In fact cylc has no concept of suite-wide cycle time; instead, each task has its own private cycle time and can run when its prerequisites are satisfied regardless of other tasks with different cycle times running at the same time - this underlies cylc's uniquely efficient and flexible scheduling abilities. However, it may sometimes still be convenient to refer to the "current cycle", the "previous cycle", or the "next cycle" and so forth, with reference to a particular task, or in the sense of all tasks that "belong to" a particular forecast cycle. Just keep in mind that the members of such groups may not all exist simultaneously in the running suite - tasks may pass through the "current cycle" (etc.) at different times as the suite evolves, particularly in delayed (catch up) operation.

5 Cylc's Example Suites

Cylc has been designed from the ground up to make prototyping and testing new suites very easy. Simply drawing (in text) a dependency graph in a new suite definition creates a valid suite that you can run (the tasks will be *dummy tasks* that default to emitting an identifying message, sleeping for a few seconds, and then exiting; but you can then arrange for particular tasks to do more complex things by configuring task runtime properties appropriately).

Cylc has quite a number of example suites intended to illustrate most facets of suite construction. These are held centrally under `$CYLC_DIR/examples` and are exported to the central suite database, accessible to all users, when cylc is installed. They all run "out the box" and can be copied and modified by users to test almost anything. Some of them just configure a suite dependency graph, in which case cylc will run dummy tasks according to the graph; some also configure task runtime properties (e.g. command scripting and environment variables) within the suite definition; and some have real task implementations that generate and consume real files and which will fail if they are not executed in the right order. All of the example suites use their registration names, portably via suite and task identity variable supplied by cylc (this is in fact the default situation) in all suite and task I/O paths, so you can run multiple copies of the same example suite at once, under different names, without changing anything in the suite definition.

5.1 Using The Example Suites

Use `gcylc` or `cylc db print --central <admin>` to see the example suites that are currently available. Note that here, and throughout the User Guide, any reference to an example suite in the central database will use `<admin>` as the username of the account under which cylc is installed (the suite owner's username is always the first component of a suite name in the central database); replace this as appropriate for your local cylc installation. Now use `cylc import` to copy an example suite (or some subset of them, or all of them at once) to your private database so that you can modify it and run it:

```
% cylc db print --central --tree <admin>.*\.QuickStart
# or just this:
% cylc db print --central --tree QuickStart
<admin>
 `-cylc-x:y:z
   `-QuickStart
     |-a   Quick Start Example A | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/a
     |-b   Quick Start Example B | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/b
     |-c   Quick Start Example C | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/c
     `-z   Quick Start Example Z | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/z
```



```
% cylc db import <admin>.cylc-x:y:z.QuickStart.z eg.z $TMPDIR
COPY /home/<admin>/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/z
  TO /tmp/oliver/eg/z
REGISTER eg.z: /tmp/oliver/eg/z

% cylc db print eg
eg.z | Quick Start Example Z | /tmp/oliver/eg/z

% cylc prep edit eg.z
```

```
title = "Quick Start Example Z"
description = "(Example A without the visualization config)"

[scheduling]
    runahead limit = 12
    initial cycle time = 2011010106
    final cycle time = 2011010200
    [[special tasks]]
        start-up        = Prep
        clock-triggered = GetData(1)
    [[dependencies]]
        [[[0,6,12,18]]]
            graph   = """Prep => GetData => Model => PostA
                        Model[T-6] => Model"""
        [[[6,18]]]
            graph = "Model => PostB"
```

(This suite is explained in the *Quick Start Guide*, Section 7).

5.2 Choosing The Initial Cycle Time

When running any suite in live mode (or at least suites that depend on clock-triggered tasks) do not give an initial cycle time in the future unless you want nothing to happen until that time. However, you can also run any suite in *simulation mode* (Section E) in which case a future start time is fine. In simulation mode each task is replaced by the default dummy task (above) and the suite runs quickly on an accelerated clock. As far as cylc is concerned this is almost identical to real operation, so simulation mode can be used to test recovery strategies for certain kinds of failure, for instance. In simulation mode, by configuring how the accelerated clock is initialized, you can watch how any suite catches up and transitions from delayed to real time operation.

6 Suite Registration

Before you can do anything to a suite with cylc commands you must register it in a *suite database*. Cylc commands target particular suites via their registered names so that you don't need to remember and continually re-type the actual location of the suite definition directory on disk. Suite names are hierarchical like directory paths but delimited by '.' (foo.bar.baz), allowing suites to be grouped and organised into trees (c.f. directory trees). Task groups are virtual and do not need to be explicitly created before use or removed if they contain no tasks.

6.1 Private Suite Databases

Each cylc user has a private suite database for working with his own suites. By right-clicking on a suite in your private database (or using cylc commands) you can:

1. start a suite control GUI to run the suite (or connect to a running suite),
2. submit a single task to run, just as it would be submitted by its suite
3. view the suite stdout and stderr streams,
4. view the suite log (which records all events and messages from tasks),

5. retrieve the suite description,
6. list tasks in the suite,
7. view the suite namespace hierarchy,
8. edit the suite definition in your editor,
9. plot the suite dependency graph,
10. search the suite definition and bin scripts,
11. validate the suite definition,
12. copy the suite or group,
13. alias the suite name to another name,
14. compare (difference) the suite with another suite,
15. export the suite or group to the central database,
16. unreregister the suite or group,
17. reregister the suite or group.

Note that the suite title shown in gcylc is parsed from the suite.rc file at the time of initial registration; if you change the title (by editing the suite.rc file) use `cylc db refresh` or gcylc View → Refresh to update the database.

6.2 The Central Suite Database

The central suite database facilitates sharing of suites among cylc users. It works similarly to a private suite database except that central suite registration names always begin with the suite owner's username. Users can export suites to the central database to make them available to others.

Suites in the central database can be inspected by various means, but you can't run them without *importing* them to your private database.

By right-clicking on a suite in the central database, or using the cylc command line, you can:

1. retrieve the suite description,
2. list tasks in the suite,
3. view the suite namespace hierarchy,
4. view the suite definition in your editor,
5. plot the suite dependency graph,
6. search the suite definition and bin scripts,
7. validate the suite definition,
8. compare (difference) the suite with another suite,
9. import the suite or group to your private database,
10. unreregister or delete the suite or group (if you own it),
11. reregister the suite or group (if you own it).

6.2.1 The Central Database Is Local To The Cylc Host

The central suite database is not currently accessible on the network, it is local to the cylc host and accessed through the filesystem. Consequently it has to be world, or at least group, writeable, which is not very secure.

We intend to develop a client/server interface to the central suite database so that users can easily share suites across the network. The required server functionality is similar to that of the cylc lockserver (and to cylc itself) so this will not be difficult. In the meantime, "importing" a suite manually from another user on another host is simply a matter of copying the suite definition directory over and registering the new copy in your private database.

6.3 Database Operations

On the command line, the 'database' (or 'db') command category contains commands to implement the aforementioned operations.

```
CATEGORY: db|database - private and central suite registration.

HELP: cylc [db|database] COMMAND help,--help
  You can abbreviate db|database and COMMAND.
  The category db|database may be omitted.

COMMANDS:
  alias ............... Register an alias for another registered suite
  copy|cp ............. Copy a suite or group of suites.
  export .............. Export private registrations to the central database
  gcylc ............... The cylc Graphical User Interface
  get-directory ....... Print a suite definition directory path
  import .............. Import central registrations to your private database
  print ............... Print private or central suite registrations
  refresh ............. Report invalid registrations and update suite titles
  register ............ Add a suite to your private registration database
  reregister|rename ... Change a suite's registered name.
  unregister .......... Remove suites from the private or central database
```

Groups of suites (at any level in the registration hierarchy) can be deleted, copied, imported, and exported; as well as individual suites. To do this, just use suite group names as source and/or target for operations, as appropriate. For instance, if a group `foo.bar` contains the suites `foo.bar.baz` and `foo.bar.waz`, you can copy a single suite like this:

```
% cylc copy foo.bar.baz boo $TMPDIR
```

(resulting in a new suite `boo`); or the group like this:

```
% cylc copy foo.bar boo $TMPDIR
```

(resulting in new suites `boo.baz` and `boo.waz`); or the group like this:

```
% cylc copy foo boo $TMPDIR
```

(resulting in new suites `boo.bar.baz` and `boo.bar.waz`). When suites are copied, the suite definition directories are copied into a directory tree, under the target directory, that reflects the registration name hierarchy. `cylc copy --help` has some explicit examples.

The same functionality is also available by right-clicking on suites or groups in the gcylc GUI, as shown in Figure 12.

7 Quick Start Guide

This section works through some basic cylc functionality using the "QuickStart" example suites, which should have been registered in the central suite database, under the cylc admin username (replace `<admin>` below with the correct username for your system) during cylc installation,

```
% cylc db print --central --tree QuickStart
<admin>
 `-cylc-x:y:z
   `-QuickStart
     |-a        Quick Start Example A | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/a
     |-b        Quick Start Example B | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/b
     |-c        Quick Start Example C | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/c
     `-z        Quick Start Example Z | ~/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/z
```

7.1 Configure Your Environment

To get access to cylc you just need to set a few environment variables in your login script, as described in Section 2.3.

7.2 Starting The gcylc GUI

At the command prompt:

```
% gcylc &
```

and use the Database menu to switch to the central database, which displays suites accessible to all users (see Figure 13).

7.3 Import The QuickStart Suites

You can register new suites directly in your private database, via gcylc or `cylc db register`, or you can import suites from the central database that is visible to all users. Suites in the central database can be view, validated, and graphed, etc., but you have to import them in order to run them. Section 6 explains exactly what actions can be performed on suites in central and private databases.

In gcylc, find the Quick Start suites (use View → Filter to filter for "QuickStart" if you like), right click on the *QuickStart* group and choose 'Import' to copy the whole group to your private database. In the dialog box that pops up, enter 'QuickStart' as the TARGET suite name, and `$TMPDIR` (or some other directory of your choice) as a destination for the suite definition directories in the group. Equivalently, on the command line:

```
% cylc db import <admin>.cylc-x:y:z.QuickStart Quickstart $TMPDIR
COPY /home/<admin>/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/a
  TO /tmp/oliver/QuickStart/a
REGISTER QuickStart.a: /tmp/oliver/QuickStart/a
COPY /home/<admin>/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/b
  TO /tmp/oliver/QuickStart/b
REGISTER QuickStart.b: /tmp/oliver/QuickStart/b
COPY /home/<admin>/cylc/CDB/<admin>/cylc-x:y:z/QuickStart/c
  TO /tmp/oliver/QuickStart/c
REGISTER QuickStart.c: /tmp/oliver/QuickStart/c
```

Note that you can also import individual suites, and that registration groups do not need to be created explicitly - they are automatically created and deleted as required.

Now switch gcylc back to your private database and confirm that you have a copy of the example suites; on the command line:

```
% cylc db pr -t Quick
QuickStart
 |-a       Quick Start Example A | /tmp/oliver/QuickStart/a
 |-b       Quick Start Example B | /tmp/oliver/QuickStart/b
 |-c       Quick Start Example C | /tmp/oliver/QuickStart/c
 `-z       Quick Start Example Z | /tmp/oliver/QuickStart/z
```

7.4 View The QuickStart.a Suite Definition

Cylc suites are defined by *suite.rc files*, discussed at length in *Suite Definition* (Section 8) and the *Suite.rc Reference* (Appendix A). To view the Quickstart.a suite definition right-click on the suite name and choose 'Edit'; or use the edit command:

```
% cylc edit QuickStart.a
```

This opens the suite definition in your editor (`$EDITOR`, or `$GEDITOR` from gcylc) *from the suite definition directory so that you can easily open other suite files in the editor*. You can of course do this manually, but by using the cylc interface you don't have to remember suite definition directory locations. For the rare occasions that you do need to move to a suite definition directory, you can do this:

```
% cd $( cylc db get-dir QuickStart.a )
```

Suites that use include-files can optionally be edited in a temporarily inlined state - the inline file will be split back into its constituent include-files when you save it and exit the editor.

Anyhow, you should now see the following suite.rc file in your editor:

```
title = "Quick Start Example A"
description = "(see the Cylc User Guide)"

[scheduling]
    runahead limit = 12
    initial cycle time = 2011010106
    final cycle time = 2011010200
    [[special tasks]]
        start-up        = Prep
        clock-triggered = GetData(1)
    [[dependencies]]
        [[[0,6,12,18]]]
            graph   = """Prep => GetData => Model => PostA
                        Model[T-6] => Model"""
        [[[6,18]]]
            graph = "Model => PostB"

[visualization] # optional
    [[node groups]]
        post = PostA, PostB
    [[node attributes]]
        post  = "style=unfilled", "color=blue", "shape=rectangle"
        PostB = "style=filled", "fillcolor=seagreen2"
        Model  = "style=filled", "fillcolor=red"
        GetData = "style=filled", "fillcolor=yellow3", "shape=septagon"
        Prep = "shape=box", "style=bold", "color=red3"
```

(Cylc comes with syntax highlighting and section folding for the *vim* editor - see Section 8.2.1.1).

This defines a complete, valid, runnable suite. Here's how to interpret it: At 0, 6, 12, and 18 hours each day a clock-triggered task called GetData triggers 1 hour after the wall clock reaches its (GetData's) nominal cycle time; then a task called Model triggers when GetData finishes; and a task called PostA triggers when Model is finished. Additionally, Model depends on its own previous instance from 6 hours earlier; and twice per day at 6 and 18 hours another task called PostB also triggers off Model.

All the tasks in this suite can run in parallel with their own previous instances if the opportunity arises (i.e. if their prerequisites are satisfied before the previous instance is finished). Most tasks should be capable of this (see Section 12.4) but if necessary you can force particular tasks to run sequentially like this:

```
# SUITE.RC
[scheduling]
    [[special tasks]]
        sequential = GetData, PostB
```

Finally, when the suite is *cold-started* (started from scratch) it is made to wait on a special *synchronous start-up task* called Prep. Start-up tasks are one-off (non-spawning) tasks that are only used at suite start-up, and any dependence on them only applies at suite start-up. Start-up tasks are *synchronous* because they have a defined cycle time even though they are not cycling tasks. Cylc also has *asynchronous one-off tasks*, which have no cycle time:

```
# SUITE.RC
```

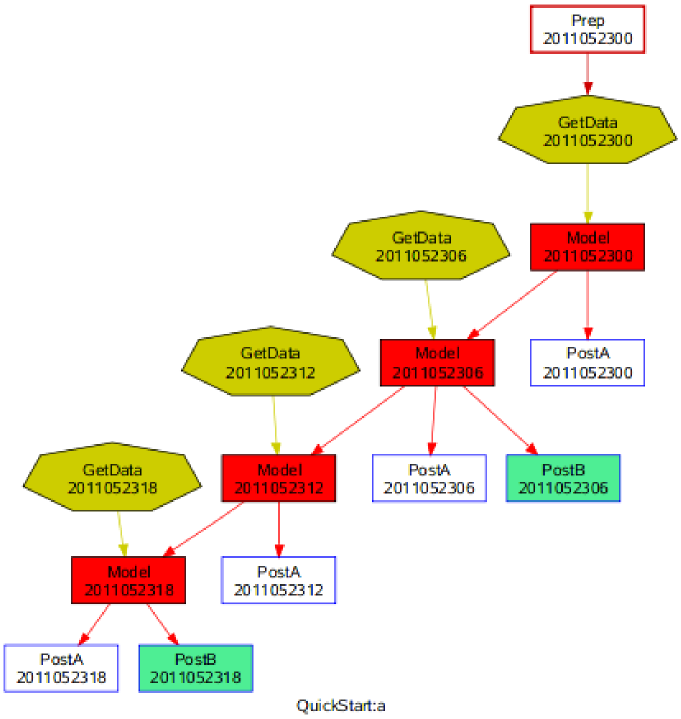

Figure 20: The *QuickStart.a* dependency graph, plotted by cylc.

```
[scheduling]
    [[dependencies]]
        graph = "prep"      # an asynchronous one-off task (no cycle time)
        [[[ 0,6,12,18 ]]]
            graph = "prep => foo => bar"    # followed by cycling tasks
```

The optional visualization section configures graph plotting.

7.5 Plotting The QuickStart.a Dependency Graph

Right-click on the *QuickStart.a* suite in gcylc and choose Graph; or by command line,

```
% cylc graph QuickStart.a 2011052300 2011052318 &
```

This will pop up a zoomable, pannable, graph viewer showing the graph of Figure 20. If you edit the suite.rc file the viewer will update in real time whenever you save the file.

7.6 Run The QuickStart.a Suite

Each cylc task defines command scripting to invoke the right external processing when the task is ready to run. This has not been explicilty configured in the example suite, so it defaults, for all tasks, to the *dummy task* scripting inherited from the *root namespace* (see Section 8):

```
% cylc get-config QuickStart.a runtime GetData 'command scripting'
['echo DUMMY $CYLC_TASK_ID; sleep 10; echo BYE']
```

where `$CYLC_TASK_ID` is the unique identifier by which cylc knows the task (`NAME%CYCLE`) exported to each task's execution environment by cylc. The command arguments reflect the suite definition section nesting. If it were explicitly configured in the example suite it would look like this:

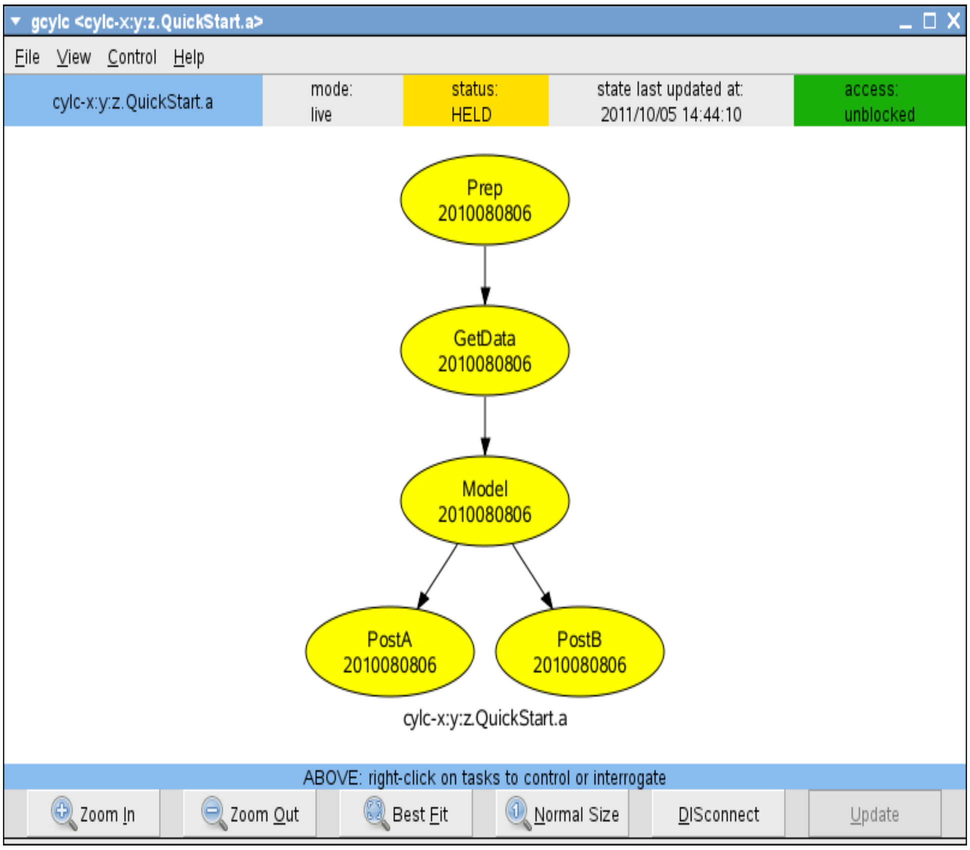

Figure 21: Suite *QuickStart:one* at start-up with an initial cycle time ending in 06 or 18 hours. Yellow nodes represent waiting tasks in the held state.

```
[runtime]
    [[GetData]]
        command scripting = "echo DUMMY $CYLC_TASK_ID; sleep 10; echo BYE"
```

Now start a suite control GUI by right-clicking on the suite in gcylc and choosing ‘Control (graph)’. You can also open a text treeview control GUI for the same suite, if you like. Multiple GUIs running at the same time will automatically connect to the same running suite (they won’t try to run separate instances). Note also that if you shut down a suite control GUI, the suite will keep running. You can reconnect to it later by opening another control GUI.

In the control GUI click on Control → Run, enter an initial cold-start cycle time (e.g. 2011052306), and select “Hold (pause) on start-up” so that the suite will start in the held state (tasks will not be submitted even if they are ready to run).

Do not choose an initial cycle time in the future unless you’re running in simulation mode, or nothing much will happen until that time.

If the initial cycle time ends in 06 or 18 the suite controller should look like Figure 21, or otherwise (00 or 12) like Figure 22.

The reason for the difference in graph structure between the two figures is this: cylc starts up with every task present in the waiting state (blue) at the initial cycle time *or* at the first subsequent valid cycle time for the task - and PostB does not run at 00 or 12. The off-white tasks are from the base graph, defined in the suite.rc file, and aren’t actually present in the suite as yet (they are shown in the graph in order to put the live tasks in context).

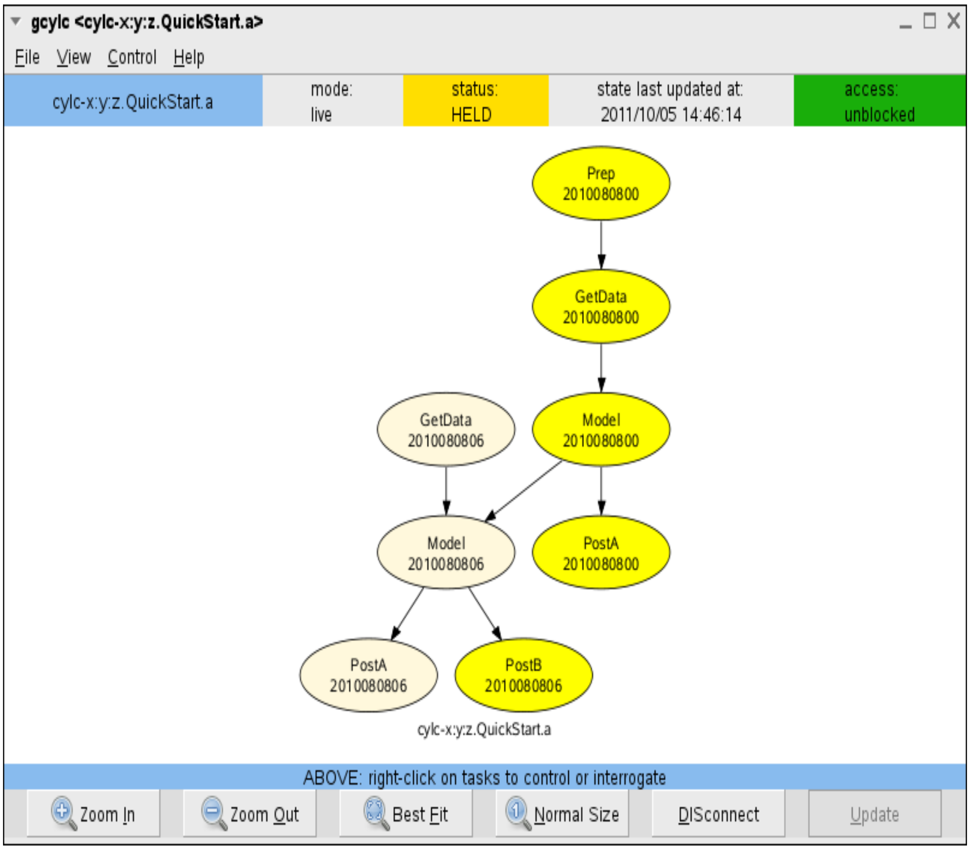

Figure 22: Suite *QuickStart.a* at start-up with an initial cycle time ending in 00 or 12 hours. Yellow nodes represent waiting tasks in the held state and off-white nodes are tasks from the base graph, defined in the suite.rc file, that aren't currently live in the suite.

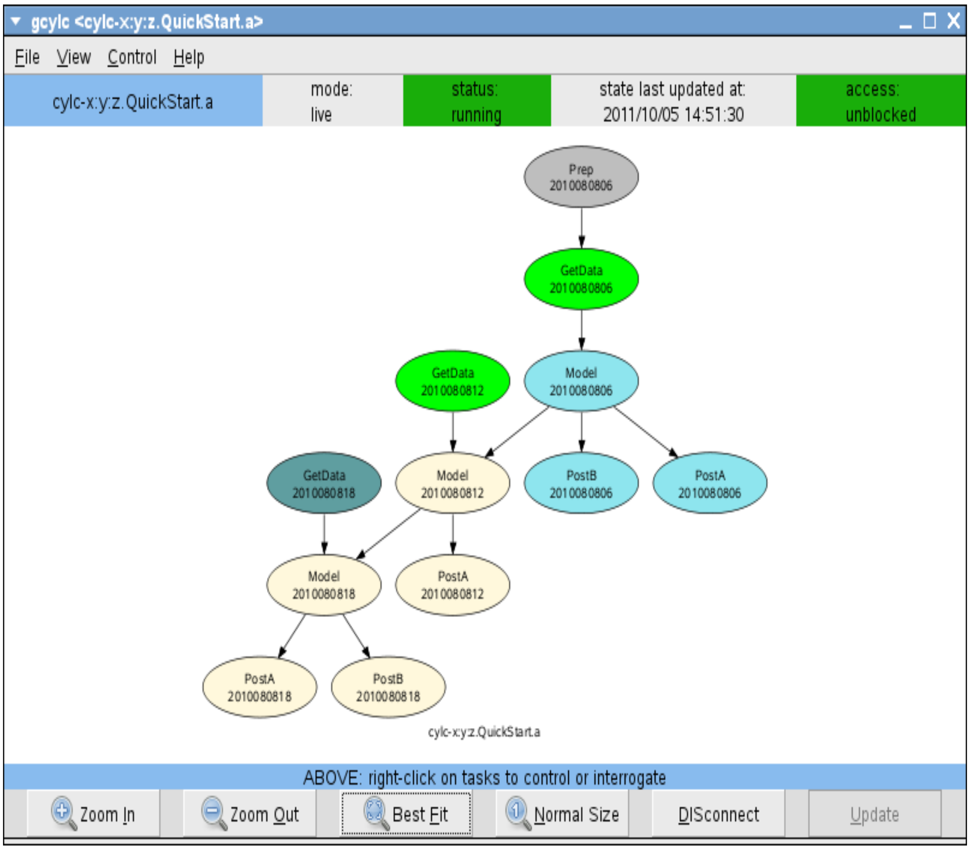

Figure 23: Suite *QuickStart.a* running, showing several consecutive instances of the clock-triggered GetData task running at once, out to the suite runahead limit of 12 hours.

Now, click on Control → Release in the suite control GUI to *release the hold on the suite*, and observe what happens: the GetData tasks will rapidly go off in parallel out to a few cycles ahead (how far ahead depends on the suite runahead limit, explained below) and then the suite will stall, as shown in Figures 23 and 24.

The Prep task runs immediately because it has no prerequisites and is not clock-triggered. The clock-triggered GetData tasks then all go off at once because they have no prerequisites (i.e. they do not have to wait on any upstream tasks), their trigger time has long passed (the initial cycle time was in the past), and they are not sequential tasks (so they are able to run in parallel - try declaring GetData sequential to see the difference). Beyond the suite *runahead limit* of 12 hours (set in the suite.rc file), however, GetData is put into a special 'runahead' held state indicated by the darker blue graph node. The task will be released from this state once the slower tasks in the suite have caught up sufficiently. The runahead limit is designed to stop free tasks like this from running off too far into the future in delayed operation. It is of little conseqence in real time operation[10] because clock triggered tasks are then constrained by the wall clock, and other tasks have to wait on them, generally speaking.

[10]So long as the runahead limit is sufficient to span the normal range of cycle times present in the suite - task that only run once per day, for example, have to spawn a successor that is 24 hours ahead.

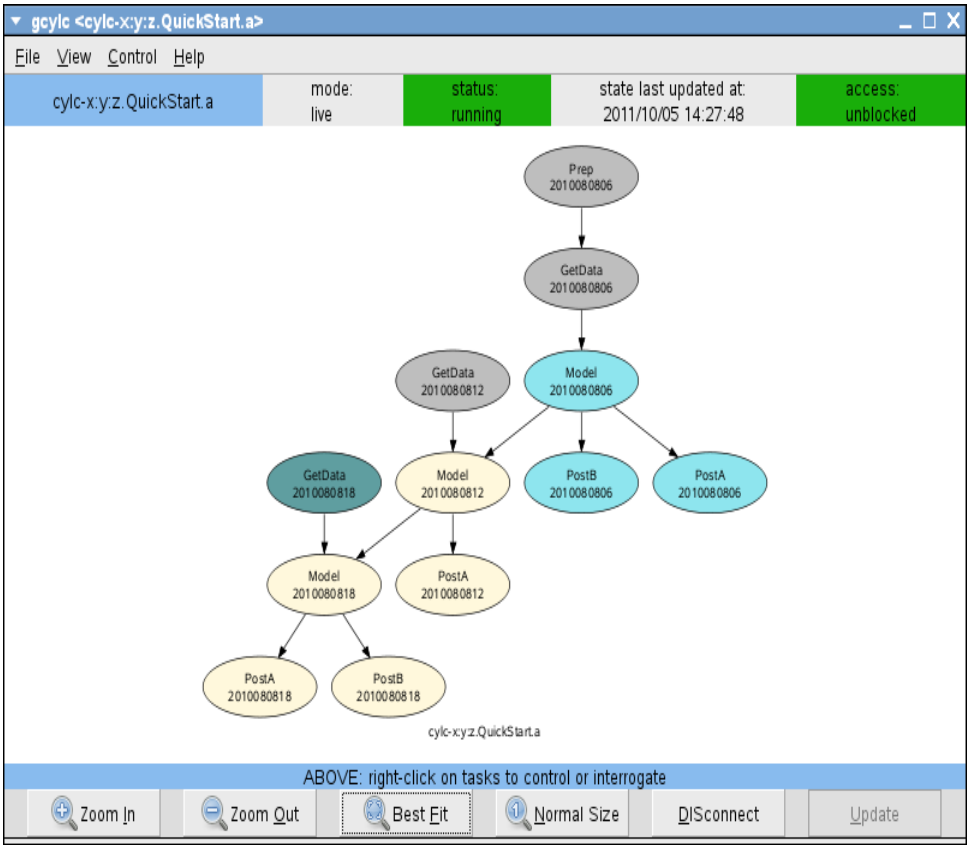

Figure 24: Suite *QuickStart.a* stalled after the clock-triggered GetData tasks have finished, because of Model's previous-cycle dependence and the suite runahead limit (see main text).

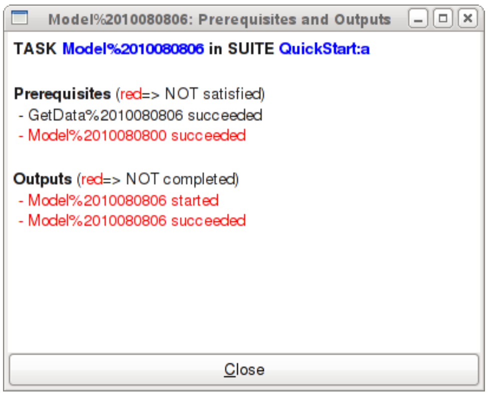

Figure 25: Viewing current task state after right-clicking on a task in gcylc. The same information is available from the `cylc show` command.

7.6.1 Viewing The State Of Tasks

If you're wondering why a particular task has not triggered yet in a running suite you can view the current state of its prerequisites by right-clicking on the task and choosing 'View State', or using `cylc show`. Do this for the first Model task, which appears to be stuck in the waiting state; it will pop up a small window as in Figure 25.

It is clear that the reason the task is not running, and consequently, by virtue of the runahead limit, why the suite has stalled, is that Model[T] is waiting on Model[T-6] which does not exist at suite start-up. Model represents a warm-cycled forecast model that depends on a model background state or restart file(s) generated by its own previous run.

7.6.2 Triggering Tasks Manually

Right-click on the waiting Model task and choose Trigger, or use `cylc trigger`, to force the task to trigger, and thereby get the suite up and running. In a real suite this would not be sufficient: the real forecast model that Model represents would fail for lack of the real restart files that it requires as input. Well see how to handle this properly shortly.

7.6.3 Suite Shut-Down And Restart

Having watched the *QuickStart.a* suite run for a while, choose Stop from the Control menu, or `cylc stop`, to shut it down. The default stop method waits for any tasks that are currently running to finish before shutting the suite down, so that the final recorded suite state is perfectly consistent with what actually happened.

You can restart the suite from where it left off by choosing Control → Run and selecting the 'restart' option, or using `cylc restart`. Note that cylc always writes a special state dump, and logs its name, prior to actioning any intervention, and you can also restart a suite from one of these states, rather than the default most recent state.

7.7 QuickStart.b - Handling Cold-Starts Properly

Now take a look at *QuickStart.b*, which is a minor modification of *QuickStart.a*. Its suite.rc file has a new *cold-start* task called ColdModel,

```
# SUITE.RC
[scheduling]
    [[special tasks]]
```

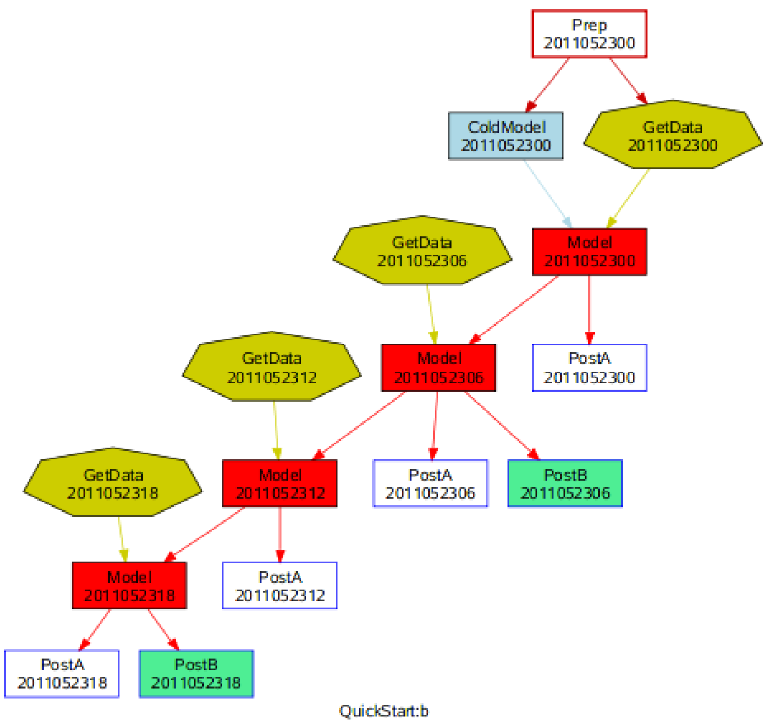

Figure 26: The *QuickStart.b* dependency graph showing a model cold start task.

```
        cold-start = ColdModel
```

and the dependency graph (see also Figure 26) looks like this:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        [[[ 0,6,12,18 ]]]
            graph   = """Prep => GetData & ColdModel
                        GetData => Model => PostA
                        ColdModel | Model[T-6] => Model"""
        [[[ 6,18 ]]]
            graph = "Model => PostB"
```

In other words, Model[T] can trigger off *either* Model[T-6] *or* ColdModel[T].

A cold-start task is a one-off task used to satisfy the previous-cycle dependence of a cotemporal task whose previous-cycle trigger is not available. The obvious use for this is to cold-start warm-cycled forecast models at suite start-up, when there is no previous cycle. Unlike start-up tasks though, cold-start dependencies are not restricted to suite start-up because it is sometimes useful to be able to insert a cold-start task into a running suite, to get a model restarted after it had to skip one or more cycles due to problems, without having to restart the whole suite.

A model cold-start task in a real suite may submit a real "cold start forecast" to generate the previous-cycle input files required by its associated model, or it may just stand in for some external spinup process, or similar, that has to be completed before the suite is started (in the latter case the cold-start task would be a dummy task that just reports successful completion in order to satisfy the initial previous-cycle dependence of the model).

Run *QuickStart.b* to confirm that that no manual triggering is required to get the suite started now.

7.8 QuickStart.c - Real Task Implementations

The suite *QuickStart.c* is the same as *QuickStart.b* except that it has real task implementations (scripts located in the suite bin directory) that generate and consume files in such a way that they have to run according to the graph of Figure 26. The suite gets them to run together out of a common I/O workspace, configured via the suite.rc file.

By studying this suite and its tasks, and by making quick copies of it to modify and run, you should be able to learn a lot about how to build real cylc suites. Here's the complete suite definition

```
title = "Quick Start Example C"
description  = "(Quick Start b plus real tasks)"

# A clock-triggered data-gathering task, a warm-cycled model, and two
# post-processing tasks (one runs every second cycle). The tasks are not
# cylc-aware, have independently configured I/O directories, and abort
# if their input files do not exist. This suite gets them all to run out
# of a common I/O workspace (although the warm-cycled model uses a
# private running directory for its restart files).

[scheduling]
    runahead limit = 12
    initial cycle time = 2011010106
    final cycle time = 2011010200
    [[special tasks]]
        start-up        = Prep
        cold-start      = ColdModel
        clock-triggered = GetData(1)
    [[dependencies]]
        [[[0,6,12,18]]]
            graph  = """Prep => GetData & ColdModel
                        GetData => Model => PostA
                        ColdModel | Model[T-6] => Model"""
        [[[6,18]]]
            graph = "Model => PostB"

[runtime]
    [[root]]
        [[[environment]]]
            TASK_EXE_SECONDS = 5
            WORKSPACE = /tmp/$USER/$CYLC_SUITE_REG_NAME/common

    [[Prep]]
        description = "prepare the suite workspace for a new run"
        command scripting = clean-workspace.sh $WORKSPACE

    [[GetData]]
        description = "retrieve data for the current cycle time"
        command scripting = GetData.sh
        [[[environment]]]
            GETDATA_OUTPUT_DIR = $WORKSPACE

    [[Models]]
        [[[environment]]]
            MODEL_INPUT_DIR = $WORKSPACE
            MODEL_OUTPUT_DIR = $WORKSPACE
            MODEL_RUNNING_DIR = $WORKSPACE/Model
    [[ColdModel]]
        inherit = Models
        description = "cold start the forecast model"
        command scripting = Model.sh --coldstart
    [[Model]]
        inherit = Models
        description = "the forecast model"
        command scripting = Model.sh

    [[Post]]
        description = "post processing for model"
```

```
        [[[environment]]]
            INPUT_DIR   = $WORKSPACE
            OUTPUT_DIR = $WORKSPACE
    [[PostA,PostB]]
        inherit = Post
        command scripting = $(TASK).sh

[visualization]
    default node attributes = "shape=ellipse"
    [[node groups]]
        post = PostA, PostB
        models = ColdModel, Model
    [[node attributes]]
        post   = "style=unfilled", "color=blue", "shape=rectangle"
        PostB = "style=filled", "fillcolor=seagreen2"
        models   = "style=filled", "fillcolor=red"
        ColdModel = "fillcolor=lightblue"
        GetData = "style=filled", "fillcolor=yellow", "shape=septagon"
        Prep = "shape=box", "style=bold", "color=red3"
```

Here's the namespace hierarchy defined by this suite:

```
% cylc list --tree QuickStart.c
root
 |-GetData     retrieve data for the current cycle time
 |-Models
 | |-ColdModel cold start the forecast model
 | `-Model     the forecast model
 |-Post
 | |-PostA     post processing for model
 | `-PostB     post processing for model
 `-Prep        prepare the suite workspace for a new run
```

And here, for example, is the complete implementation for the PostA task (located with the other task scripts in the suite bin directory):

```
#!/bin/bash

set -e

cylc checkvars  TASK_EXE_SECONDS
cylc checkvars -d INPUT_DIR
cylc checkvars -c OUTPUT_DIR

# CHECK INPUT FILES EXIST
PRE=$INPUT_DIR/surface-winds-${CYLC_TASK_CYCLE_TIME}.nc
if [[ ! -f $PRE ]]; then
    echo "ERROR, file not found $PRE" >&2
    exit 1
fi

echo "Hello from $CYLC_TASK_NAME at $CYLC_TASK_CYCLE_TIME in $CYLC_SUITE_REG_NAME"

sleep $TASK_EXE_SECONDS

# generate outputs
touch $OUTPUT_DIR/surface-wind.products
```

7.9 Monitoring Running Suites

7.9.1 Suite stdout And stderr

When cylc runs a suite it writes warnings and other informative messages, such as when and how each task is submitted, to the stdout stream. If you start a suite at the command line you can direct this output wherever you like. If you start a suite via gcylc, however, the output is directed to a special file, `$HOME/.cylc/[SUITE].out` that can be accessed again later if you reconnect to the suite with a new control GUI.

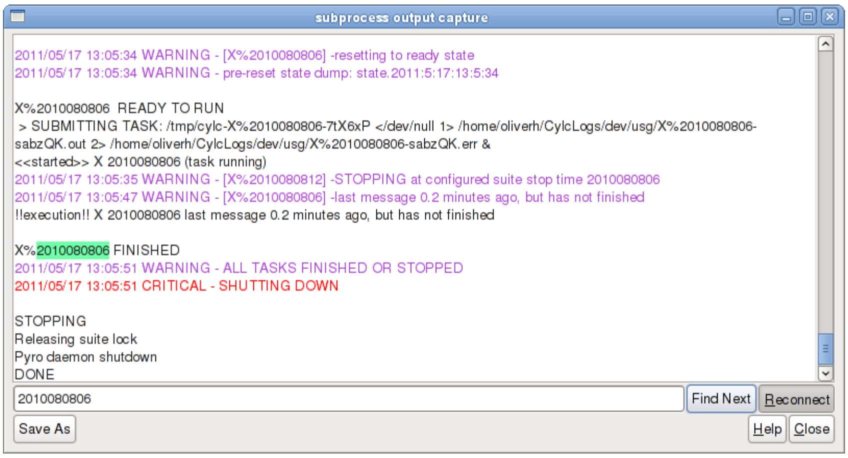

Figure 27: Cylc suite stdout/stderr example.

7.9.2 Suite Logs

The cylc suite log records every event that occurs (incoming messages from tasks and so on) along with the time of the event. The top level logging directory, under which a suite-specific log is written, is configurable in the suite.rc file. Suite logs are (optionally) rolled over at start-up and the five most recent back ups are kept.

Figure 28 shows a suite log viewed from within gcylc. The `cylc log` command also enables viewing and filtering of suite logs without having to remember the actual log file location. But if you want to know the location:

```
% cylc get-config --directories QuickStart.a
SUITE LOG DIRECTORY:
  /home/oliverh/cylc-run/QuickStart.a/log/suite
SUITE STATE DUMP DIRECTORY:
  /home/oliverh/cylc-run/QuickStart.a/state
JOB SUBMISSION LOG DIRECTORIES:
  + root: /home/oliverh/cylc-run/QuickStart.a/log/job
```

(There would be other job submission log directories listed here if any of the tasks in the suite had overridden the default location set by the root namespace).

7.9.3 Task stdout And stderr Logs

The stdout and stderr logs generated when a task is submitted end up in the suite.rc *job submission log directory*, default location `$HOME/cylc-run/$CYLC_SUITE_REG_NAME/log/job/` (where the variable `$CYLC_SUITE_REG_NAME` evalutes to the registered suite name hierarchy, e.g. foo.bar.baz):

```
% get-config QuickStart.a runtime GetData 'job submission' 'log directory'
/home/oliverh/cylc-run/QuickStart.a/log/job
```

(or use the `--directories` argument as shown just above).

These files will contain the complete stdout and stderr record for tasks whose initiating processes do not detach and exit before all task processing is finished. The location of output generated by secondary processes, if the original process detaches and exits early, is up to the task implementation (of said secondary processes, specifically).

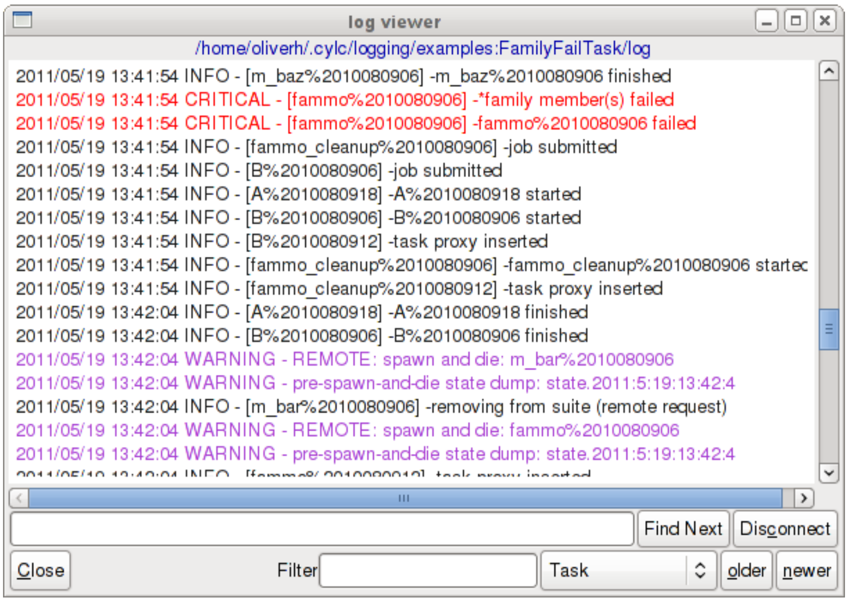

Figure 28: A cylc suite log viewed via gcylc.

7.10 Searching A Suite

The cylc suite search tool reports matches in the suite.rc file by line number, suite section, and file, even if include-files are used (and even if they are nested), and by file and line number for matches in the suite bin directory. The following output listing is from a search of the *QuickStart.c* suite.

```
% cylc grep OUTPUT_DIR QuickStart.c
```

```
SUITE: QuickStart.c /tmp/oliverh/QuickStart/c/suite.rc
PATTERN: OUTPUT_DIR

FILE: /tmp/oliverh/QuickStart/c/suite.rc
   SECTION: [runtime]->[[GetData]]->[[[environment]]]
      (40):            GETDATA_OUTPUT_DIR = $WORKSPACE
   SECTION: [runtime]->[[Models]]->[[[environment]]]
      (45):            MODEL_OUTPUT_DIR = $WORKSPACE
   SECTION: [runtime]->[[Post]]->[[[environment]]]
      (60):            OUTPUT_DIR = $WORKSPACE

FILE: /tmp/oliverh/QuickStart/c/bin/PostB.sh
   (7): cylc checkvars -c OUTPUT_DIR
   (21): touch $OUTPUT_DIR/precip.products

FILE: /tmp/oliverh/QuickStart/c/bin/Model.sh
   (11): cylc checkvars -c MODEL_OUTPUT_DIR MODEL_RUNNING_DIR
   (54): touch $MODEL_OUTPUT_DIR/surface-winds-${CYLC_TASK_CYCLE_TIME}.nc
   (55): touch $MODEL_OUTPUT_DIR/precipitation-${CYLC_TASK_CYCLE_TIME}.nc

FILE: /tmp/oliverh/QuickStart/c/bin/PostA.sh
   (7): cylc checkvars -c OUTPUT_DIR
   (21): touch $OUTPUT_DIR/surface-wind.products

FILE: /tmp/oliverh/QuickStart/c/bin/GetData.sh
   (6): cylc checkvars -c GETDATA_OUTPUT_DIR
   (11): touch $GETDATA_OUTPUT_DIR/obs-${CYLC_TASK_CYCLE_TIME}.nc
```

(Suite search is also available from the gcylc right-click menu).

7.11 Comparing Suites

Cylc can also compare suites and report differences by suite.rc section and item. For instance, comparing the example suites `QuickStart.a` and `QuickStart.b` by GUI or command line results in:

```
% cylc diff QuickStart.a QuickStart.b
```

```
Parsing QuickStart.a
Parsing QuickStart.b
Suite definitions QuickStart.a and QuickStart.b differ.
!SNIP!
13 common items differ QuickStart.a(<) QuickStart.b(>)
   (top)
 <   description = (see the Cylc User Guide)
 >   description = (Quick Start a plus a cold-start task)
 <   title = Quick Start Example A
 >   title = Quick Start Example B

   [cylc] [[logging]]
 <   directory = /home/oliverh/cylc-run/QuickStart.a/log/suite
 >   directory = /home/oliverh/cylc-run/QuickStart.b/log/suite

   [scheduling] [[dependencies]] [[[0,6,12,18]]]
 <   graph = Prep => GetData => Model => PostA
                       Model(T-6) => Model
 >   graph = Prep => GetData & ColdModel
                       GetData => Model => PostA
                       ColdModel | Model(T-6) => Model
!SNIP!
```

(much of the diff output has been omitted here for the sake of brevity). Note that suite log directories and the like may differ even though they are not explicitly configured in either suite, because their default values (see the *Suite.rc Reference*, Appendix A) are suite-registration-specific.

7.12 Validating A Suite

Suite validation checks for errors by parsing the suite definition, comparing all items against the suite.rc specification file, and then parsing the suite graph and attempting to instantiate all task proxy objects. This can be done using gcylc or `cylc validate`:

```
% cylc validate -v foo.bar
Parsing Suite Definition
LOADING suite.rc
VALIDATING against the suite.rc specification.
PARSING clock-triggered tasks
PARSING runtime generator expressions
PARSING runtime hierarchies
PARSING SUITE GRAPH
Instantiating Task Proxies:
root
 |-GEN
 | |-OPS
 | | |-aircraft    ... OK
 | | |-atovs       ... OK
 | | `-atovs_post  ... OK
 | `-VAR
 |   |-AnPF        ... OK
 |   `-ConLS       ... OK
 |-baz
 | |-bar1          ... OK
 | `-bar2          ... OK
 |-foo             ... OK
 `-prepobs         ... OK
Suite foo.bar validates OK.
```

For more information on suite validation see Section 8.2.3.

8 Suite Definition

A cylc suite is defined entirely by a single structured, validated, configuration file called *suite.rc* that concisely specifies the properties of, and the relationships between, the various tasks managed by the suite. This section of the User Guide deals with the format and content of the suite.rc file, including task definition. Task implementation - what's required of the real commands, scripts, or programs that do the processing that the tasks represent - is covered in Section 9; and task job submission - how tasks are submitted to run - is in Section 10.

8.1 Suite Definition Directories

A cylc *suite definition directory* contains:

- **A suite.rc file**: this is *the* suite definition.
  - And any include-files used in it (see below; may be kept in sub-directories).
- **A suite `bin/` directory**.
  - For scripts and executables that implement, or are used by, suite tasks.
  - Automatically added to `$PATH` in task execution environments.
  - Technically optional as tasks can call external commands, scripts, or programs; or they can be scripted entirely within the suite.rc file.
- **Any other sub-directories and files** - documentation, control files, etc.
  - The entire directory tree is copied if you copy the suite with cylc or gcylc.
  - Run time task access via `$CYLC_SUITE_DEF_PATH` (Section 8.4.4).
  - Holding everything in one place makes proper suite revision control possible.

An imaginary example:

```
/path/to/my/suite   # suite definition directory
    suite.rc           # THE SUITE CONFIGURATION FILE
    bin/               # scripts and executables used by tasks
        foo.sh
        bar.sh
        ...
    # (OPTIONAL) any other suite-related files, for example:
    inc/               # suite.rc include-files
        nwp-tasks.rc
        globals.rc
        ...
    doc/               # documentation
    control/           # control files
    ancil/             # ancillary files
    ...
```

8.2 Suite.rc Overview

Cylc suite.rc files are parsed directly into a nested data structure that mirrors the file section nesting. This makes it very easy to add new configuration items to cylc. The underlying file structure is based on ConfigObj (http://www.voidspace.org.uk/python/configobj.html), slightly modified for cylc.

8.2.1 Syntax

- **All entries** are of the form `item = value` (some values require post-validation parsing).
- **Comments** follow a hash character (#) to the end of the line.
- **List Values** are comma separated.
- **Strings** must be quoted if they contain commas (which indicate list values).
- **Multiline Strings** must be triple-quoted.
- **Boolean Values** are written as True or False (capitalized).
- **White Space** is ignored; indentation can be used for clarity.
- **Continuation Lines** follow a trailing backslash.
- [**Section Headings**] are enclosed in square brackets.
- [[**Subsection Nesting**]] is indicated by the number of square brackets.[11]
- **Duplicate Items** are illegal, except in `environment` and `directives` sections.[12]
- **Include-files** can be used, and may be multiply-included and nested. Inclusion boundaries are arbitrary (they can span sections). Include-file paths should be specified portably relative to the suite definition directory, e.g.:

```
# include the file $CYLC_SUITE_DEF_PATH/inc/foo.rc:
%include inc/foo.rc
```

The following pseudo-listing illustrates suite.rc syntax:

```
# a full line comment
an item = value # a trailing comment
a boolean item = True # or False
one string item = the quick brown fox # string quotes optional ...
two string item = "the quick, brown fox" # ... unless internal commas
a multiline string item = """the quick brown fox
jumped over the lazy dog""" # triple quoted
a list item = foo, bar, baz   # comma separated
a list item with continuation = a, b, c, \
                                d, e, f
[section]
    item = value
%include inc/vars/foo.inc  # include file
    [[subsection]]
        item = value
        [[[subsubsection]]]
            item = value
[another section]
    [[another subsection]]
        # ...
    # ...
# ...
```

8.2.1.1 Syntax Highlighting In Vim

Cylc comes with a syntax file to configure suite.rc syntax highlighting and section folding in the *vim* editor, as shown in Figure 14. To use this, copy `$CYLC_DIR/conf/cylc.vim` to your `$HOME/.vim/syntax/` directory and make some minor modifications, as described in the syntax file, to your `$HOME/.vimrc` file.

[11] Sections are closed by the next section heading, so items within a section must be defined before any subsequent subsection headings.

[12] The exceptions were designed to allow tasks to override environment variables defined in include-files that could be included in multiple tasks, to assist in factoring out common task configuration. However, namespace inheritance now provides a better way to do this in most cases.

8.2.2 Gross File Structure

Cylc suite.rc files consist of a suite title and description followed by configuration items grouped under several top level section headings:

- **[cylc]** - *non task-related suite configuration*
  - suite logging directories, simulation mode, UTC mode, etc.
- **[scheduling]** - *determines when tasks are ready to run*
  - tasks with special behaviour, e.g. clock-triggered tasks
  - the dependency graph, defines relationships between tasks
- **[runtime]** - *determines how, where, and what to execute when tasks are ready*
  - command scripting, environment, job submission, remote hosting, etc.
  - suite-wide defaults in the *root* namespace
  - a nested family hierarchy with common properties inherited by related tasks
- **[visualization]** - suite graphing and the graph-based control GUI.

8.2.3 Validation

Cylc suite.rc files are automatically validated against a specification file that defines all legal entries, values, options, and defaults (`$CYLC_DIR/conf/suiterc.spec`). This detects any formatting errors, typographic errors, illegal items and illegal values prior to run time. Some values are complex strings that require further parsing by cylc to determine their correctness (this is also done during validation). All legal entries are documented in the *Suite.rc Reference* (Appendix A).

The validator reports the line numbers of detected errors. Here's an example showing a subsection heading with a missing right bracket.

```
% cylc validate foo.bar
Parsing Suite Config File
ERROR: [[special tasks]
NestingError('Cannot compute the section depth at line 19.',)
_validate foo.bar failed:  1
```

If the suite.rc file contains include-files use `cylc inline` (or the gcylc right-click 'Edit' inline option) to view an inlined copy with correct line numbers (you can also edit suites in a temporarily inlined state using `cylc edit --inline`).

8.2.4 String Interpolation

Configuration items beginning with an underscore are ignored by cylc, but may be used with the string interpolation mechanism provided by the ConfigObj file format. An example:

```
# SUITE.RC
_greeting = HELLO
[runtime]
    [[root]]
        command scripting = "echo %(_greeting)s from $CYLC_TASK_ID; sleep 10; echo BYE"
```

Then, assuming this suite is registered as `foo`,

```
% cylc get-config foo runtime root 'command scripting'
['echo HELLO from $CYLC_TASK_ID; sleep 10; echo BYE']
```

```
# SUITE.RC
title = "Dependency Graph Example"
[scheduling]
    [[dependencies]]
        [[[0,6,12,18]]] # validity (hours)
            graph = """
A => B & C    # B and C trigger off A
A[T-6] => A   # Model A restart trigger
                    """
        [[[6,18]]] # hours
            graph = "C => X"
[visualization]
    [[node attributes]]
        X = "color=red"
```

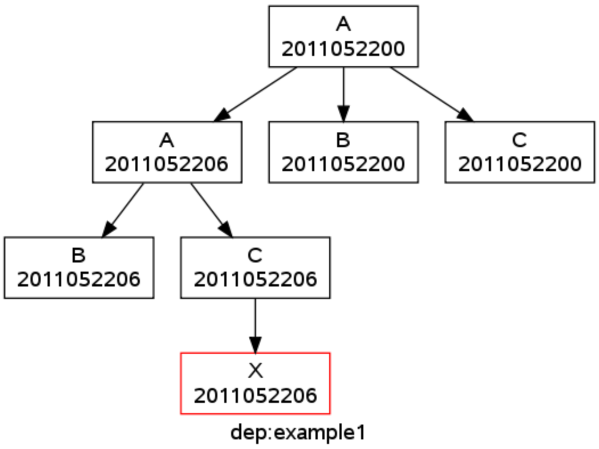

Figure 29: Example Suite SDepG

8.3 Scheduling - Dependency Graphs

The `[scheduling]` section of a suite.rc file defines the relationships between tasks in a suite - the information that allows cylc to determine when tasks are ready to run. The most important component of this is the suite dependency graph. Cylc graph notation makes clear textual graph representations that are more concise than the real thing because sections of the graph that repeat at different hours of the day only have to be defined once. Here's an example showing a simple task tree with dependencies that vary at certain hours:

```
[scheduling]
    [[dependencies]]
        [[[0,6,12,18]]] # validity (hours of the day)
            graph = """
A => B & C    # B and C trigger off A
A[T-6] => A   # Model A restart trigger
                    """
        [[[6,18]]] # hours
            graph = "C => X"
```

Figure 29 shows the complete suite.rc listing alongside the suite graph, plotted with,

```
% cylc graph SDepG 2011052200 2011052206
# (or use right-click Graph in gcylc)
```

This is actually a complete, valid, runnable suite (it will use default runtime properties such as dummy command scripting, because no runtime properties are explicitly defined, and you'll need to trigger task A manually to get the suite started because A[T] depends on A[T-6] and at start-up there is no previous cycle to satisfy that dependence - how to handle this properly is described in *Satisfying Intercycle Dependencies At Start-Up* (Section 8.3.5) and in the *Quick Start Guide* (Section 7).

8.3.1 Graph String Syntax

Multiline graph strings may contain:

- **blank lines**
- **arbitrary white space**
- **task names**
- **internal comments:** following the # character
- **cycle time offsets:** `A[T-6]`

- **success triggers:** `foo => bar`
- **failure triggers:** `foo:fail => bar`
- **suicide triggers:** `foo => !bar`
- **explicit output triggers:** `foo:out1 => bar`
- **conditional triggers:** `(A | B) & C => D`
- **trigger-generators:** `Python:list( "m"+str(i)+"=> p"+str(i) for i in range(1,7))`

8.3.2 Interpreting Graph Strings

Suite dependency graphs can be broken down into pairs in which the left side (which may be a single task or family, or several that are conditionally related) defines a trigger for the task or family on the right. For instance the "word graph" *C triggers off B which triggers off A* can be deconstructed into pairs *C triggers off B* and *B triggers off A*.

In the case of cycling tasks, the triggers defined by a graph string are valid for cycle times matching the list of hours specified for the graph section. For example this graph,

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        [[[0,12]]]
            graph = "A => B"
```

implies that B triggers off A for cycle times in which the hour matches 0 or 12.

To define intercycle dependencies, attach an offset indicator to the left side of a pair:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        [[[0,12]]]
            graph = "A[T-12] => B"
```

This means B[T] triggers off A[T-12] for cycle times T with hours matching 0 or 12. Note that *T must be left implicit unless there is a cycle time offset* (this helps to keep graphs clean and concise because the majority of tasks in a typical suite will only depend on others with the same cycle time) and that *cycle time offsets can only appear on the left* (because each pair defines a trigger for the right task at cycle time T).

Now, having explained that dependency graphs are interpreted pairwise, you can optionally chain pairs together to "follow a path" through the graph. So this,

```
# SUITE.RC
    graph = """A => B   # B triggers off A
               B => C   # C triggers off B"""
```

is equivalent to this:

```
# SUITE.RC
    graph = "A => B => C"
```

Cycle time offsets, if they appear in a chain of triggers, must be leftmost (because, as explained previously they can't appear on the right of any pair). So this is legal:

```
# SUITE.RC
    graph = "A[T-6] => B => C"   # OK
```

but this isn't:

```
# SUITE.RC
    graph = "A => B[T-6] => C"   # ERROR!
```

The trigger `A => B[T-6]` does not make sense in any case - if this kind of relationship seems necessary it probably means that B should be "reassigned" to the next cycle (keep in mind that cycle time is really just a label used to define the relatonships between tasks).

Each trigger in the graph must be unique but *the same task can appear in multiple pairs or chains*. Separately defined triggers for the same task have an AND relationship. So this:

```
# SUITE.RC
    graph = """A => X   # X triggers off A
               B => X   # X also triggers off B"""
```

is equivalent to this:

```
# SUITE.RC
    graph = "A & B => X"   # X triggers off A AND B
```

In summary, the branching tree structure of a dependency graph can be partitioned into lines (in the suite.rc graph string) of pairs or chains, in any way you like, with liberal use of internal white space and comments to make the graph structure as clear as possible.

```
# SUITE.RC
# B triggers if A succeeds, then C and D trigger if B succeeds:
    graph = "A => B => C & D"
# which is equivalent to this:
    graph = """A => B => C
               B => D"""
# and to this:
    graph = """A => B => D
               B => C"""
# and to this:
    graph = """A => B
               B => C
               B => D"""
# and it can even be written like this:
    graph = """A => B # blank line follows:

               B => C # comment ...
               B => D"""
```

8.3.3 Graph Types (VALIDITY)

A suite definition can contain multiple graph strings that are combined to generate the final graph. There are different graph VALIDITY section headings (the heading of the suite.rc section that encloses the graph string) for cycling, one-off asynchronous, and repeating asynchronous tasks. Additionally, there may be multiple graph strings (under different VALIDITY sections) for cycling tasks, for tasks with different dependencies at different cycle times.

The small suite definitions used in the following subsections can be found in the central database registered under the validity *group name.*

8.3.3.1 One-off Asynchronous Tasks

Figure 30 shows a small suite of one-off asynchronous tasks; these have no associated cycle time and don't spawn successors (once they're all finished the suite just exits). The integer 1 attached to each graph node is just an arbitrary label, akin to the task cycle time in cycling tasks; it increments when a repeating asynchronous task (below) spawns.

8.3.3.2 Cycling Tasks

Valid cycle times for cycling tasks are defined by the graph VALIDITY section headings - lists of hours in the day - for the graph strings in which the tasks appear. Figure 31 shows a small suite of cycling tasks.


```
# SUITE.RC
title = some one-off asynchronous tasks
[scheduling]
    [[dependencies]]
        graph = "foo => bar & baz => waz"
```

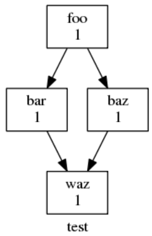

Figure 30: One-off Asynchronous Tasks.

```
# SUITE.RC
title = some one-off asynchronous tasks
# (no dependence between cycles here)
[scheduling]
    [[dependencies]]
        [[[0,12]]]
            graph = "foo => bar & baz => waz"
```

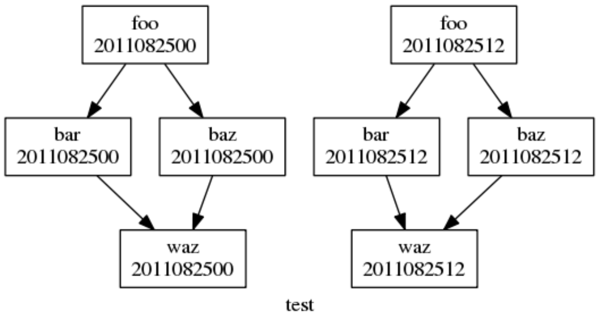

Figure 31: Cycling Tasks.

8.3.3.3 Combined Graphs

Cycling tasks can be made to wait on one-off asynchronous tasks, as shown in Figure 32. Alternatively, they can be made to wait on one-off *synchronous* start-up tasks, which have an associated cycle time event though they are non-cycling tasks - see Figure 33.

8.3.3.3.1 Synchronous Start-up vs One-off Asynchronous Tasks

One-off synchronous start-up tasks run only when a cycling suite is *cold-started* and they are often associated with subsequent one-off *cold-start tasks* used to bootstrap a cycling suite into existence.

The distinction between cold- and warm-start is only meaningful for cycling tasks, and one-off asynchronous tasks may be best used in constructing entirely non-cycling suites.

However, one-off asynchronous tasks can precede cycling tasks in the same suite, as shown above. It seems likely that, if used in this way, they will be intended as start-up tasks - so currently *one-off asynchronous tasks only run in a cold-start.*

8.3.3.4 Repeating Asynchronous Tasks

Repeating asynchronous tasks can be used to process satellite data that arrives asynchronously - i.e. at irregular time intervals. Each new dataset must have a unique "asynchronous ID" - if it doesn't already have such an ID you could use some string representation of the data arrival time. The graph VALIDITY section heading must contain "ASYNCID:" followed by a regular expression designed to match the actual IDs. Addtionally, one task in the suite must be designated the "daemon" - it waits indefinitely on incoming data and reports each new dataset and its ID back to the suite by means of a special output message. When the task proxy receives that message it dynamically registers a new output (containing the asynchronous ID) that downstream tasks can then trigger off. The downstream tasks likewise have prerequisites containing the ID pattern (because they trigger off the aforementioned outputs) and when these

```
# SUITE.RC
title = one-off async and cycling tasks
# (with dependence between cycles too)
[scheduling]
    [[dependencies]]
        graph = "prep1 => prep2"
        [[[0,12]]]
            graph = """
    prep2 => foo => bar & baz => waz
    foo[T-12] => foo
                    """
```

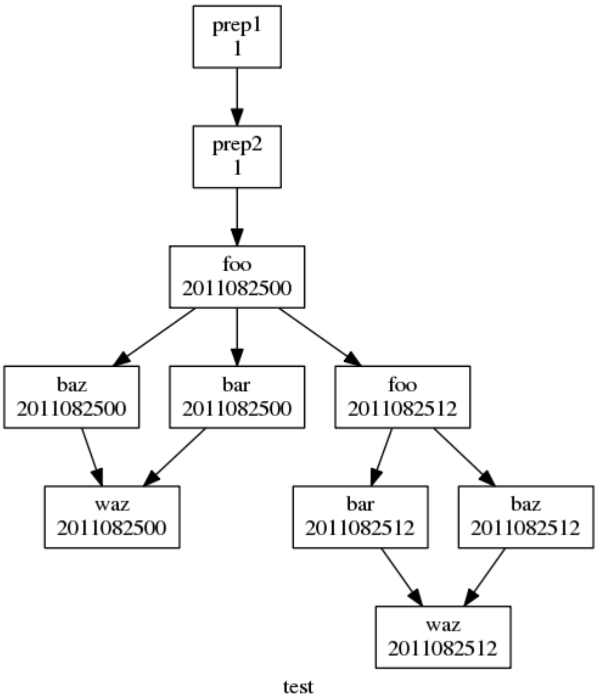

Figure 32: One-off asynchronous and cycling tasks in the same suite.

```
# SUITE.RC
title = one-off start-up and cycling tasks
# (with dependence between cycles too)
[scheduling]
    [[special tasks]]
        start-up = prep1, prep2
    [[dependencies]]
        [[[0,12]]]
            graph = """
    prep1 => prep2 => foo => bar & baz => waz
    foo[T-12] => foo
                    """
```

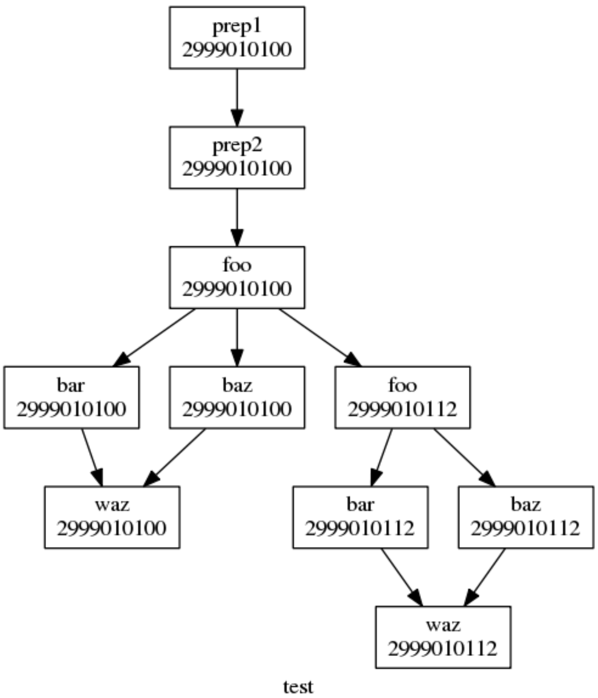

Figure 33: One-off synchronous and cycling tasks in the same suite.

```
# SUITE.RC
title = a suite of repeating asynchronous tasks
# for processing real time satellite datasets
[scheduling]
    [[dependencies]]
        [[[ASYNCID:satX-\d{6}]]]
            # match datasets satX-1424433 (e.g.)
            graph = "watcher:a => foo:a & bar:a => baz"
            daemon = watcher
[runtime]
    [[watcher]]
        [[[outputs]]]
            a = "New dataset $(ASYNCID) ready for processing"
    [[foo,bar]]
        [[[outputs]]]
            a = "Products generated from dataset $(ASYNCID)"
```

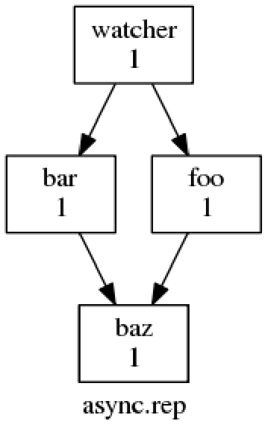

Figure 34: Repeating Asynchronous Tasks.

get satisfied the actual ID is substituted into their own registered outputs. Additionally, each asynchronous task proxy passes the ID to its task execution environment as `$ASYNCID` to allow identification of the correct dataset by task scripts. In this way the whole tree of tasks becomes dedicated to processing the new dataset, but if the data arrives quickly successive datasets can be processed in parallel. As Figure 34 shows, a repeating asynchronous suite currently plots just like a one-off asynchronous suite. But at run time the daemon task stays put, while the others continually spawn successors to wait for new datasets to come in. The *asynchronous.repeating* example suite demonstrates how to do this in a real suite.

8.3.4 Trigger Types

8.3.4.1 Success Triggers

To trigger off a task finishing successfully:

```
# SUITE.RC
# B triggers if A SUCCEEDS:
    graph = A => B
```

8.3.4.2 Failure Triggers

To trigger off a task that fails:

```
# SUITE.RC
# B triggers if A FAILS:
    graph = A:fail => B
```

See *Suicide Triggers* (Section 8.3.4.6) for how to handle task B if A does not fail.
To trigger off a task finishing (success or failure):

```
# SUITE.RC
# B triggers if A either SUCCEEDS or FAILS:
    graph = A | A:fail => B
```

8.3.4.3 Internal Triggers

Only required if you need to trigger before the upstream task finishes.

```
# SUITE.RC
[scheduling]
    [[dependencies]]
```

```
        [[[6,18]]]
            # B triggers off specific OUTPUT "out1" of task A:
            graph = A:out1 => B
[runtime]
    [[A]]
        [[[outputs]]]
            out1 = "NWP products uploaded for $(CYLC_TASK_CYCLE_TIME)"
```

Task A must emit this message when the actual output has been completed - see *Reporting Internal Outputs Completed*, Section 9.4.2.

8.3.4.4 Intercycle Triggers

Typically most tasks in a suite will trigger off other cotemporal tasks (same cycle time) but some may depend on tasks with earlier cycle times. This notably applies to warm-cycled forecast models, which depend on their own previous instances (see below); but other kinds of intercycle are possible too.[13] Here's how to express this kind of relationship in cylc:

```
# SUITE.RC
[dependencies]
    [[0,6,12,18]]
        # B triggers off A in the previous cycle
        graph = "A[T-6] => B"
```

This notation can be used with explicit outputs too:

```
# SUITE.RC
    # B triggers if A in the previous cycle fails:
    graph = "A[T-6]:fail => B"
```

8.3.4.5 Conditional Triggers

AND operators (`&`) can appear on both sides of an arrow. They provide a concise alternative to defining multiple triggers separately:

```
# SUITE.RC
# 1/ this:
    graph = "A & B => C"
# is equivalent to:
    graph = """A => C
           B => C"""
# 2/ this:
    graph = "A => B & C"
# is equivalent to:
    graph = """A => B
           A => C"""
# 3/ and this:
    graph = "A & B => C & D"
# is equivalent to this:
    graph = """A => C
           B => C
           A => D
           B => D"""
```

OR operators (`|`), for truly conditional triggers, can only appear on the left,[14]

```
# SUITE.RC
# C triggers when either A or B finishes:
    graph = "A | B => C"
```

Forecasting suites typically have quite simple requirements for conditional triggers, but should you need it you can use any valid parenthesised conditional expression, e.g.:

[13] In NWP forecast analysis suites parts of the observation processing and data assimilation subsystem will typically also depend on model background fields generated by the previous forecast.

[14] An OR operator on the right doesn't make much sense: if "B or C" triggers off A, what exactly should cylc do when A finishes?

```
# SUITE.RC
title = asynchronous automated recovery
description = """
Model task failure triggers diagnosis
and recovery tasks, which take themselves
out of the suite if model succeeds. Model
post processing triggers off model OR
recovery tasks.
              """
[scheduling]
    [[dependencies]]
        graph = """
pre => model
model:fail => diagnose => recover
model => !diagnose & !recover
model | recover => post
                """
[runtime]
    [[model]]
        # UNCOMMENT TO TEST FAILURE:
        # command scripting = false
```

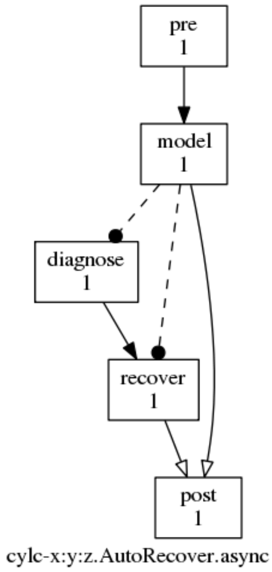

Figure 35: Automated failure recovery via suicide triggers.

```
# SUITE.RC
    graph = """
        # D triggers if A or (both B and C) succeed:
        A | B & C => D
        # Z triggers if both (W or X) and Y succeed.
        (W | X) & Y => Z
            """
```

Condititional triggers are plotted with open arrow heads.

8.3.4.6 Suicide Triggers

Suicide triggers take tasks out of the suite. This can be used for automated failure recovery. The suite.rc listing and accompanying graph in Figure 35 show how to define a chain of failure recovery tasks that trigger if they're needed but otherwise remove themselves from the suite (you can run the *AutoRecover.async* example suite to see how this works). The dashed graph edges ending in solid dots indicate suicide triggers, and the open arrowheads indicate conditional triggers as usual.

8.3.4.7 Family Triggers

Family names defined by the namespace inheritance hierarchy (see Section 8.4) can be used in the suite graph as shorthand for the effective dependencies on member tasks. Member dependencies are substituted in such a way that downstream tasks will not trigger until all family members have either succeeded or failed. For example the graph in this suite:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        graph = "foo => fam => bar"
[runtime]
    [[fam]]  # a family (because others inherit from it)
    [[a,b]]  # family members (inherit namespace fam)
        inherit = fam
```

is equivalent to this:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        graph = "foo => a & b => bar"
```

And the graph in this suite:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        graph = "fam:fail => bar"
[runtime]
    [[fam]]
    [[a,b]]
        inherit = fam
```

is equivalent to this:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        graph = "( a:fail|b:fail ) & ( a|a:fail ) & ( b|b:fail ) => bar"
```

Task family triggers should be used sparingly as a convenient simplification for groups of tasks that would naturally trigger at the same time anyway (e.g. forecast ensembles and multiple tasks for processing different types of observations that are all made available at the same time) and whose outputs would all be put to use at a similar time too. Otherwise family triggers will unnecessarily constrain cylc's ability to achieve maximum functional parallelism, because all family members have to finish before downstream processing can continue.[15]

8.3.5 Satisfying Intercycle Dependence At Start-Up

In suites with intercycle dependence some kind of bootsrapping process is required to get the suite going initially. In the example shown in *Intercycle Triggers* (Section 8.3.4.4), for instance, in the very first cycle there is no previous instance of task A to satisfy B's prerequisites.

8.3.5.1 Cold-Start Tasks

A *cold-start* task is a special one-off task used to satisfy the initial previous-cycle dependence of another cotemporal task. In effect, the cold-start task masquerades as the previous-cycle trigger of its associated cycling task.

A cold-start task may invoke real processing, probably to generate the files that are normally produced by the associated cycling task; or it could be a dummy task that represents some external spinup process, presumably resulting in the same files, that has to be completed before the suite is started. In the latter case the cold-start task can just report itself successfully completed after checking that the required files are present.

This kind of relationship can easily be expressed with a conditional trigger:

```
# SUITE.RC
[scheduling]
    [[special tasks]]
        cold-start = ColdFoo
    [[dependencies]]
        [[[0,6,12,18]]]
            graph = "ColdFoo | Bar[T-6] => Foo"
```

i.e. Foo[T] can trigger off *either* Bar[T-6] *or* ColdFoo[T]. At start-up ColdFoo will do the job, and thereafter Bar[T-6] will do it.

[15] Actually downstream tasks can also trigger off individual family members, rather than off the family as a whole, but extensive use of this would probably defeat the purpose of using a family in the first place.

Cold-start tasks can also be inserted into the suite at run time to cold-start just their associated cycling tasks, if a problem of some kind prevents continued normal cycling.

8.3.5.2 Warm-Starting A Suite

Cold-start tasks have to be declared as such in the suite.rc “special tasks” section so that cylc knows they are one-off (non-spawning) tasks, but also because they play a critical role in suite warm-starts. A suite that has previously been running and was then shut down can be warm-started at a particular cycle time, an alternative to *restarting* from a previous state (although restarting is preferred because a warm start is likely to involve re-running some tasks). A warm-start assumes the existence of a previous cycle (i.e. that any files from the previous cycle required by the new cycle are in place already) so cold-start tasks do not need to run *but* cylc itself does not know the details of the previous cycle (it does in a restart, but not in a warm-start) so it still has to solve the bootstrapping problem to get the suite started. It does this by starting the suite with designated cold start tasks in the *succeeded* state - in other words finished cold start tasks stand in for the previous finished cycle, rather than pretending to be a running previous cycle as they do in a cold-start.

8.3.6 Model Restart Dependencies

Warm cycled forecast models generate *restart files*, e.g. model background fields, that are required to initialize the next forecast (this is essentially the definition of “warm cycling”). In fact restart files will often be written for a whole series of subsequent cycles in case the next cycle (or the next and the next-next, and so on) cycle has to be omitted:

```
# SUITE.RC
[scheduling]
    [[special tasks]]
        sequential = A
    [[dependencies]]
        [[[0,6,12,18]]]
            # Model A cold-start and restart dependencies:
            graph = "ColdA | A[T-6] | A[T-12] | A[T-18] | A[T-24] => A"
```

In other words, task A can trigger off a cotemporal cold-start task, *or* off its own previous instance, *or* off the instance before that, and so on. Restart dependencies are unusual because although A *could* trigger off A[T-12] we don’t actually want it to do so unless A[T-6] fails and can’t be fixed. *This is why Task A, above, is declared to be ‘sequential’.*[16] Sequential tasks do not spawn a successor until they have succeeded (by default, tasks spawn as soon as they start running in order to get maximum functional parallelism in a suite) which means that A[T+6] will not be waiting around to trigger off an older predecessor while A[T] is still running. If A[T] fails though, the operator can force it, on removal, to spawn A[T+6], whose restart dependencies will then automatically be satisfied by the older instance, A[T-6].

Forcing a model to run sequentially means, of course, that its restart dependencies cannot be violated anyway, so we might just ignore them. This is certainly an option, but it should be noted that there are some benefits to having your suite reflect all of the real dependencies between the tasks that it is managing, particularly for complex multi-model operational suites in which the suite operator might not be an expert on the models. Consider such a suite in which a failure in a driving model (e.g. weather) precludes running one or more cycles of the downstream models (sea state, storm surge, river flow, . . .). If the real restart dependencies of each model are known to the suite, the operator can just do a recursive purge to remove the

[16] A warm cycling model that only writes out one set of restart files, for the very next cycle, does not need to be declared sequential because this early triggering problem cannot arise.

subtree of all tasks that can never run due to the failure, and then cold-start the failed driving model after a gap (skipping as few cycles as possible until the new cold-start input data are available). After that the downstream models will kick off automatically so long as the gap is spanned by their respective restart files, because their restart dependencies will automatically be satisfied by the older pre-gap instances in the suite. Managing this kind of scenario manually in a complex suite can be quite difficult.

Finally, if a warm cycled model is declared to have explicit restart outputs, and is not declared to be sequential, and you define appropriate labeled restart outputs which *must contain the word 'restart'*, then the task will spawn as soon its last restart output is completed so that successives instances of the task will be able to overlap (i.e. run in parallel) if the opportunity arises. Whether or not this is worth the effort depends on your needs.

```
# SUITE.RC
[scheduling]
    [[special tasks]]
        explicit restart outputs = A
    [[dependencies]]
        [[[0,6,12,18]]]
            graph = "ColdA | A[T-18]:res18 | A[T-12]:res12| A[T-6]:res6 => A"
[runtime]
    [[A]]
        [[[outputs]]]
            r6  = restart files completed for $(CYLC_TASK_CYCLE_TIME+6)
            r12 = restart files completed for $(CYLC_TASK_CYCLE_TIME+12)
            r18 = restart files completed for $(CYLC_TASK_CYCLE_TIME+18)
```

8.4 Runtime - Namespaces

The [runtime] section of a suite.rc file configures what to execute, and where and how, when tasks are ready to run. The sections immediately below [runtime] are, for want of a better term, *namespaces*[17] that define runtime properties for individual tasks and families of tasks.

Every namespace contains the same set of configuration items (see the *Suite.rc Reference*, Appendix A, for the complete list of items). Namespaces can inherit from other namespaces, overriding inherited items as required; *this allows configuration of related tasks without repetition.* A namespace represents a family if other namespaces inherit from it.

A namespace that does not explicitly inherit from another automatically inherits from the *root* namespace (below). Namespaces thus form a tree-like hierarchy of nested task families, rooted on the root namespace, in which the leaves are the individual tasks of the suite.

Nested families from the namespace inheritance hierarchy, even if they are not used as family triggers in the graph, can be expanded or collapsed in suite graphs, and in the graph-based suite control GUI (this will also be added to the text treeview control GUI in a future release). See *Visualization* (Section 8.5) for more on this.

The following listing of the *namespace.one* example suite (which you can import from the central database if you like) illustrates basic runtime property inheritance. How it works should be reasonable clear by inspection; if not read on.

```
# SUITE.RC
title = "User Guide [runtime] example."
[cylc]
    simulation mode only = True # (no task implementations)
[scheduling]
    initial cycle time = 2011010106
    final cycle time = 2011010200
```

[17] We use the term namespace in loose analogy with its meaning in modern programming languages. Possible future enhancements to cylc, such as ability to import specific items from other namespaces rather than just wholesale inheritance, may tighten the analogy.

```
    [[dependencies]]
        graph = "foo => obs => bar"
             # (this is shorthand for "foo => land & ship => bar")
[runtime]
    [[root]] # base namespace for all tasks (defines suite-wide defaults)
        [[[job submission]]]
            method = at_now
        [[[environment]]]
            COLOR = red
    [[obs]]  # family (inherited by land, ship); implicitly inherits root
        [[[environment]]]
            RUNNING_DIR = $HOME/running/$CYLC_TASK_NAME
    [[land]] # a task (a leaf on the inheritance tree) in the obs family
        inherit = obs
        description = land obs processing
        command scripting = run-land.sh
    [[ship]] # a task (a leaf on the inheritance tree) in the obs family
        inherit = obs
        description = ship obs processing
        command scripting = run-ship.sh
        [[[job submission]]]
            method = loadleveler
        [[[environment]]]
            RUNNING_DIR = $HOME/running/ship  # override obs environment
            OUTPUT_DIR = $HOME/output/ship    # add to obs environment
    [[foo]]
        # (just inherits from root)

    # The task [[bar]] is implicitly defined by its presence in the
    # graph. It is also a dummy task that just inherits from root.
```

8.4.1 Namespace Names

Namespace names may contain letters, digits, underscores, and hyphens. They may not contain colons as that would preclude use of suite registration names in shell `$PATH` variables. The '.' character is the suite registration hierarchy delimiter (which separates suite registration groups and names, e.g. my_suites.test.foo). *Task names should not be hardwired into task implementations* because task and suite identity can be extracted portably from the task execution environment supplied by cylc (Section 8.4.4) - then to rename a task, can just change its name in the suite.rc file.

8.4.2 Root - Runtime Defaults

The root namespace, at the base of the inheritance hierarchy, provides default configuration for all tasks in the suite.

Most root items are unset by default, but some values are set, sufficient to allow simple test suites to be defined by dependency graph alone - as can be seen from many of the examples that illustrate this User Guide (command scripting, for example, defaults to printing a simple message, sleeping for ten seconds, and then exiting). These default values are documented with each configuration item in Appendix A, and Section A.6 shows them all in context. You can override them or provide your own defaults for other items by explicitly configuring the root namespace.

8.4.3 Defining Multiple Namespaces At Once

Groups of similar tasks or families that differ only in a few configuration details can be defined in single namespace sections by putting a list of names, or a Python expression that generates a list of names, in the section heading. Potential applications include groups of similar obs processing tasks, and forecasting ensembles.

Any occurrence of `$(TASK)`, in any configuration items in the namespace, will be replaced with the actual name of the task or family prior to run time.

The two annotated multidef *example suites use this feature.*

8.4.4 Task Execution Environment

The task execution environment contains suite and task identity variables provided by cylc, and user-defined environment variables. The environment is explicitly exported (by the task job script) prior to executing task command scripting (see *Task Job Submission*, Section 10).

Suite and task identity are exported first, so that user-defined variables can refer to them. Order of definition is preserved throughout so that variable assignment expressions can safely refer to previously defined variables.

Additionally, access to cylc itself is configured prior to the user-defined environment, so that variable assignment expressions can make use of cylc utility commands:

```
# SUITE.RC
[runtime]
    [[foo]]
        [[[environment]]]
            REFERENCE_TIME = $( cylc util cycletime --add=6 )
```

8.4.4.1 User Environment Variables

The user-defined environment is the sum of a task's inherited `[[[environment]]]` sections.

```
# SUITE.RC
[runtime]
    [[root]]
        [[[environment]]]
            COLOR = red
            SHAPE = circle
    [[foo]]
        [[[environment]]]
            COLOR = blue  # root override
            TEXURE = rough # new variable
```

This results in a task *foo* with `SHAPE=circle`, `COLOR=blue`, and `TEXTURE=rough` in its environment.

8.4.4.2 Overriding Environment Variables

When you override inherited namespace items the original parent item definition is *replaced* by the new definition. This applies to all items including those in the environment sub-sections which, strictly speaking, are not “environment variables” until they are written, post inheritance processing, to the task job script that executes the associated task. Consequently, if you override an environment variable you cannot also access the original parent value:

```
# SUITE.RC
[runtime]
    [[foo]]
        [[[environment]]]
            COLOR = red
    [[bar]]
        inherit = foo
        [[[environment]]]
            tmp = $COLOR        # !! ERROR: $COLOR is undefined here
            COLOR = dark-$tmp   # !! as this overrides COLOR in foo.
```

The compressed variant of this, `COLOR = dark-$COLOR`, is also in error for the same reason. To achieve the desired result you must use a different name for the parent variable:

```
# SUITE.RC
[runtime]
```



```
    [[foo]]
        [[[environment]]]
            FOO_COLOR = red
    [[bar]]
        inherit = foo
        [[[environment]]]
            COLOR = dark-$FOO_COLOR  # OK
```

8.4.4.3 Suite And Task Identity Variables

The task identify variables provided to tasks by cylc are:

```
$CYLC_TASK_ID                      # X%2011051118 (e.g.)
$CYLC_TASK_NAME                    # X
$CYLC_TASK_CYCLE_TIME              # 2011051118
$CYLC_TASK_NAMESPACE_HIERARCHY     # "root postproc X" (e.g.)
```

The suite identify variables provided to tasks by cylc are:

```
$CYLC_SUITE_DEF_PATH    # $HOME/mysuites/baz (e.g.)
$CYLC_SUITE_REG_NAME    # foo.bar.baz (e.g.)
$CYLC_SUITE_REG_PATH    # foo/bar/baz
$CYLC_SUITE_HOST        # orca.niwa.co.nz (e.g.)
$CYLC_SUITE_PORT        # 7766 (e.g.)
$CYLC_SUITE_OWNER       # oliverh (e.g.)
```

The variable `$CYLC_SUITE_REG_PATH` is just `$CYLC_SUITE_REG_NAME` (the name under which the suite definition is registered in your suite database) translated into a directory path. This can be used when configuring suite logging directories and the like to put suite output in a directory tree that reflects the suite registration hierarchy (as opposed to the namespace hierarchy).

Some of these variables are also used by cylc task messaging commands in order to automatically target the right task proxy object in the right suite.

8.4.4.4 Suite Share And Task Work Directories

The following variables, and the directories they represent, are also available to running tasks:

```
$CYLC_TASK_WORK_PATH     # task running directory
$CYLC_SUITE_SHARE_PATH   # suite shared directory
```

These two directories are configured in the suite.rc file, and they are created on the fly, if necessary, by task job scripts prior to executing the task command scripting. The job script changes directory to the work/running directory before executing the task command scripting. The shared directory is intended as a shared data directory for multiple tasks (however, the directory path can be different for different families of tasks or individual tasks if you like).

8.4.4.5 Environment Variable Evaluation

Variables in the task execution environment are not evaluated in the shell in which the suite is running prior to submitting the task. They are written in unevaluated form to the job script that is submitted by cylc to run the task (Section 10.1) and are therefore evaluated when the task begins executing under the task owner account on the task host. Thus `$HOME`, for instance, evaluates at run time to the home directory of task owner on the task host.

8.4.5 Remote Task Hosting

If a task declares an owner other than the suite owner and/or a host other than the suit host, e.g.:

```
# SUITE.RC
[runtime]
    [[foo]]
        [[[remote]]]
            host = orca.niwa.co.nz
            owner = bob
            cylc directory = /path/to/remote/cylc/installation/on/foo
            suite definition directory = /path/to/remote/suite/definition/on/foo
```

cylc will attempt to execute the task on the declared host, by the configured job submission method, as the declared owner, using passwordless ssh.

- passwordless ssh must be configured between the suite owner on the suite host and the task owner on the remote host.
- cylc and Pyro must be installed on the remote host so that the remote task can communicate with the suite (but graphviz is not needed).
- the suite definition directory must be installed on the remote host, if the task needs access to scripts in the suite bin directory or to any other files stored there.

A local task to run under another user account is treated as a remote task.

You may not need this functionality if you have a cross-platform resource manager, such as loadleveler, that allows you to submit a job locally to run on the remote host.

Remote host functionality, like other namespace properties, can be declared globally (in the root namespace) or per family, or per individual task. Use the global settings if all or most of your tasks need to run on the same remote host.

The remote cylc directory is required to give remote tasks access to cylc commands; the remote suite directory gives them access to suite files via `$CYLC_SUITE_DEF_PATH` on the remote platform, and to the suite bin directory via `$PATH`. If a remote suite definition directory is not given it will be assumed that the local path should be used - but the local user's home directory (if present) will be substituted in the file path for the literal `$HOME` in case the user's home directory path is different on the remote host.

Note that you can easily run the cylc example suites on a remote host. For the example suites with task implementations that work with "real" files, you'll have to run *all* tasks on the same remote host so that they can access their common input and output files. To distribute a suite across several hosts you must arrange (using additional tasks) to transfer files between the hosts as required to satisfy the real I/O dependencies.

8.4.5.1 Remote Log Directories

The stdout and stderr from local tasks is directed into files in the *job submission log directory* (specified in the suite.rc file) as explained in Section 10.3. The same goes for remotely hosted tasks, except that the task owner's home directory is substituted as described above for the remote suite definition directory. Remote log directories are created on the fly by cylc, during job submission, if they do not already exist.

8.5 Visualization

This is the final major section in the suite.rc file. It is used to configure suite graph plotting - principally graph node (task) and edge (dependency arrow) style attributes. Tasks can be grouped for the purpose of applying common style attributes. See the suite.rc reference (Appendix A) for details.

8.5.1 Collapsible Task Families In Suite Graphing And GUIs

```
# SUITE.RC
[visualization]
    # list namespace families to be shown in collapsed form
    collapsed families = family1, family2
```

Nested families from the namespace inheritance hierarchy, even if they are not used as family triggers in the graph, can be expanded or collapsed in suite graphs and by menu options in the graph-based suite control GUI (this will also be added to the text treeview control GUI in a future release).

Family nodes in the graph-based suite control GUI are not currently color-coded in real time according to what state their members are in. However, any ungraphed tasks, which includes the members of collapsed families, are automatically plotted as rectangular nodes to the right of the main graph if they are doing anything interesting (submitted, running, or failed).

Note that family relationships can be defined purely for visualization purposes - you can group tasks at root level in the inheritance hierarchy prior to defining real properties at higher levels.

Figure 36 illustrates successive expansion of nested task families in the *namespaces* example suite, which has the following namespace hierarchy:

```
% cylc list --tree cylc-x:y:z.namespaces
root
 |-GEN
 | |-OPS
 | | |-aircraft   OPS aircraft obs processing
 | | |-atovs      OPS ATOVS obs processing
 | | `-atovs_post OPS ATOVS postprocessing
 | `-VAR
 |   |-AnPF       runs VAR AnalysePF
 |   `-ConLS      runs VAR ConfigureLS
 |-baz
 | |-bar1         Task bar1 of baz
 | `-bar2         Task bar2 of baz
 |-foo            No description provided
 `-prepobs        obs preprocessing
```

9 Task Implementation

This section lays out the minimal requirements on external commands, scripts, or executables invoked by cylc to carry out task processing.

9.1 Most Tasks Require No Modification For Cylc

Any existing command, script, or executable can function as a cylc task (or rather, perform the external processing that the task represents) if the following conditions are met:

- The task must not have internal/early outputs that others trigger off - otherwise the external task processing must be modified to report when those outputs are completed.
- The external process invoked by cylc must not detach and exit after spawning secondary processes to complete the job - otherwise the secondary processes must be modified to report final success or failure to the suite.
- Any external processing must return zero exit status on success, and non-zero on failure (this is normal practice anyway) to allow cylc's automatic error trapping to work.

If these requirements are not met, see *Some Tasks Require Modification For Cylc*, Section 9.4.

The following suite runs a couple of external scripts that are not cylc-aware, but which meet the requirements above so that no special treatment is required at all:

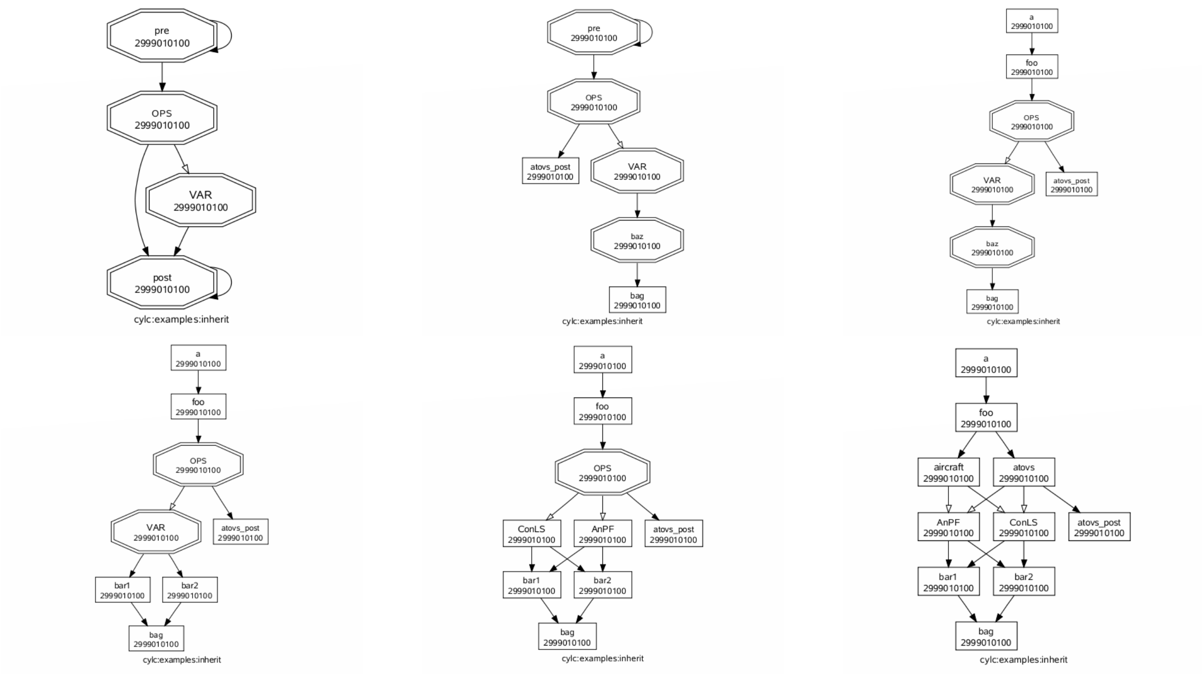

Figure 36: Graphs of the *namespaces* example suite showing various states of expansion of the nested namespace family hierarchy, from all families collapsed (top left) through to all expanded (bottom right). This can also be done by right-clicking on tasks in the graph-based suite control GUI.

```
# SUITE.RC
[runtime]
    [[foo]]
        description = a task that runs foo.sh
        command scripting = foo.sh OPTIONS ARGUMENTS
    [[bar]]
        description = a task that runs bar.sh
        command scripting = """echo HELLO
                               bar.sh
                               echo BYE"""
    [[baz]]
        description = a task that runs baz.sh and retries on failure
        command scripting = """echo attempt No.1
                               baz.sh""",
                            """echo attempt No.2  # only invoked if try No.1 fails
                               baz.sh --retry"""
```

9.2 Suite.rc Inlined Tasks

Simple tasks can be entirely implemented within the suite.rc file because the task *command scripting* string can contain as many lines of code as you like (or even a list of such strings, for retry on failure).

9.3 Return Non-zero Exit Status On Error

The requirement to abort with non-zero exit status on error (which should be normal scripting practice in any case) allows the task job script to trap errors and send a `cylc task failed` message to alert the suite. You can use `set -e` to avoid writing explicit error checks for every operation:

```
#!/bin/bash
set -e   # abort on error
mkdir /illegal/dir   # this error will abort the script with non-zero exit status
```

9.4 Some Tasks Require Modification For Cylc

9.4.1 Voluntary Messaging

You can, if you like, modify task scripts to send any explanatory or progress messages to the suite as the task runs. For example, a task can send a priority critical message before aborting on error:

```
#!/bin/bash
set -e  # abort on error
if ! mkdir /illegal/dir; then
    # (use inline error checking to avoid triggering the above 'set -e')
    cylc task message -p CRITICAL "Failed to create directory /illegal/dir"
    exit 1 # now abort non-zero exit status to trigger the task failed message
fi
```

You can also use this syntax:

```
#!/bin/bash
set -e
mkdir /illegal/dir || {  # inline error checking using OR operator
    cylc task message -p CRITICAL "Failed to create directory /illegal/dir"
    exit 1
}
```

But not this:

```
#!/bin/bash
set -e
mkdir /illegal/dir  # aborted via 'set -e'
if [[ $? != 0 ]]; then  # so this will never be reached.
    cylc task message -p CRITICAL "Failed to create directory /illegal/dir"
    exit 1
fi
```

You can also send warning messages, or general information:

```
#!/bin/bash
# a warning message (this will be logged by the suite):
cylc task message -p WARNING "oops, something's fishy here"
# information (this will also be logged by the suite):
cylc task message "Hello from task foo"
```

This may be useful - any message received from a task is logged by cylc - but it is not a requirement. If error messages are not reported, for instance, task failure will still be registered, and task stdout and stderr logs can still be examined for evidence of what went wrong.

9.4.2 Reporting Internal Outputs Completed

Tasks with internal outputs that allow downstream processing to trigger before they are finished must report when those internal outputs have been completed:

```
#!/bin/bash
# (task foo implementation)
# ...
# report an output completed:
cylc task message "foo products uploaded for $CYLC_TASK_CYCLE_TIME"
```

This must match one of the task's registered outputs:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        [[[6,18]]]
            graph = foo:output1 => bar
[runtime]
    [[foo]]
        [[[outputs]]]
            output1 = "foo products uploaded for $(CYLC_TASK_CYCLE_TIME)"
```

otherwise it will just be logged as a progress report or similar. *Note the required round brackets in the suite.rc cycle time variable.*

9.4.3 Tasks With Initiating Processes That Detach And Exit Early

Tasks with initiating scripts or processes that spawn jobs internally (e.g. to a batch queue scheduler or to another host) and then detach and exit without seeing the resulting processing through must arrange for the spawned processing to send its own "cylc task succeeded" or "cylc task failed" messages on completion - because the cylc-generated job script (Section 10.1) that otherwise does automatic completion cannot know when the task is really finished.

For tasks in this category you must disable automatic completion messaging:

```
# SUITE.RC
[runtime]
    [[root]]
        manual completion = True    # global setting
    [[foo]]
        manual completion = False   # task-specific setting
```

Reporting success or failure is just a matter of calling the cylc messaging commands:

```
#!/bin/bash
# ...
if $SUCCESS; then
    # release my task lock and report success
    cylc task succeeded
    exit 0
else
    # release my task lock and report failed
    cylc task failed "Input file X not found"
    exit 1
fi
```

Bear in mind, however, that cylc messaging commands read environment variables that identify the calling task and the target suite, so if your job submission method does not automatically copy its parent environment you must arrange for these variables, at the least, to be propagated through to your spawned sub-jobs.

One way to handle this is to write a *task wrapper* that modifies a copy of the detaching native job scripts, on the fly, to insert completion messaging in the appropriate places, and other variables if necessary, before invoking the (now modified) native process. A significant advantage of this method is that you don't need to permanently modify the model or its associated native scripting for cylc. Another is that you can configure the native job setup for a single test case (running it without cylc) and then have your custom wrapper modify the test case on the fly with suite, task, and cycle-specific parameters as required.

To make this easier, for tasks that declare manual completion messaging, cylc makes non user-defined environment scripting available in a single variable called `$CYLC_SUITE_ENVIRONMENT` that can be inserted into the aforementioned native task scripts prior to calling the cylc messaging commands.[18]

9.4.4 A Custom Task Wrapper Example

The *detaching* example suite contains a script `model.sh` that runs a pseudo-model executable as follows:

[18] Note that `$CYLC_SUITE_ENVIRONMENT` is a string containing embedded newline characters and it has to be handled accordingly. In the bash shell, for instance, it should be echoed in quotes to avoid concatenation to a single line.


```
#!/bin/bash
set -e

MODEL="sleep 10; true"
#MODEL="sleep 10; false"  # uncomment to test model failure

echo "model.sh: executing pseudo-executable"
eval $MODEL
echo "model.sh: done"
```

this is in turn executed by a script `run-model.sh` that detaches immediately after job submission (i.e. it exits before the model executable actually runs):

```
#!/bin/bash
set -e
echo "run-model.sh: submitting model.sh to 'at now'"
SCRIPT=model.sh  # location of the model job to submit
OUT=$1; ERR=$2   # stdout and stderr log paths
# submit the job and detach
RES=$TMPDIR/atnow$$.txt
( at now <<EOF
$SCRIPT 1> $OUT 2> $ERR
EOF
) > $RES 2>&1
if grep 'No atd running' $RES; then
    echo 'ERROR: atd is not running!'
    exit 1
fi
# model.sh should now be running at the behest of the 'at' scheduler.
echo "run-model.sh: done"
```

Note that your **at** *scheduler daemon must be up if you want to test this suite.*

Here's a cylc suite to run this unruly model:

```
title = "Custom Task Wrapper Example"

description = """This suite runs a single task that internally submits a
'model executable' before detaching and exiting immediately - so we have
to handle task completion messaging manually - see the User Guide 8.4.3."""

[scheduling]
    initial cycle time = 2011010106
    final cycle time = 2011010200
    [[special tasks]]
        sequential = model
    [[dependencies]]
        [[[0,6,12,18]]]
        graph = "model"

[runtime]
    [[model]]
        manual completion = True
        command scripting = model-wrapper.sh  # invoke the task via a custom wrapper
        [[[environment]]]
            # location of native job scripts to modify for this suite:
            NATIVESCRIPTS = $CYLC_SUITE_DEF_PATH/native
            # output path prefix for detached model stdout and stderr:
            PREFIX = $HOME/detach
            FOO = "$HOME bar $PREFIX"
```

The suite invokes the task by means of the custom wrapper `model-wrapper.sh` which modifies, on the fly, a temporary copy of the model's native job scripts as described above:

```
#/bin/bash
set -e

# A custom wrapper for the 'model' task from examples:detaching.
# See documentation in the Cylc User Guide.

# Check inputs:
```

```
# location of pristine native job scripts:
cylc util checkvars -d NATIVESCRIPTS
# path prefix for model stdout and stderr:
cylc util checkvars PREFIX

# Get a temporary copy of the native job scripts:
TDIR=$TMPDIR/detach$$
mkdir -p $TDIR
cp $NATIVESCRIPTS/* $TDIR

# Insert task-specific execution environment in $TDIR/model.sh:
SRCH='echo "model.sh: executing pseudo-executable"'
perl -pi -e "s@^${SRCH}@${CYLC_SUITE_REG_NAME_ENVIRONMENT}\n${SRCH}@" $TDIR/model.sh

# Task completion message scripting. Use single quotes here - we don't
# want the $? variable to evaluate in this shell!
MSG='
if [[ $? != 0 ]]; then
   cylc task message -p CRITICAL "ERROR: model executable failed"
   exit 1
else
   cylc task succeeded
   exit 0
fi'
# Insert error detection and cylc messaging in $TDIR/model.sh:
SRCH='echo "model.sh: done"'
perl -pi -e "s@^${SRCH}@${MSG}\n${SRCH}@" $TDIR/model.sh

# Point to the temporary copy of model.sh, in run-model.sh:
SRCH='SCRIPT=model.sh'
perl -pi -e "s@^${SRCH}@SCRIPT=$TDIR/model.sh@" $TDIR/run-model.sh

# Execute the (now modified) native process:
$TDIR/run-model.sh ${PREFIX}-${CYLC_TASK_CYCLE_TIME}-$$.out ${PREFIX}-${CYLC_TASK_CYCLE_TIME}-$$.err

echo "model-wrapper.sh: see modified job scripts under ${TDIR}!"
# EOF
```

If you run this suite, or submit the model task alone with `cylc submit`, you'll find that the usual job submission log files for task stdout and stderr end before the task is finished. To see the "model" output and the final task completion message (success or failure), examine the log files generated by the job submitted internally to the *at* scheduler (their location is determined by the `$PREFIX` variable in the suite.rc file).

It should not be difficult to adapt this example to real tasks with detaching internal job submission. You will probably also need to replace other parameters, such as model input and output filenames, with suite- and cycle-appropriate values, but exactly the same technique can be used: identify which job script needs to be modified and use text processing tools (such as the single line *perl* search-and-replace expressions above) to do the job.

10 Task Job Submission

Task Implementation (Section 9) describes what requirements a command, script, or program, must fulfill in order to function as a cylc task. This section explains how tasks are submitted by cylc when they are ready to run, and how to define new task job submission methods.

10.1 Task Job Scripts

When a task is ready to run cylc generates a temporary *task job script* to configure the execution environment and call the task's command scripting. The job script is the final result of suite.rc task configuration, in particular of the runtime namespace inheritance hierarchy for the task. The script is submitted to run by means of the *job submission method* specified for the task.

Different tasks can have different job submission methods. Like other namespace properties, you can set a default for the suite and override it on a per task basis:

```
# SUITE.RC
[runtime]
   [[root]] # suite defaults
        [[[job submission]]]
            method = loadleveler
   [[foo]] # just task foo
        [[[job submission]]]
            method = at_now
```

The actual command line used to submit the job script is written to stdout by cylc. In the following shell transcript we generate a job script for a task in the QuickStart.c example suite and then examine it:

```
% cylc submit --dry-run QuickStart.c Model%2011080506
> JOB SCRIPT: ~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123
> THIS IS A DRY RUN. HERE'S HOW I WOULD SUBMIT THE TASK:
~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123 </dev/null
    1> ~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123.out
    2> ~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123.err &
```

And here is the generated job script:

```
#!/bin/bash

# ++++ THIS IS A CYLC TASK JOB SCRIPT ++++
# Task: Model%2011080506
# To be submitted by method: 'background'

echo "TASK JOB SCRIPT STARTING"

# CYLC LOCATION, SUITE LOCATION, SUITE IDENTITY:
export CYLC_DIR=/home/oliverh/cylc
export CYLC_MODE=submit
export CYLC_SUITE_HOST=oliverh-33586DL.greta.niwa.co.nz
export CYLC_SUITE_PORT=NONE
export CYLC_SUITE_DEF_PATH=/tmp/oliverh/QuickStart/c
export CYLC_SUITE_REG_NAME=QuickStart.c
export CYLC_SUITE_REG_PATH=QuickStart/c
export CYLC_SUITE_OWNER=oliverh
export CYLC_USE_LOCKSERVER=False
export CYLC_UTC=False

# TASK IDENTITY:
export CYLC_TASK_ID=Model%2011080506
export CYLC_TASK_NAME=Model
export CYLC_TASK_CYCLE_TIME=2011080506
export CYLC_TASK_NAMESPACE_HIERARCHY="root Models Model"

# ACCESS TO CYLC:
PATH=$CYLC_DIR/bin:$PATH
# Access to the suite bin dir:
PATH=$CYLC_SUITE_DEF_PATH/bin:$PATH
export PATH

# SET ERROR TRAPPING:
set -u # Fail when using an undefined variable
# Define the trap handler
HANDLE_TRAP() {
  echo Received signal "$@"
  cylc task failed "Task job script received signal $@"
  trap "" EXIT
  exit 0
}
# Trap signals that could cause this script to exit:
trap "HANDLE_TRAP EXIT" EXIT
trap "HANDLE_TRAP ERR"  ERR
trap "HANDLE_TRAP TERM" TERM
```

```
trap "HANDLE_TRAP XCPU" XCPU

# SEND TASK STARTED MESSAGE:
cylc task started || exit 1

# SHARE DIRECTORY CREATE:
CYLC_SUITE_SHARE_PATH=$CYLC_SUITE_DEF_PATH/share
export CYLC_SUITE_SHARE_PATH
mkdir -p $CYLC_SUITE_DEF_PATH/share || true

# WORK DIRECTORY CREATE:
CYLC_TASK_WORK_PATH=$CYLC_SUITE_DEF_PATH/work/$CYLC_TASK_ID
export CYLC_TASK_WORK_PATH
mkdir -p $(dirname $CYLC_TASK_WORK_PATH) || true
mkdir -p $CYLC_TASK_WORK_PATH
cd $CYLC_TASK_WORK_PATH

# ENVIRONMENT:
TASK_EXE_SECONDS="5"
WORKSPACE="/tmp/$USER/$CYLC_SUITE_REG_NAME/common"
MODEL_INPUT_DIR="$WORKSPACE"
MODEL_OUTPUT_DIR="$WORKSPACE"
MODEL_RUNNING_DIR="$WORKSPACE/Model"
export TASK_EXE_SECONDS WORKSPACE MODEL_INPUT_DIR MODEL_OUTPUT_DIR MODEL_RUNNING_DIR

# TASK COMMAND SCRIPTING:
Model.sh

# WORK DIRECTORY REMOVE:
cd
rmdir $CYLC_TASK_WORK_PATH 2>/dev/null || true

# SEND TASK SUCCEEDED MESSAGE:
cylc task succeeded

echo "JOB SCRIPT EXITING (TASK SUCCEEDED)"
trap "" EXIT

#EOF
```

You can also generate a job script and print it directly to stdout, with `cylc jobscript`.

10.2 Available Methods

There are two basic job submission methods that should be available on any platform, sufficient for running cylc's example suites if not real forecasting systems:

- `background` - run tasks directly in a background shell.
- `at_now` - submit tasks to the rudimentary `at` scheduler (`atd` must be running).

Tasks in a real forecasting system should be submitted to a batch queue scheduler or cross-platform resource manager such as *loadleveler* (IBM). Methods currently available are:

- `loadleveler` - This method submits general (non loadleveler-specific) task scripts to loadleveler. Any *directives* you provide in the suite.rc file will be written to the job script, which will then be submitted to run via `llsubmit`.
- `ll_raw` - This method submits loadleveler-ready scripts (i.e. scripts containing hardwired directives) to loadleveler. This may be necessary for complex (e.g. multi-step) jobs. The original script is copied to make the temporary job script, and cylc environment scripting is inserted into it immediately after the loadleveler directives.
- `ll_ecox` - This is derived from the basic `loadleveler` method. It automatically adapts certain task parameters (such as owner username) to NIWA's EcoConnect operational

environment so that the same suite definition can be used in distinct *oper*, *test*, and *devel* environments in which the suite and task owners, and their home directories, vary accordingly.

10.3 Whither Task stdout And stderr?

When a task is ready to run cylc generates task-specific stdout and stderr filenames containing the task name, cycle time, and a string of digits (seconds since epoch) to ensure uniqueness - rerunning the task won't overwrite any old output:

```
# task job script:
~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123
# task stdout:
~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123.out
# task stderr:
~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123.err
```

(the job submission log directory is configurable in the suite.rc file).

How the stdout and stderr streams are directed into these files depends on the job submission method. The `background` method just uses appropriate output redirection on the command line, as shown above. The `loadleveler` method writes appropriate directives to the job script that is submitted to loadleveler.

Cylc obviously has no control over the stdout and stderr output from tasks that do their own internal output management (e.g. tasks that submit internal jobs and direct their output to other files). For less internally complex tasks, however, these will be complete task job logs. *They can be viewed updating in real time in the suite control GUIs.*

10.4 Defining New Job Submission Methods

Defining a new job submission method requires some minimal amount of Python programming. You can derive (in the sense of object oriented programming inheritance) new methods from one of the existing ones, or directly from cylc's job submission base class,

```
$CYLC_DIR/lib/cylc/job_submission/job_submit.py
```

using the existing methods as examples. Most often this should merely be a matter of defining the command line used to execute the aforementioned job scripts and using the provided stdout and stderr file paths appropriately. For example, here is the entire class code for the `background` method:

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

from job_submit import job_submit

class background( job_submit ):
    """
Run the task job script directly in a background shell.
    """
    # stdin redirection (< /dev/null) allows background execution on
    # remote hosts - ssh needn't wait for the process to finish.
    COMMAND_TEMPLATE = "%s </dev/null 1>%s 2>%s &"
    def construct_jobfile_submission_command( self ):
        command_template = self.job_submit_command_template
        if not command_template:
            command_template = self.COMMAND_TEMPLATE
        self.command = command_template % ( self.jobfile_path,
                                            self.stdout_file,
                                            self.stderr_file )
```

Here is the `at_now` method:

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

from job_submit import job_submit

class at_now( job_submit ):
    """
Submit the task job script to the simple 'at' scheduler. The 'atd' daemon
service must be running.
    """
    COMMAND_TEMPLATE = "echo \"%s 1>%s 2>%s\" | at now"
    def construct_jobfile_submission_command( self ):
        command_template = self.job_submit_command_template
        if not command_template:
            command_template = self.COMMAND_TEMPLATE
        self.command = command_template % ( self.jobfile_path,
                                            self.stdout_file,
                                            self.stderr_file )
```

Finally, even the `loadleveler` method is quite simple:

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

from job_submit import job_submit

class loadleveler( job_submit ):
    """
Minimalist loadleveler job submission.
    """
    COMMAND_TEMPLATE = "llsubmit %s"

    def set_directives( self ):
        self.directive_prefix = "# @ "
        self.final_directive  = "# @ queue"

        defaults = {}
        defaults[ 'job_name' ] = self.task_id

        defaults[ 'output'   ] = self.stdout_file
        defaults[ 'error'    ] = self.stderr_file
        defaults[ 'shell'    ] = '/bin/ksh'

        # In case the user wants to override the above defaults:
        for d in self.directives:
            defaults[ d ] = self.directives[ d ]
        self.directives = defaults

    def construct_jobfile_submission_command( self ):
        command_template = self.job_submit_command_template
        if not command_template:
            command_template = self.COMMAND_TEMPLATE
        self.command = command_template % ( self.jobfile_path )
```

To use your new method, save it in a source file with the same name as the job submission class (see examples above), install it in the cylc source tree,

```
$CYLC_DIR/lib/cylc/job_submission/MyNewJobSubmitMethod.py
```

and, in the spec file `$CYLC_DIR/conf/suiterc.spec`, add its name to the list of allowed values for the suite.rc *job submission method* configuration items at suite level and in the tasks section.

11 Other Topics In Brief

For more help on any topic, see cylc command line help (which is comprehensive, and is included verbatim in Section B); the suite.rc reference (Section A); the gcylc help menus; and the *Quick Start Guide* (Section 7); and all of the examples suites installed into the central suite database.

- The difference between cold-, warm-, raw-, and re-starting, a suite (see `cylc run help`)

- Intervening in suites, e.g. stopping, removing, inserting tasks; (see `cylc control help`)
- Interrogating suites and tasks (`cylc info help`, `cylc show help`, and `cylc discovery help`)
- Understanding cylc suite evolution, particularly in catch up operation (the *Quick Start Guide* will also help here, along with the cylc example suites, and running your real suites in simulation mode)
- suite security - use of secure passphrases (this is trivial to configure, see documentation of the *use secure passphrase* configuration item in the *Suite.rc Reference*, Appendix A)
- automatic state dump backups, named pre-intervention state dumps (mentioned in the *Quick Start Guide*; watch the suite log after intervening in a suite)
- centralized alerting and timeouts (see documentation of *task event hooks* in the *Suite.rc Reference*, Appendix A)
- Recursive purge - this is a powerful suite intervention but you need to understand how it works before using it. See `cylc purge help` for details.
- Handling spin-up processes via temporary tasks and adding prerequisites on-the-fly - see `cylc depend --help`, and note that when you insert a task into a running suite (a) the initial cycle time can be in the past; and (b) you can give a final cycle time after which the task will be eliminated from the suite.
- Sub-suites - running another suite inside a task:

```
[runtime]
    [[foo]]
        command scripting = "cylc run SUITE $CYLC_TASK_CYCLE_TIME --until=$CYLC_TASK_CYCLE_TIME"
```

12 Suite Design Principles

12.1 Make Fine-Grained Suites

A suite can contain a small number of large, internally complex tasks; a large number of small, simple tasks; or anything in between. Cylc can easily handle a large number of tasks, however, so there are definite advantages to fine-graining:

- a more modular and transparent suite.
- better functional parallelism (multiple tasks running at the same time).
- faster debugging and failure recovery: rerun just the tasks(s) that failed.
- code reuse: similar tasks can often call the same script or command with differing task-specific input parameters (consider tasks that move files around, for example).

12.2 Make Tasks Rerunnable

It should be possible to rerun a task by simply resubmitting it for the same cycle time. In other words, failure at any point during execution of a task should not render a rerun impossible by corrupting the state of some internal-use file, or whatever. It's difficult to overstate the usefulness of being able to rerun the same task multiple times, either outside of the suite with `cylc submit`, or by retriggering it within the running suite, when debugging a problem.

12.3 Make Models Rerunnable

If a warm-cycled model simply overwrites its restart files in each run, the only cycle that can subsequently run is the next one. This is dangerous because if, accidentally or otherwise, the task runs for the wrong cycle time, its restart files will be corrupted such that the correct cycle can no longer run (probably necessitating a cold-start). Instead, consider organising restart files by cycle time, through a file or directory naming convention, and keep them in a simple rolling archive (cylc's filename templating and housekeeping utilities can easily do this for you). Then, given availability of any external inputs, you can easily rerun the task for any cycle still in the restart archive.

12.4 Limit Previous-Instance Dependence

Cylc does not require that successive instances of the same task run sequentially. In order to task advantage of this and achieve maximum functional parallelism whenever the opportunity arises (usually when catching up from a delay) you should ensure that tasks that in principle do not depend on their own previous instances (the vast majority of tasks in most suites, in fact) do not do so in practice. In other words, they should be able to run as soon as their prerequisites are satisfied regardless of whether or not their predecessors have finished yet. This generally just means ensuring that all file I/O contains the generating task's cycle time in the file or directory name so that there is no interference between successive instances. If this is difficult to achieve in particular cases, however, you can declare the offending tasks to be *sequential*.

12.5 Put Task Cycle Time In All Output File Paths

Having all filenames, or perhaps the names of their containing directories, stamped with the cycle time of the generating task greatly aids in managing suite disk usage, both for archiving and cleanup. It also enables the aforementioned task rerunnability recommendation by avoiding overwrite of important files from one cycle to the next. Cylc has powerful utilities for cycle time offset based filename templating and housekeeping.

12.5.1 Use Cylc's Cycle Time Filename Template Utility

The command line utility program `cylc template` determines filenames based on a template string containing YYYYMMDDHH (or variations thereof) by substituting the current cycle time, or some offset from it, into the template. This can be used in the suite.rc environment sections, or in task implementation scripts if necessary, to instantly generate cycle time appropriate filenames for any purpose in the suite.

See `cylc util template help` for more information.

12.6 How To Manage Input/Output File Dependencies

Dependencies between tasks usually, though not always, take the form of files generated by one task that are used by other tasks. It is possible to manage these files across a suite without hard wiring I/O locations and therefore comprising suite flexibility and portability.

- **Use A Common I/O Workspace**
  For small suites you may be able to have all tasks read and write from a common workspace, thereby avoiding the need to move common files around. You should be able to define the workspace location once in the suite.rc file rather than hard wiring it into the task implementations.
- **Add Connector Tasks To The Suite**
  Tasks can be added to a suite to move files from A's output directory to B's input directory, and so on. These connector tasks may all be able to call the same file transfer script or command, with differing input parameters defined in the suite.rc file.

- **Dynamic Configuration Of I/O Paths**
  Whether or not your suite uses a single common workspace, passing common I/O paths to tasks via variables defined once in the suite.rc file should allow you to avoid using connector tasks at all, except where it is necessary to transfer files between machines, or similar.

12.7 Use Generic Task Scripts

If your suite contains multiple logically distinct tasks that actually have similar functionality (e.g. for moving files around, or for generating similar products from the output of several similar models) have the corresponding cylc tasks all call the same command, script, or executable - just provide different input parameters via the task command scripting and/or execution environment, in the suite.rc file.

12.8 Make Suites Portable

If every task in a suite is configured to put its output under `$HOME` (i.e. the environment variable, literally, not the explicit path to your home directory; and similarly for temporary directories, etc.) then other users will be able to copy the suite and run it immediately, after merely ensuring that any external input files are in the right place.

For the ultimate in portability, construct suites in which all task I/O paths are dynamically configured to be user and suite (registration) specific, e.g.

```
$HOME/output/$CYLC_SUITE_REG_PATH
```

(these variables are automatically exported to the task execution environment by cylc - see *Task Execution Environment*, Section 8.4.4). Then you can run multiple instances of the suite at once (even under the same user account) without changing anything, and they will not interfere with each other.

You can test changes to a portable suite safely by making a quick copy of it in a temporary directory, then modifying and running the test copy without fear of corrupting the output directories, suite logs, and suite state, of the original.

12.9 Make Tasks As Self-Contained As Possible

Where possible, no task should rely on the action of another task, except for the prerequisites embodied in the suite dependency graph that it has no choice but to depend on. If this rule is followed, your suite will be as flexible as possible in terms of being able to run single tasks, or subsets of the suite, whilst debugging or developing new features.[19] For example, every task should create its own output directories if they do not already exist, instead of assuming their existence due to the action of some another task; then you will be able to run single tasks without having to manually create output directories first.

```
# manual task scripting:
  # 1/ create $OUTDIR if it doesn't already exist:
  mkdir -p $OUTDIR
  # 2/ create the parent directory of $OUTFILE if it doesn't exist:
  mkdir -p $( dirname $OUTFILE )

# OR using the cylc checkvars utility:
  # 1/ check vars are defined, and create directories if necessary:
  cylc util checkvars -c OUTDIR1 OUTDIR2 #...
  # 2/ check vars are defined, and create parent dirs if necessary:
  cylc util checkvars -p OUTFILE1 OUTFILE2 #...
```

[19]The `cylc submit` command runs a single task exactly as its suite would, in terms of both job submission method and execution environment.

12.10 Make Suites As Self-Contained As Possible

The only compulsory content of a cylc suite definition directory is the suite.rc file (and you'll almost certainly have a suite `bin` sub-directory too). However, you can store whatever you like in a suite definition directory;[20] other files there will be ignored by cylc but suite tasks can access them via the `$CYLC_SUITE_DEF_PATH` variable that cylc automatically exports into the task execution environment. Disk space is cheap - if all programs, ancillary files, control files (etc.) required by the suite are stored in the suite definition directory instead of having the suite reference external build directories (etc.), you can turn the directory into a revision control repository and be virtually assured of the ability to exactly reproduce earlier versions as required, regardless of suite complexity.

12.11 Orderly Product Generation?

Correct scheduling is not equivalent to "orderly generation of products by cycle time". Under cylc, a product generation task will trigger as soon as its prerequisites are satisfied (i.e. when its input files are ready, generally) regardless of whether other tasks with the same cycle time have finished or have yet to run. If your product delivery or presentation system demands that all products for one cycle time are uploaded (or whatever) before any from the next cycle, then be aware that this may be quite inefficient if your suite is ever faced with catching up from a significant delay or running over historical data.

If you must, however, you can introduce artificial dependencies into your suite to ensure that the final products never arrive out of sequence. One way of doing this would be to have a final "product upload" task that depends on completion of all the real product generation tasks at the same cycle time, and then declare it to be sequential.

12.12 Clock-triggered Tasks Wait On External Data

All tasks in a cylc suite know their own private cycle time, but most don't care about the wall clock time - they just run when their prerequisites are satisfied. The exception to this is *clock-triggered* tasks, which wait on a wall clock time expressed as an offset from their own cycle time, in addition to any other prerequisites. The usual purpose of these tasks is to retrieve real time data from the external world, triggering at roughly the expected time of availability of the data. Triggering the task at the right time is up to cylc, but the task itself should go into a check-and-wait loop in case the data is delayed; only on successful detection or retrieval should the task report success and then exit (or perhaps report failure and then exit if the data has not arrived by some cutoff time).

12.13 Do Not Treat Real Time Operation As Special

Cylc suites, without modification, can handle real time and delayed operation equally well.

In real time operation clock-triggered tasks constrain the behaviour of the whole suite, or at least of all tasks downstream of them in the dependency graph.

In delayed operation (due to an actual delay in an operational suite or because you're running an historical case study) clock-triggered tasks will not constrain the suite at all, and cylc's multi-cycling abilities come to the fore, because their trigger times have already passed. But if a clock-triggered task happens to catch up to the wall clock, it will automatically wait again. In this way a cylc suite naturally and seamlessly transitions between delayed and real time operation as required.

[20] If you copy a suite using cylc commands or gcylc, the entire suite definition directory will be copied.

A Suite.rc Reference

This appendix documents all legal entries in a suite.rc file. Most suites will only need to explicitly configure a few of these items - many of them have sensible default values, some may only be needed for critical operational suites (e.g. secure passphrases), and some are primarily used for cylc development. In general your suite.rc files shouldn't be a lot more complicated than those of the cylc example suites.

See also *Suite Definition - Suite.rc Overview* (Section 8.2) for a descriptive overview of suite.rc files.

A.1 Top Level Items

The only top level configuration items at present are the suite title and description.

A.1.1 title

The suite title is displayed in the gcylc suite database window. It can also be retrieved from a suite at run time with `cylc show` (or use `cylc get-config`).

- *type:* string
- *default:* "No title provided"

A.1.2 description

The suite description can be retrieved by gcylc right-click menu. It can also be retrieved from a suite at run time with `cylc show` (or use `cylc get-config`).

- *type:* string
- *default:* "No description provided"

A.2 [cylc]

This section is for suite configuration that is not specifically task-related.

A.2.1 [cylc] → UTC mode

Cylc runs off the suite host's system clock by default. This item allows you to run the suite in UTC even if the system clock is set to local time. Clock-triggered tasks will trigger when the current UTC time is equal to their cycle time plus offset; other time values used, reported, or logged by cylc will also be in UTC.

- *type:* boolean
- *default:* False

A.2.2 [cylc] → simulation mode only

This prevents a suite from running in real mode - use for demo suites created by copying real suites out of their normal operating environment.

- *type:* boolean
- *default:* False

A.2.3 [cylc] → use secure passphrase

Critical operational suites can be made to ignore commands unless the originating user account has a special passphrase (the same one used by the suite owner at startup) with secure permissions (as for ssh keys) in the file `$HOME/.cylc/security/$CYLC_SUITE_REG_NAME` (remotely hosted tasks also need the passphrase in their host accounts). The passphrase itself is never transferred across the network (a secure MD5 checksum is). Note that cylc's normal owner-only suite access is implemented by means of a simple username comparison, which could in principle be subverted by a malicious user copying and modifying the cylc code base. Secure passphrases, on the other hand, should guarantee suite security so long as your user accounts aren't breached.

- *type:* boolean
- *default:* False

A.2.4 [cylc] → [[logging]]

This section configures cylc's logging functionality, which records time-stamped events to a special log file.

A.2.4.1 [cylc] → [[logging]] → directory

The cylc log and its backups are stored in this directory. If you change the directory make sure it remains suite-specific by using suite identity environment variables in the path.

- *type:* string (directory path, may contain environment variables)
- *default:* `$HOME/cylc-run/$CYLC_SUITE_REG_NAME/log/suite`

A.2.4.2 [cylc] → [[logging]] → roll over at start-up

Suite logs roll over (start anew) automatically when they reach a certain size - currently hard-wired to 1MB in `$CYLC_DIR/lib/cylc/pimp_my_logger.py`. They can also be rolled automatically whenever a suite is started or restarted.

- *type:* boolean
- *default:* True

A.2.5 [cylc] → [[state dumps]]

State dump files allow cylc to restart suites from previous states of operation.

A.2.5.1 [cylc] → [[state dumps]] → directory

The rolling archive of suite state dump files, backups of the default state dump, and any special pre-intervention state dumps, are stored under this directory. If you change this directory make sure it remains suite-specific by using suite identity environment variables in the path.

- *type:* string (directory path, may contain environment variables)
- *default:* `$HOME/cylc-run/$CYLC_SUITE_REG_NAME/state`

A.2.5.2 [cylc] → [[state dumps]] → number of backups

This is the length, in number of changes, of the automatic rolling archive of state dump files that allows you to restart a suite from a previous state. Every time a task changes state cylc updates the state dump and rolls previous states back one on the archive. You'll probably only ever need the latest (most recent) state dump, which is automatically used in a restart, but any previous state still in the archive can be used. Additionally, special labeled state dumps are written out prior to actioning any suite intervention - their filenames are logged by cylc.

- *type:* integer (≥ 1)
- *default:* 10

A.2.6 [cylc] → [[lockserver]]

The cylc lockserver brokers suite and task locks on the network (these are somewhat analagous to traditional local *lock files*). It prevents multiple instances of a suite or task from being invoked at the same time (via scheduler instances or `cylc submit`).

See `cylc lockserver --help` for how to run the lockserver, and `cylc lockclient --help` for occasional manual lock management requirements.

A.2.6.1 [cylc] → [[lockserver]] → enable

The lockserver is currently disabled by default. It is intended mainly for operational use.

- *type:* boolean
- *default:* False

A.2.6.2 [cylc] → [[lockserver]] → simultaneous instances

By default the lockserver prevents multiple simultaneous instances of a suite from running even under different registered names. But allowing this may be desirable if the I/O paths of every task in the suite are dynamically configured to be suite specific (and similarly for the suite state dump and logging directories, by using suite identity variables in their directory paths). Note that *the lockserver cannot protect you from running multiple distinct* **copies** *of a suite simultaneously.*

- *type:* boolean
- *default:* False

A.2.7 [cylc] → [[environment]]

Variables defined here are exported to the environment in which cylc itself runs, for possible use by processes spawned directly by cylc (e.g. event hook scripts). *Note that these variables are only available to executing tasks if you happen to choose a local direct job submission method - use task runtime environments for this purpose.*

A.2.7.1 [cylc] → [[environment]] → __VARIABLE__

Replace __VARIABLE__ with an environment variable assignment and repeat for as many local environment variables as you need. Values may contain local environment variables. General shell variable expansion expressions will not be expanded (these are legal in task runtime environments, however).

- *type:* string (may contain local environment variables).
- *default:* (none)
- *examples:*
  - `FOO = $HOME/foo`

A.2.8 [cylc] → [[simulation mode]]

Items specific to running suites in simulation mode.

A.2.8.1 [cylc] → [[simulation mode]] → clock rate

This determines the speed at which the simulation mode clock runs, in real seconds per simulated hour. A value of 10, for example, means it will take 10 real seconds to simulate one hour of operation.

- *type:* integer (≥ 0, real seconds per simulated hour)
- *default:* 10

A.2.8.2 [cylc] → [[simulation mode]] → clock offset

The clock offset determines the initial time on the simulation clock, at suite startup, relative to the initial cycle time. An offset of 0 simulates real time operation; greater offsets simulate catch up from a delay and subsequent transition to real time operation.

- *type:* integer (≥ 0, hours behind initial cycle time)
- *default:* 24

A.2.8.3 [cylc] → [[simulation mode]] → command scripting

The command scripting to execute for all tasks when running in simulation mode.

- *type:* string (scripting valid in job submission shell; triple quote for multiple lines)
- *default:* `echo SIMULATION MODE $CYLC_TASK_ID; sleep 10; echo BYE`

A.2.8.4 [cylc] → [[simulation mode]] → [[[job submission]]]

Configure job submission for simulation mode.

A.2.8.4.1 [cylc] → [[simulation mode]] → [[[job submission]]] → method

The job submission method to use for all tasks in simulation mode. Any available method can be used but the default is probably sufficient for simulation mode.

- *type:* string (a job submission method name - see Section A.4.1.7.1)
- *default:* `background`

A.2.8.5 [cylc] → [[simulation mode]] → [[[event hooks]]]

Configure event hooks for simulation mode.

A.2.8.5.1 [cylc] → [[simulation mode]] → [[[event hooks]]] → enable

Currently event hooks are disabled by default in simulation mode. They can be enabled in order to test automated alerts, for example, without running the real suite tasks, but be aware that timeouts will be relative to the accelerated simulation mode clock which by default runs very quickly.

- *type:* boolean
- *default:* False

A.3 [scheduling]

This section allows cylc to determine when tasks are ready to run.

A.3.1 [scheduling] → initial cycle time

At startup each cycling task (unless specifically excluded under [special tasks]) will be inserted into the suite with this cycle time, or with the closest subsequent valid cycle time for the task. Note that whether or not *cold-start tasks*, specified under [special tasks], are inserted, and in what state they are inserted, depends on the start up method - cold, warm, or raw. If this item is provided you can override it on the command line or in the gcylc suite start panel.

- *type:* integer (YYYYMMDDHH)
- *default:* (none)

A.3.2 [scheduling] → final cycle time

Cycling tasks are held (i.e. not allowed to spawn a successor) once they pass the final cycle time, if one is specified. Once all tasks have achieved this state the suite will shut down. If this item is provided you can override it on the command line or in the gcylc suite start panel.

- *type:* integer (YYYYMMDDHH)
- *default:* (none)

A.3.3 [scheduling] → runahead limit

A cycling task spawns a successor either when it starts running or, for sequential tasks, when it finishes. If the successor's cycle time is ahead of the oldest non-failed task by more than the runahead limit it will be put into special "runahead held" state until other tasks catch up sufficiently. This prevents quick-running unconstrained tasks in historical trials (i.e. sustained catch up operation) from running off into the future and potentially swamping the system. In real time operation the runahead limit is of little consequence because the suite will be constrained by its clock-triggered tasks (the limit must be long enough to cover the range of tasks present in the suite, however; a task that only runs once per day, for instance, needs to spawn 24 hours ahead). Failed tasks, which are not automatically removed from a suite, are ignored when computing the runahead limit (but tasks that can't run because they depend on a failed task are not ignored).

- *type:* integer (≥ 0, hours)
- *default:* 24

A.3.4 [scheduling] → [[special tasks]]

This section identifies any tasks with special behaviour. By default (i.e. non "special" behaviour) tasks submit as soon as their prerequisites are satisfied, and they spawn a successor at the next valid cycle time for the task as soon as they enter the running state[21]

A.3.4.1 [scheduling] → [[special tasks]] → clock-triggered

Clock-triggered tasks wait on a wall clock time specified as an offset *in hours* relative to their own cycle time, in addition to any dependence they have on other tasks. *Generally speaking, only tasks that wait on external real time data need to be clock-triggered.*

- *type:* list of tasknames with offsets (hours, positive or negative)
- *default:* (none)
- *example:* `clock-triggered = foo(1.5), bar(2.25)`

Clock-triggered tasks cannot be triggered manually prior to their trigger time. Workaround: just run the task alone with `cylc submit` then set the in-suite task's state to 'succeeded' so that downstream tasks can trigger off it.

A.3.4.2 [scheduling] → [[special tasks]] → start-up

Start-up tasks are one-off tasks that are only used when a suite is cold started (i.e. starting up without assuming any previous cycle). They can be used to clean out or prepare a suite workspace, for example, before other tasks run.

- *type:* list of task names
- *default:* (none)

A.3.4.3 [scheduling] → [[special tasks]] → cold-start

A cold-start task (or possibly a sequence of them) is used to satisfy the dependence of an associated task with the same cycle time, on outputs from a previous cycle - when those outputs are not available. The primary use for this is to cold-start a warm-cycled forecast model that normally depends on restart files (e.g. model background fields) generated by its previous forecast, when there is no previous forecast. This is required when cold-starting the suite, but cold-start tasks can also be inserted into a running suite to restart a model that has had to skip some cycles after running into problems (e.g. critical inputs not available). Cold-start tasks can invoke real cold-start processes, or they can just be dummy tasks that represent some external process that has to be completed before the suite is started.

- *type:* list of task names
- *default:* (none)

A.3.4.4 [scheduling] → [[special tasks]] → sequential

By default, a task spawns a successor as soon as it starts running, so that successive instances of the same task can run in parallel if the opportunity arises (i.e. if their prerequisites happen to be satisfied before their predecessor has finished). *Sequential tasks*, however, will not spawn a successor until they have finished successfully. This should be used for (a) *tasks that cannot run in parallel with their own previous instances* because they would somehow interfere with

[21] Spawning any earlier than this brings no advantage in terms of functional parallelism.

each other (use cycle time in all I/O paths to avoid this); and (b) *Warm cycled forecast models that write out restart files for multiple cycles ahead* (exception: see "explicit restart outputs" below).[22]

- *type:* list of task names
- *default:* (none)

A.3.4.5 [scheduling] → [[special tasks]] → one-off

One-off tasks do not spawn a successor; they run once and are then removed from the suite when they are no longer needed. *Start-up* and *cold-start* tasks are automatically one-off tasks and do not need to be listed here.

- *type:* list of task names
- *default:* (none)

A.3.4.6 [scheduling] → [[special tasks]] → explicit restart outputs

This is only required in the event that you need a warm cycled forecast model to start at the instant its restart files are ready (if other prerequisites are satisfied) *even if its previous instance has not finished yet*. If so, the model task has to depend on special output messages emitted by the previous instance as soon as its restart files are ready, instead of just on the previous instance finishing. *Tasks in this category must define special restart output messages containing the word "restart"*, in [runtime] → [[TASK]] → [[[outputs]]] - see Section 9.4.2.

- *type:* list of task names
- *default:* (none)

A.3.4.7 [scheduling] → [[special tasks]] → exclude at start-up

Any task listed here will be excluded from the initial task pool (this goes for suite restarts too). If an *inclusion* list is also specified, the initial pool will contain only included tasks that have not been excluded. Excluded tasks can still be inserted at run time. Other tasks may still depend on excluded tasks if they have not been removed from the suite dependency graph, in which case some manual triggering, or insertion of excluded tasks, may be required.

- *type:* list of task names
- *default:* (none)

A.3.4.8 [scheduling] → [[special tasks]] → include at start-up

If this list is not empty, any task *not* listed in it will be excluded from the initial task pool (this goes for suite restarts too). If an *exclusion* list is also specified, the initial pool will contain only included tasks that have not been excluded. Excluded tasks can still be inserted at run time. Other tasks may still depend on excluded tasks if they have not been removed from the suite dependency graph, in which case some manual triggering, or insertion of excluded tasks, may be required.

[22] This is because you don't want Model(T) waiting around to trigger off Model(T-12) if Model(T-6) has not finished yet. If Model is forced to be sequential this can't happen because Model(T) won't exist in the suite until Model(T-6) has finished. But if Model(T-6) fails, it can be spawned-and-removed from the suite so that Model(T) can *then* trigger off Model(T-12), which is the correct behaviour.

- *type:* list of task names
- *default:* (none)

A.3.5 [scheduling] → [[dependencies]]

The suite dependency graph is defined under this section. You can plot the dependency graph as you work on it, with `cylc graph` or by right clicking on the suite in gcylc. See also Section 8.3.

A.3.5.1 [scheduling] → [[dependencies]] → graph

The dependency graph for any one-off asynchronous (non-cycling) tasks in the suite goes here. This can be used to construct a suite of one-off tasks (e.g. build jobs and related processing) that just completes and then exits, or an initial suite section that completes prior to the cycling tasks starting (if you make the first cycling tasks depend on the last one-off ones). But note that synchronous *start-up* tasks can also be used for the latter purpose. See Section A.3.5.2.1 below for graph string syntax, and Section 8.3.

- *type:* string
- *example:* (see Section A.3.5.2.1 below)

A.3.5.2 [scheduling] → [[dependencies]] → [[[__VALIDITY__]]]

Replace VALIDITY with a comma-separated list of integer hours, $0 \leq H \leq 23$, defining the valid cycle times for the subsequent graph of cycling tasks, or multiple such subsections as required for different dependencies at different hours; or with 'ASYNCID:*pattern*', where *pattern* is a regular expression that matches an asynchronous task ID, for each graph of repeating asynchronous tasks.

- *examples:*
  - (cycling tasks) `[[0,6,12,18]]`
  - (repeating asynchronous tasks) `[[ASYNCID:SAT-\d+]]`
- *default:* (none)

A.3.5.2.1 [scheduling] → [[dependencies]] → [[[__VALIDITY__]]] → graph

The dependency graph for the specified validity section (described just above) goes here. Syntax examples follow; see also Section 8.3.

- *type:* string
- *example:*

```
graph = """
   foo => bar => baz & waz    # baz and waz both trigger off bar
   baz:out1 => faz            # faz triggers off an internal output of baz
   ColdFoo | foo(T-6) => foo # cold-start or restart for foo
   X:fail => Y                # Y triggers if X fails
   X => !Y                    # Y suicides if X succeeds
   X | X:fail => Z            # Z triggers if X finishes or fails
   (A | B & C ) | D => foo    # general conditional triggers
   # comment
   # A python expression to generate dependencies automatically:
   Python:list( "m" + str(i) + '=>' + "p" + str(i) for i in range(1,7))
   """
```

- *default:* (none)

A.3.5.2.2 [scheduling] → [[dependencies]] → [[[__VALIDITY__]]] → daemon

For [[[ASYNCID:pattern]]] validity sections only, list *asynchronous daemon* tasks by name. This item is located here rather than under [scheduling] → [[special tasks]] because a damon task is associated with a particular asynchronous ID.

- *type:* list of task names
- *default:* (none)

A.4 [runtime]

This section defines how, where, and what to execute when tasks are ready to run. Runtime subsections define an inheritance hierarchy of *namespaces*, each of which represents a family of tasks or an individual task, as described in Section 8.4).

A.4.1 [runtime] → [[__NAME__]]

Replace __NAME__ with a namespace name, or a comma separated list of names, or a Python expression that generates a list of names; and repeat as needed to define all tasks in the suite.

Namespace names may contain letters, digits, underscores, and hyphens. They may not contain colons (which would preclude use of directory paths involving the registration name in `$PATH` variables). They may not contain the '.' character (it will be interpreted as the namespace hierarchy delimiter, separating groups and names). *Task names should not be hardwired into task implementations.* Rather, task and suite identity should be extracted portably from the task execution environment supplied by cylc (Section 8.4.4) - then to rename a task you can just change its name in the suite.rc file.

A namespace represents a family if other namespaces inherit from it; it represents a task if it is a leaf on the inheritance tree (i.e. no other namespaces inherit from it).

- *legal values:*
  - `[[foo]]`
  - `[[foo, bar, baz]]`
  - `[['Python:list( "m"+ str(i) for i in range(0,25))']]`

If multiple names are listed, the subsequent namespace configuration items apply to each member, but any instance of `$(TASK)` in any value will be replaced by cylc with the actual namespace name. This can be used to define many tasks that are almost identical.

All namespaces inherit initially from *root*, which you can explicitly configure to override or provide default runtime settings for all tasks in the suite.

A.4.1.1 [runtime] → [[__NAME__]] → inherit

Specify here the namepace from which this namespace should inherit all of its runtime configuration; specific items can then be overridden as required. Note that many of the available items are left undefined even in the root namespace.

- *type:* string (another namespace name)
- *default:* root

A.4.1.2 [runtime] → [[__NAME__]] → description

A description of this namespace, retrievable from running tasks via `cylc show`.

- *type:* string
- *root default:* "No description provided"

A.4.1.3 [runtime] → [[__NAME__]] → command scripting

The scripting to execute when the associated task is ready to run. The value can be a single command, or a multiline string of scripting, or a list of commands or multiline strings. If a list is provided the task will automatically resubmit with the second list member if the task fails, and then the third if the task fails again, and so on until no command scripting list members remain - this is one of the ways you can do automated failure recovery in cylc.

- *type:* (a list of) strings (each)
- *root default:* `echo DUMMY $CYLC_TASK_ID; sleep 10; echo BYE`

A.4.1.4 [runtime] → [[__NAME__]] → pre-command scripting

Scripting to be executed immediately *before* the command scripting. This would typically be used to add scripting to every task in a family (for individual tasks you could just incorporate the extra commands into the main command scripting). See also *post-command scripting*, below.

- *type:* string, or a list of strings
- *default:* (none)
- *example:*

```
pre-command scripting = """
  . $HOME/.profile
  echo Hello from suite ${CYLC_SUITE_REG_NAME}!"""
```

A.4.1.5 [runtime] → [[__NAME__]] → post-command scripting

Scripting to be executed immediately *after* the command scripting. This would typically be used to add scripting to every task in a family (for individual tasks you could just incorporate the extra commands into the main command scripting). See also *pre-command scripting*, above.

- *type:* string, or a list of strings
- *default:* (none)

A.4.1.6 [runtime] → [[__NAME__]] → manual completion

If a task's initiating process detaches and exits before task processing is finished then cylc cannot arrange for the task to automatically signal when it has succeeded or failed. In such cases you must use this configuration item to tell cylc not to arrange for automatic completion messaging, and insert some minimal completion messaging yourself in appropriate places in the task implementation (see Section 9.4.3).

- *type:* boolean
- *default:* False

A.4.1.7 [runtime] → [[__NAME__]] → [[[job submission]]]

This is where to configure the means by which cylc submits task job scripts to run.

A.4.1.7.1 [runtime] → [[__NAME__]] → [[[job submission]]] → method

Cylc currently has a handful of defined job submission methods. *Task Job Submission* (Section 10) explains how job submission works, and how to define new methods.

- *type:* string
- *legal values:*
  - `background` - direct background execution
  - `at_now` - the rudimentary Unix 'at' scheduler
  - `loadleveler` - loadleveler, generic (with directives defined in the suite.rc file)
  - `ll_ecox` - loadleveler, customized for EcoConnect triplicate environment at NIWA
  - `ll_raw` - loadleveler, for existing job scripts
- *default:* `background`

A.4.1.7.2 [runtime] → [[__NAME__]] → [[[job submission]]] → command template

This allows you to override the actual command used by the chosen job submission method. The template's first %s will be substituted by the job file path. Where applicable the second and third %s will be substituted by the paths to the stdout and stderr files.

- *type:* string
- *legal values:* a string template
- *example:* `llsubmit %s`

A.4.1.7.3 [runtime] → [[__NAME__]] → [[[job submission]]] → shell

This is the shell used to interpret the job script submitted by cylc when a task is ready to run. *It has no bearing on the shell used in task implementations.* Command scripting and suite environment variable assignment expressions must be valid for this shell. The latter is currently hardwired into cylc as `export item=value` - valid for both bash and ksh because `value` is entirely user-defined - but cylc would have to be modified slightly to allow use of the C shell.

- *type:* string
- *root default:* `/bin/bash`

A.4.1.7.4 [runtime] → [[__NAME__]] → [[[job submission]]] → log directory

This is where task job scripts, and the stdout and stderr logs for local tasks, are written. The directory path may contain environment variables, including suite identity variables to make the path suite-specific (as the default value does). The job script filename is constructed, just before job submission, from the task ID and *seconds since epoch*, and then *.out* and *.err* are appended to construct the stdout and stderr log names, respectively. These filenames are thus unique even if a task gets retriggered, and yet will be correctly time ordered if the log directory is listed. The filenames are also recorded by each task proxy for access via cylc commands and the suite control GUIs.

- *type:* string (directory path, may contain environment variables)
- *default:* `$HOME/cylc-run/$CYLC_SUITE_REG_NAME/log/job`

A.4.1.7.5 [runtime] → [[__NAME__]] → [[[job submission]]] → work directory

Each task runs in its own working directory. The work directory is created on the fly by the task job script prior to executing the task command scripting. For non-detaching tasks the work directory is removed, if it is empty, before the job script exits. It can be accessed by tasks at run time via the environment variable `$CYLC_TASK_WORK_PATH`.

- *type:* string (directory path, may contain environment variables)
- *default:* `$CYLC_SUITE_DEF_PATH/work/$CYLC_TASK_ID`

A.4.1.7.6 [runtime] → [[__NAME__]] → [[[job submission]]] → share directory

This is intended as a shared data directory for multiple tasks. As such it should be defined at root level, if for all tasks; or in a family namespace, if for a subset of tasks; however, it can be made unique for every task if you like. The share directory is created on the fly by the task job script prior to executing the task command scripting. It can be accessed by tasks at run time via the environment variable `$CYLC_SUITE_SHARE_PATH`.

- *type:* string (directory path, may contain environment variables)
- *default:* `$CYLC_SUITE_DEF_PATH/share`

A.4.1.8 [runtime] → [[__NAME__]] → [[[remote]]]

Remote hosting configuration, for tasks that need to run on platforms other than the suite host. If a remote host is specified cylc will attempt to execute the task on that host by passwordless ssh. Relevant task scripts and executables, cylc itself, and possibly all or part of the suite definition directory, must be installed on the remote host. Passwordless ssh must be configured between the local suite owner and remote task owner accounts.

A.4.1.8.1 [runtime] → [[__NAME__]] → [[[remote]]] → host

The remote host for this task or family.

- *type:* string (a valid hostname on the network)
- *default:* (none)

A.4.1.8.2 [runtime] → [[__NAME__]] → [[[remote]]] → owner

User name under which to access the remote host, via passwordless ssh.

- *type:* string (a valid username on the remote host)
- *default:* (none)

A.4.1.8.3 [runtime] → [[__NAME__]] → [[[remote]]] → cylc directory

The path to the cylc installation on the remote host (this is a required item for tasks that define a remote host).

- *type:* string (a valid directory path on the remote host)
- *default:* (none)

A.4.1.8.4 [runtime] → [[__NAME__]] → [[[remote]]] → suite definition directory

The path to the suite definition directory on the remote host, needed if remote tasks require access to files stored there (via `$CYLC_SUITE_DEF_PATH`) or in the suite bin directory (via `$PATH`). If this item is not defined, the local suite definition directory path will be assumed, with the suite owner's home directory, if present, replaced by `'$HOME'` for interpretation on the remote host.

- *type:* string (a valid directory path on the remote host)
- *default:* (local suite definition path with `$HOME` replaced)

A.4.1.8.5 [runtime] → [[__NAME__]] → [[[remote]]] → remote shell template

A template for the remote shell command for a submitting a remote task. The template's first %s will be substituted by the remote user@host.

- *type:* string (a string template)
- *root default:* `ssh -oBatchMode=yes %s`

A.4.1.8.6 [runtime] → [[__NAME__]] → [[[remote]]] → log directory

This log directory is used for the stdout and stderr logs of remote tasks. The directory will be created on the fly if necessary. If not specified, the local log path will be used (see A.4.1.7.4) with the suite owner's home directory path, if present, replaced by `'$HOME'` for interpretation on the remote host. The stdout and stderr log file names are the same as for local tasks, and are recorded for access via gcylc. You could use an event hook script to automatically retrieve remote log files on task completion to make them available as for local task logs.

- *type:* string (a valid directory path on the remote host)
- *default:* (local log path with `$HOME` replaced)

A.4.1.8.7 [runtime] → [[__NAME__]] → [[[remote]]] → work directory

Use this item if you need to override the local task work directory (see A.4.1.7.5). If omitted, the local directory will be used with the suite owner's home directory path, if present, replaced by `'$HOME'` for interpretation on the remote host.

- *type:* string (directory path, may contain environment variables)
- *default:* (local task work path with `$HOME` replaced)

A.4.1.8.8 [runtime] → [[__NAME__]] → [[[remote]]] → share directory

Use this item if you need to override the local share directory (see A.4.1.7.6). If omitted, the local directory will be used with the suite owner's home directory path, if present, replaced by `'$HOME'` for interpretation on the remote host.

- *type:* string (directory path, may contain environment variables)
- *default:* (local task share path with `$HOME` replaced)

A.4.1.9 [runtime] → [[__NAME__]] → [[[event hooks]]]

Cylc can call a nominated script when certain events, such as task failure or timeout, occur. This is intended to facilitate centralized alerting and automated handling of critical events. Event hook scripts can do anything you like, such as send emails or SMS, call pagers, or intervene in the operation of their parent suite with cylc commands. The utility command `cylc [util] email-alert` is a ready made event hook script.

Currently a single script can be nominated to handle a list of chosen events, and the event name is passed on the command line to allow the script to distinguish between events (we originally allowed distinct hook scripts for each event, but this required rather verbose configuration for little gain).

Event hook scripts can be located in the suite bin directory. They are passed the following arguments by cylc:

```
<hook-script> EVENT SUITE TASKID MESSAGE
```

where EVENT is one of the following strings:

- 'submitted' - the task was submitted
- 'started' - the task started running
- 'succeeded' - the task succeeded
- 'submission_failed' - the task failed in job submission
- 'failed' - the task failed
- 'timeout' - the task timed out

MESSAGE, if provided, describes what has happened, and TASKID identifies the offending task (e.g. `TASK%CYCLE` for cycling tasks).

Note that event hook scripts are called by cylc itself, not by the running tasks so if you wish to pass in additional information via the environment, use [cylc] → [[environment]] not the runtime namespace environments.

A.4.1.9.1 [runtime] → [[__NAME__]] → [[[event hooks]]] → events

The list of events to handle, as listed for EVENT just above.

- *type:* list of strings
- *default:* (none)

A.4.1.9.2 [runtime] → [[__NAME__]] → [[[event hooks]]] → script

The script to call when one of the nominated events occurs.

- *type:* string
- *default:* (none)

A.4.1.9.3 [runtime] → [[__NAME__]] → [[[event hooks]]] → submission timeout

If a task has not started the specified number of minutes after it was submitted, the hook script will be called by cylc with 'timeout' as the EVENT argument:

- *type:* float (minutes)
- *default:* (none)

A.4.1.9.4 [runtime] → [[__NAME__]] → [[[event hooks]]] → execution timeout

If a task has not finished the specified number of minutes after it started running, the hook script will be called by cylc with 'timeout' as the EVENT argument:

- *type:* float (minutes)
- *default:* (none)

A.4.1.9.5 [runtime] → [[__NAME__]] → [[[event hooks]]] → reset timer

If you set an execution timeout the timer can be reset to zero every time a message is received from the running task (which indicates the task is still alive) . Otherwise, the task will timeout if it does not finish in the alotted time regardless of incoming messages.

- *type:* boolean
- *default:* False

A.4.1.10 [runtime] → [[__NAME__]] → [[[environment]]]

The user defined execution environment. Variables defined here can refer to cylc suite and task identity variables, which are exported earlier in the task job script, and variable assignment expressions can use cylc utility commands because access to cylc is also configured earlier in the script. See also *Task Execution Environment*, Section 8.4.4.

A.4.1.10.1 [cylc] → [[environment]] → __VARIABLE__

Replace __VARIABLE__ with an environment variable assignment expression, which must be valid in the job submission shell, and repeat for as many environment variables as you need. Order of definition is preserved, so values can refer to previously defined variables.

- *type:* string (any environment variable assignment expression valid in the job submission shell)
- *default:* (none)
- *legal values:* (Section A.4.1.7.3). White space around the '=' is allowed (the `suite.rc` file is not a shell script).
- *examples* for the bash shell:
  - `FOO = $HOME/bar/baz`
  - `BAR = ${FOO}$GLOBALVAR`
  - `BAZ = $(echo "hello world")`
  - `WAZ = ${FOO%.jpg}.png`
  - `NEXT_CYCLE = $( cylc cycletime -a 6 )`
  - `PREV_CYCLE = `cylc cycletime -s 6``
  - `ZAZ = "${FOO#bar}"#<-- QUOTED to escape the suite.rc comment character`

A.4.1.11 [runtime] → [[__NAME__]] → [[[directives]]]

Batch queue scheduler directives, or similar. Whether or not these are used depends on the job submission method. For the loadleveler method, for example, directives are written to the top of the task job script in the format required by loadleveler.

A.4.1.11.1 [cylc] → [[environment]] → __DIRECTIVE__

Replace __DIRECTIVE__ with each directive assignment, e.g. `class = parallel`

- *type:* string
- *default:* (none)

A.4.1.12 [runtime] → [[__NAME__]] → [[[outputs]]]

This section is only required if other tasks need to trigger off specific internal outputs of this task (as opposed to triggering off it finishing). The task implementation must report the specified output messages by calling `cylc task message` when the corresponding real outputs have been completed.

A.4.1.12.1 [cylc] → [[environment]] → __OUTPUT__

Replace __OUTPUT__ with any number of labelled output messages.

- *type:* string (a message containing `$(CYCLE_TIME)` with an optional offset as shown below. Note the round parentheses on `$(CYCLE_TIME)` - this is not a shell variable, although without an offset it does correspond to `$CYLC_TASK_CYCLE_TIME` in the task execution environment.
- *default:* (none)
- *examples:*

```
foo = "sea state products ready for $(CYCLE_TIME)"
bar = "nwp restart files ready for $(CYCLE_TIME+6)"
```

where the item name must match the output label associated with this task in the suite dependency graph, e.g.:

```
[scheduling]
    [dependencies]
        graph = TaskA:foo => TaskB
```

A.5 [visualization]

Configuration of suite graphing and, where explicitly stated, the graph-based suite control GUI.

A.5.1 [visualization] → initial cycle time

The first cycle time to use when plotting the suite dependency graph.

- *type:* integer
- *default:* 2999010100

A.5.2 [visualization] → final cycle time

The last cycle time to use when plotting the suite dependency graph. Typically this should be just far enough ahead of the initial cycle to show the full suite.

- *type:* integer
- *default:* 2999010123

A.5.3 [visualization] → collapsed families

A list of family (namespace) names to be shown in the collapsed state (i.e. the family members will be replaced by a single family node) when the suite is plotted in the graph viewer or the graph-based suite control GUI. This item determines how family groups are shown *initially* in the suite control GUI; subsequently you can use the interactive controls to group and ungroup nodes at will. For the same reason (presence of interactive grouping controls) this item is ignored if the suite is reparsed during graph viewing (other changes to graph styling will be picked up and applied if the graph viewer detects that the suite.rc file has changed).

- *type:* list of family names
- *default:* (none)

A.5.4 [visualization] → use node color for edges

Graph edges (dependency arrows) can be plotted in the same color as the upstream node (task or family) to make paths through a complex graph easier to follow.

- *type:* boolean
- *default:* True

A.5.5 [visualization] → default node attributes

Set the default attributes (color and style etc.) of graph nodes (tasks and families). Attribute pairs must be quoted to hide the internal `=` character.

- *type:* list of quoted `'attribute=value'` pairs
- *legal values:* see graphviz or pygraphviz documentation
- *default:* `'style=unfilled'`, `'color=black'`, `'shape=box'`

A.5.6 [visualization] → default edge attributes

Set the default attributes (color and style etc.) of graph edges (dependency arrows). Attribute pairs must be quoted to hide the internal `=` character.

- *type:* list of quoted `'attribute=value'` pairs
- *legal values:* see graphviz or pygraphviz documentation
- *default:* `'color=black'`

A.5.7 [visualization] → [[node groups]]

Define named groups of graph nodes (tasks and families) that can have attributes assigned to them en masse in the [visualization] → [[node attributes]] section.

A.5.7.1 [visualization] → [[node groups]] → __GROUP__

Replace __GROUP__ with each named group of task or family names.

- *type:* comma separated list of task or family names
- *default:* (none)
- *example:*

```
PreProc = foo, bar
PostProc = baz, waz
```

A.5.8 [visualization] → [[node attributes]]

Here you can assign graph node attributes to specific tasks, or named groups defined in [visualization] → [[node groups]].

A.5.8.1 [visualization] → [[node attributes]] → __NAME__

Replace __NAME__ with any task, family, or group that you want to assign attributes to.

- *type:* list of quoted `'attribute=value'` pairs
- *legal values:* see graphviz or pygraphviz documentation

- *default:* (none)
- *example:* (with reference to the node groups defined above)

```
PreProc = 'style=filled', 'color=blue'
PostProc = 'color=red'
foo = 'style=unfilled'
```

A.5.9 [visualization] → [[run time graph]]

Cylc can generate graphs of dependencies resolved at run time, i.e. what actually triggers off what as the suite runs. This feature is retained mainly for development and debugging purposes. You can use simulation mode to generate run time graphs very quickly.

A.5.9.1 [visualization] → [[run time graph]] → enable

Run time graphing is now disabled by default.

- *type:* boolean
- *default:* False

A.5.9.2 [visualization] → [[run time graph]] → cutoff

New nodes will be added to the run time graph as the corresponding tasks trigger, until their cycle time exceeds the initial cycle time by more than this cutoff, in hours.

- *type:* integer (≥ 0, hours)
- *default:* 24

A.5.9.3 [visualization] → [[run time graph]] → directory

Where to put the run time graph file, `runtime-graph.dot`.

- *type:* string (a valid directory path, may contain environment variables)
- *default:* `$CYLC_SUITE_DEF_PATH/graphing`

A.6 Default Suite Configuration

Cylc provides, via the suite.rc spec file, sensible default values for many configuration items so that users may not need to explicitly configure log directories and the like. The defaults are sufficient, in fact, to define simple test suites by dependency graph alone (command scripting, for example, defaults to printing a simple message, sleeping for ten seconds, and then exiting). The following listing shows all current legal items and any default values:

```
# SUITE.RC DEFAULTS ___________________________________________________
# This file shows ALL cylc configuration items in context, with defaults.
# Below, 'None' is the Python undefined value; '(none)' is more general.
# ____________________________________________________________________
# DEVELOPER NOTE: this file must be be kept up to date with the suite.rc
# specification file $CYLC_DIR/conf/suiterc.spec, AND with higher level
# handling of configuration items (e.g. remote logging directories
# default to the local logging directories; this is effected within cylc
# not via the specification file).

title = "No title provided" # DEFAULT
description = "No description provided" # DEFAULT

[cylc]
    UTC mode = False  # DEFAULT
```

```
    simulation mode only = False  # DEFAULT
    use secure passphrase = False  # DEFAULT
    [[logging]]
        directory = '$HOME/cylc-run/$CYLC_SUITE_REG_NAME/log/suite' # DEFAULT
        roll over at start-up = True  # DEFAULT
    [[state dumps]]
        directory = '$HOME/cylc-run/$CYLC_SUITE_REG_NAME/state' # DEFAULT
        number of backups = 10  # DEFAULT
    [[lockserver]]
        enable = False # DEFAULT
        allow simultaneous instances = False # DEFAULT
    [[environment]]
        # (none)
    [[simulation mode]]
        clock offset = 25  # DEFAULT
        clock rate = 10  # DEFAULT
        command scripting = "echo SIMULATION MODE $CYLC_TASK_ID; sleep 10; echo BYE"  # DEFAULT
        [[[job submission]]]
            method = background  # DEFAULT
        [[[event hooks]]]
            enable = False # DEFAULT
[scheduling]
    initial cycle time = None
    final cycle time = None
    runahead limit = 24 # DEFAULT
    [[special tasks]]
        clock-triggered = (none)
        start-up = (none)
        cold-start = (none)
        sequential = (none)
        one-off = (none)
        explicit restart outputs = (none)
        exclude at start-up = (none)
        include at start-up = (none)
    [[dependencies]]
        graph = None
        [[[__many__]]]
            graph = None
            daemon = (none)
[runtime]
    [[root]]
        inherit = None # (other namespaces inherit first from root)
        description = "No description provided"  # DEFAULT
        pre-command scripting = None
        command scripting = "echo DUMMY $CYLC_TASK_ID; sleep 10; echo BYE"  # DEFAULT
        post-command scripting = None
        manual completion = False  # DEFAULT
        [[[job submission]]]
            method = background  # DEFAULT
            command template = None
            shell = /bin/bash  # DEFAULT
            log directory = '$HOME/cylc-run/$CYLC_SUITE_REG_NAME/log/job'  # DEFAULT
            share directory = '$CYLC_SUITE_DEF_PATH/share' # DEFAULT
            work directory = '$CYLC_SUITE_DEF_PATH/work/$CYLC_TASK_ID' # DEFAULT
        [[[remote]]]
            host = None
            owner = None
            cylc directory = None
            suite definition directory = (local directory with '$HOME' replaced) # DEFAULT
            log directory  = (local directory with '$HOME' replaced) # DEFAULT
            share directory = (local directory with '$HOME' replaced) # DEFAULT
            work directory = (local directory with '$HOME' replaced) # DEFAULT
            remote shell template = 'ssh -oBatchMode=yes %s' # DEFAULT
        [[[event hooks]]]
            script = None
            events = (none)
            submission timeout = None
            execution timeout = None
            reset time = False  # DEFAULT
        [[[environment]]]
```

```
            # (SUITE AND TASK IDENTITY VARIABLES ARE PROVIDED)
        [[[directives]]]
            # (none)
        [[[outputs]]]
            # (none)

[visualization]
    initial cycle time = 2999010100  # DEFAULT (via config.py, not suiterc.spec)
    final cycle time = 2999010123 # DEFAULT (via config.py, not suiterc.spec)
    collapsed families = (none)
    use node color for edges = True  # DEFAULT
    default node attributes = 'style=unfilled', 'color=black', 'shape=box'  # DEFAULT
    default edge attributes = 'color=black'  # DEFAULT
    [[node groups]]
        # (none)
    [[node attributes]]
        # (none)
    [[run time graph]]
        enable = False  # DEFAULT
        cutoff = 24  # DEFAULT
        directory = '$CYLC_SUITE_DEF_PATH/graphing'  # DEFAULT
```

B Command Reference

```
cylc release version: THIS IS NOT A VERSIONED RELEASE

USAGE:
  cylc -v,--version                                # cylc version
  cylc help,--help,-h,?                            # this help page
  cylc help CATEGORY                               # help by category
  cylc CATEGORY help                               # (ditto)
  cylc help [CATEGORY] COMMAND                     # command help
  cylc [CATEGORY] COMMAND help,--help              # (ditto)
Command syntax:
  cylc [CATEGORY] COMMAND [options] SUITE [arguments]
  cylc [CATEGORY] COMMAND [options] SUITE TASK [arguments]

Commands and categories can be abbreviated. Use of categories is
optional, but they can disambiguate highly abbreviated commands:
  $ cylc control trigger SUITE TASK  # trigger TASK in SUITE
  $ cylc trigger SUITE TASK          # ditto
  $ cylc con trig SUITE TASK         # ditto
  $ cylc c t SUITE TASK              # ditto

SUITE - cylc commands target suites via names registered in a suite
database so that you don't need to continually re-type the actual
location of the suite definition directory on disk. Suite names are
hierarchical like directory paths but delimited by '.' (foo.bar.baz),
allowing suites to be grouped and organised into tree-like structures.

TASK - is a unique identitifier of a task in a suite: NAME%TAG, where
for cycling tasks TAG is a cycle time (YYYYMMDDHH) and for asynchronous
tasks it is an integer (preceded by 'a:' on the command line: foo%a:1).

Here's how to drill down to usage help for a particular command:
  $ cylc help ............... list all available categories (this page)
  $ cylc help prep .......... list commands in category 'preparation'
  $ cylc help prep edit ..... command usage help for 'cylc [prep] edit'

Command CATEGORIES:
  all ........... the complete cylc command set
  db|database ... private and central suite registration.
  preparation ... suite editing, graphing, validation etc.
  information ... retrieve and print information about a suite.
  discovery ..... what suites are running at the moment?
  control ....... running, monitoring, and controlling suites.
  utility ....... cycle time arithmetic, filename templating, etc.
  task .......... running single tasks, and task messaging commands.
  admin ......... commands for use by the cylc administrator.
  license|GPL ... Software licensing information (GPL v3.0).
```

B.1 Command Categories

B.1.1 admin

```
CATEGORY: admin - commands for use by the cylc administrator.

HELP: cylc [admin] COMMAND help,--help
  You can abbreviate admin and COMMAND.
  The category admin may be omitted.

COMMANDS:
  check-examples ... (ADMIN) Check all example suites validate
  create-cdb ....... (ADMIN) Create the central suite database
  test-db .......... (ADMIN) Automated suite database test
  test-suite ....... (ADMIN) Automated cylc scheduler test
```

B.1.2 all

```
CATEGORY: all - the complete cylc command set
```

```
HELP: cylc [all] COMMAND help,--help
  You can abbreviate all and COMMAND.
  The category all may be omitted.

COMMANDS:
  alias ...................... Register an alias for another registered suite
  block ...................... Do not comply with subsequent intervention commands
  check-examples ............. (ADMIN) Check all example suites validate
  checkvars .................. Check required environment variables en masse
  conditions ................. Print the GNU General Public License v3.0
  copy|cp .................... Copy a suite or group of suites.
  create-cdb ................. (ADMIN) Create the central suite database
  cycletime .................. Cycle time arithmetic
  depend ..................... Add prerequisites to tasks on the fly
  diff|compare ............... Compare two suite definitions and print differences
  dump ....................... Print the state of each task in a running suite
  edit ....................... Edit a suite.rc file in $EDITOR, optionally inlined
  email-alert ................ Send alerts by email when tasks fail (for example)
  export ..................... Export private registrations to the central database
  failed|task-failed ......... Release task lock and report failure
  gcylc ...................... The cylc Graphical User Interface
  get-config ................. Retrieve any configuration data from a suite.
  get-directory .............. Print a suite definition directory path
  graph ...................... A dependency graph viewer that updates as you edit
  hold ....................... Put a hold on a suite or a single task
  housekeeping ............... Parallel archiving and cleanup on cycle time offsets
  import ..................... Import central registrations to your private database
  inline ..................... View suite.rc files in $EDITOR, include-files inlined
  insert ..................... Insert a task or group into a running suite
  jobscript .................. Generate a task job script and print it to stdout
  list|ls .................... Print a suite's task list or full runtime hierarchy
  lockclient|lc .............. Manual suite and task lock management
  lockserver ................. The cylc lockserver daemon
  log ........................ Print or view suite logs, with filtering
  maxrunahead ................ Change the runahead limit in a running suite.
  message|task-message ....... Report progress and completion of outputs
  monitor .................... An in-terminal suite monitor (see also gcylc)
  nudge ...................... Cause the cylc task processing loop to be invoked
  ping ....................... Check that a suite is running
  print ...................... Print private or central suite registrations
  purge ...................... Remove a full dependency tree from a running suite
  refresh .................... Report invalid registrations and update suite titles
  register ................... Add a suite to your private registration database
  release|unhold ............. Release a  hold on a suite or a single task
  remove|kill ................ Remove a task from a running suite
  reregister|rename .......... Change a suite's registered name.
  reset ...................... Force a task to waiting, ready, or succeeded state
  restart .................... Restart a suite from a previous state
  run|start .................. Start a suite running at a specified cycle time
  scan ....................... Scan a host for running suites and lockservers
  scp-transfer ............... Scp-based file transfer for cylc suites
  search|grep ................ An intelligent search tool for cylc suites
  show ....................... Print task-specific information (prerequisites...)
  started|task-started ....... Acquire a task lock and report started
  stop|shutdown .............. Stop a suite running by various means
  submit|single .............. Run a single task just as its parent suite would
  succeeded|task-succeeded ... Release task lock and report succeeded
  template ................... Powerful cycle time offset filename templating
  test-db .................... (ADMIN) Automated suite database test
  test-suite ................. (ADMIN) Automated cylc scheduler test
  trigger .................... Cause a task to trigger immediately
  unblock .................... Comply with subsequent intervention commands
  unregister ................. Remove suites from the private or central database
  validate ................... Parse and validate a suite config (suite.rc) file
  verbosity .................. Change a suite's logging verbosity level
  warranty ................... Print the GPLv3 disclaimer of warranty
```

B.1.3 control

```
CATEGORY: control - running, monitoring, and controlling suites.

HELP: cylc [control] COMMAND help,--help
  You can abbreviate control and COMMAND.
  The category control may be omitted.

COMMANDS:
  block ............ Do not comply with subsequent intervention commands
  depend ........... Add prerequisites to tasks on the fly
  gcylc ............ The cylc Graphical User Interface
  hold ............. Put a hold on a suite or a single task
  insert ........... Insert a task or group into a running suite
  lockclient|lc .... Manual suite and task lock management
  lockserver ....... The cylc lockserver daemon
  maxrunahead ...... Change the runahead limit in a running suite.
  nudge ............ Cause the cylc task processing loop to be invoked
  purge ............ Remove a full dependency tree from a running suite
  release|unhold ... Release a  hold on a suite or a single task
  remove|kill ...... Remove a task from a running suite
  reset ............ Force a task to waiting, ready, or succeeded state
  restart .......... Restart a suite from a previous state
  run|start ........ Start a suite running at a specified cycle time
  stop|shutdown .... Stop a suite running by various means
  trigger .......... Cause a task to trigger immediately
  unblock .......... Comply with subsequent intervention commands
  verbosity ........ Change a suite's logging verbosity level
```

B.1.4 database

```
CATEGORY: db|database - private and central suite registration.

HELP: cylc [db|database] COMMAND help,--help
  You can abbreviate db|database and COMMAND.
  The category db|database may be omitted.

COMMANDS:
  alias ............... Register an alias for another registered suite
  copy|cp ............. Copy a suite or group of suites.
  export .............. Export private registrations to the central database
  gcylc ............... The cylc Graphical User Interface
  get-directory ....... Print a suite definition directory path
  import .............. Import central registrations to your private database
  print ............... Print private or central suite registrations
  refresh ............. Report invalid registrations and update suite titles
  register ............ Add a suite to your private registration database
  reregister|rename ... Change a suite's registered name.
  unregister .......... Remove suites from the private or central database
```

B.1.5 discovery

```
CATEGORY: discovery - what suites are running at the moment?

HELP: cylc [discovery] COMMAND help,--help
  You can abbreviate discovery and COMMAND.
  The category discovery may be omitted.

COMMANDS:
  ping ... Check that a suite is running
  scan ... Scan a host for running suites and lockservers
```

B.1.6 information

```
CATEGORY: information - retrieve and print information about a suite.

HELP: cylc [information] COMMAND help,--help
  You can abbreviate information and COMMAND.
  The category information may be omitted.

COMMANDS:
```

```
  dump ......... Print the state of each task in a running suite
  gcylc ........ The cylc Graphical User Interface
  get-config ... Retrieve any configuration data from a suite.
  list|ls ...... Print a suite's task list or full runtime hierarchy
  log .......... Print or view suite logs, with filtering
  monitor ...... An in-terminal suite monitor (see also gcylc)
  nudge ........ Cause the cylc task processing loop to be invoked
  show ......... Print task-specific information (prerequisites...)
```

B.1.7 license

```
CATEGORY: license|GPL - Software licensing information (GPL v3.0).

HELP: cylc [license|GPL] COMMAND help,--help
  You can abbreviate license|GPL and COMMAND.
  The category license|GPL may be omitted.

COMMANDS:
  conditions ... Print the GNU General Public License v3.0
  warranty ..... Print the GPLv3 disclaimer of warranty
```

B.1.8 preparation

```
CATEGORY: preparation - suite editing, graphing, validation etc.

HELP: cylc [preparation] COMMAND help,--help
  You can abbreviate preparation and COMMAND.
  The category preparation may be omitted.

COMMANDS:
  diff|compare ... Compare two suite definitions and print differences
  edit ........... Edit a suite.rc file in $EDITOR, optionally inlined
  gcylc .......... The cylc Graphical User Interface
  graph .......... A dependency graph viewer that updates as you edit
  inline ......... View suite.rc files in $EDITOR, include-files inlined
  jobscript ...... Generate a task job script and print it to stdout
  list|ls ........ Print a suite's task list or full runtime hierarchy
  search|grep .... An intelligent search tool for cylc suites
  validate ....... Parse and validate a suite config (suite.rc) file
```

B.1.9 task

```
CATEGORY: task - running single tasks, and task messaging commands.

HELP: cylc [task] COMMAND help,--help
  You can abbreviate task and COMMAND.
  The category task may be omitted.

COMMANDS:
  failed|task-failed ......... Release task lock and report failure
  message|task-message ....... Report progress and completion of outputs
  started|task-started ....... Acquire a task lock and report started
  submit|single ............. Run a single task just as its parent suite would
  succeeded|task-succeeded ... Release task lock and report succeeded
```

B.1.10 utility

```
CATEGORY: utility - cycle time arithmetic, filename templating, etc.

HELP: cylc [utility] COMMAND help,--help
  You can abbreviate utility and COMMAND.
  The category utility may be omitted.

COMMANDS:
  checkvars ...... Check required environment variables en masse
  cycletime ...... Cycle time arithmetic
  email-alert .... Send alerts by email when tasks fail (for example)
  housekeeping ... Parallel archiving and cleanup on cycle time offsets
  scp-transfer ... Scp-based file transfer for cylc suites
  template ....... Powerful cycle time offset filename templating
```

B.2 Commands

B.2.1 alias

```
Usage: cylc [db] alias [options] SUITE ALIAS

Register an alias for suite. Using the alias on the command line is
entirely equivalent to using target suite name (and if you run a suite
via an alias, run time identity will correspond to the target suite).

  $ cylc alias global.ensemble.parallel.test3 bob
  $ cylc edit bob
  $ cylc run  bob     # work with global.ensemble.parallel.test3
  $ cylc stop bob

This differs from registering the target suite definition under another
name with 'cylc register' - in that case run time suite identity will
correspond to the new registered name.

Arguments:
     SUITE  -  Target suite.
     ALIAS  -  An alternative name for the same suite.

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Print extra information.
```

B.2.2 block

```
Usage: cylc [control] block [options] SUITE

A blocked suite refuses to comply with intervention commands until
deliberately unblocked. This is a crude security measure to guard
against accidental intervention in your own suites. It may be a
useful aid when running multiple suites at once.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.3 check-examples

```
USAGE: cylc [admin] check-examples

Check that all cylc example suites validate successfully.
Do this immediatedly prior to rolling a new cylc release.
```

B.2.4 checkvars

```
Usage: cylc checkvars [options] VARNAMES

Check that each member of a list of environment variables is defined,
and then optionally check their values according to the chosen
commandline option. Note that THE VARIABLES MUST BE EXPORTED AS THIS
SCRIPT NECESSARILY EXECUTES IN A SUBSHELL.

All of the input variables are checked in turn and the results printed.
If any problems are found then, depending on use of '-w,--warn-only',
this script either aborts with exit status 1 (error) or emits a stern
warning and exits with status 0 (success).
```

```
Arguments:
   VARNAMES     Space-separated list of environment variable names.

Options:
  -h, --help            show this help message and exit
  -d, --dirs-exist      Check that the variables refer to directories that
                        exist.
  -c, --create-dirs     Attempt to create the directories referred to by the
                        variables, if they do not already exist.
  -p, --create-parent-dirs
                        Attempt to create the parent directories of files
                        referred to by the variables, if they do not already
                        exist.
  -f, --files-exist     Check that the variables refer to files that exist.
  -i, --int             Check that the variables refer to integer values.
  -s, --silent          Do not print the result of each check.
  -w, --warn-only       Print a warning instead of aborting with error status.
```

B.2.5 conditions

```
USAGE: cylc [license] warranty [--help]
Cylc is release under the GNU General Public License v3.0
This command prints the GPL v3.0 license in full.

Options:
  --help   Print this usage message.
```

B.2.6 copy

```
Usage: cylc [db] copy|cp [OPTIONS] SOURCE TARGET TOPDIR

Make copies of suites, or groups of suites, in your private database
(to get copies of suites in the central database use 'cylc import').

Cylc suites have hierarchical "registration names" that are analogous to
file paths in a directory tree (but which are not necessarily related to
the actual directory paths in which the suite definitions are stored!).
Consider the following three suites:

% cylc db print '^foo'     # printed in flat form
foo.bag     | "Test Suite Zero" | /tmp/oliverh/zero
foo.bar.waz | "Test Suite Two"  | /tmp/oliverh/two
foo.bar.baz | "Test Suite One"  | /tmp/oliverh/one

% cylc db print -t '^foo'  # printed in tree from
foo
 |-bag    "Test Suite Zero" | /tmp/oliverh/zero
 `-bar
   |-baz  "Test Suite One"  | /tmp/oliverh/one
   `-waz  "Test Suite Two"  | /tmp/oliverh/two

These suites are stored in a flat directory structure under $TMPDIR, but
they are organised in the suite database as a group 'foo' that contains
the suite 'foo.bag' and a group 'foo.bar', which in turn contains the
suites 'foo.bar.baz' and 'foo.bar.waz'.

When you copy suites with this command, the target registration names
are determined by TARGET and the name structure underneath SOURCE, and
the suite definition directories are copied into a directory tree under
TOPDIR whose structure reflects the target registration names. If this
is not what you want, you can copy suite definition directories manually
and then register the copied directories manually with 'cylc register'.

EXAMPLES (using the three suites above):

% cylc db copy foo.bar.baz red $TMPDIR       # suite to suite
  Copying suite definition for red
% cylc db print '^red'
  red | "Test Suite One" | /tmp/oliverh/red
```

```
% cylc copy foo.bar.baz blue.green $TMPDIR   # suite to group
  Copying suite definition for blue.green
% cylc db pr '^blue'
  blue.green | "Test Suite One" | /tmp/oliverh/blue/green

% cylc copy foo.bar orange $TMPDIR           # group to group
  Copying suite definition for orange.waz
  Copying suite definition for orange.baz
% cylc db pr '^orange'
  orange.waz | "Test Suite Two" | /tmp/oliverh/orange/waz
  orange.baz | "Test Suite One" | /tmp/oliverh/orange/baz

Arguments:
    SOURCE - Source suite or group
    TARGET - Target suite or group
    TOPDIR - Destination for copied suite definition directories

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Print extra information.
```

B.2.7 create-cdb

```
USAGE: cylc [admin] create-cdb

This command is used by the cylc administrator to create the central
suite database immediately after installing cylc, or to upgrade the
centrally held cylc example suites after upgrading to a new version.

The example suites are registered with the cylc version number in
the registration path so that multiple versions can be retained.

Seconds since epoch are also appended to the cylc version number to
allow multiple uploads at the same cylc version (in case the examples
need to be modified). You can then rename the uploaded examples group
and unregister any older uploads if necessary.
```

B.2.8 cycletime

```
Usage: cylc [util] cycletime [options] [CYCLE]

Perform arithmetic operations on cycle times and print the results to
sdtout. Examples:

   # offset from explicit cycle time:
   $ cylc cycletime -s 6 2010082318
  2010082312

   # offset from $CYLC_TASK_CYCLE_TIME:
   $ export CYLC_TASK_CYCLE_TIME=2010082318
   $ cylc cycletime -s 6
  2010082312

Arguments:
   CYCLE        (YYYYMMDDHH) defaults to $CYLC_TASK_CYCLE_TIME.

Options:
  -h, --help            show this help message and exit
  -s HOURS, --subtract=HOURS
                        Subtract HOURS from CYCLE
  -a HOURS, --add=HOURS
                        Add HOURS to CYCLE
  -o HOURS, --offset=HOURS
                        Apply an offset of +/-HOURS to CYCLE
  --year                Print only YYYY of result
  --month               Print only MM of result
  --day                 Print only DD of result
  --hour                Print only HH of result
```

B.2.9 depend

```
Usage: cylc [control] depend [options] SUITE TASK DEP

Add new dependencies on the fly to tasks in a running suite. If DEP
is a task ID the target TASK will depend on that task finishing,
otherwise DEP can be an explicit quoted message such as
  "Data files uploaded for 2011080806"
(presumably there will be another task in the suite, or you will insert
one, that reports that message as an output).

Prerequisites added on the fly are not propagated to the successors
of TASK, and they will not persist in TASK across a suite restart.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                 Target suite.
   TASK                  Target task.
   DEP                   New dependency for the target task.

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.10 diff

```
Usage: cylc [prep] diff|compare [options] SUITE1 SUITE2

Compare two suite definitions and display any differences.

Differencing is done after parsing the suite.rc files so it takes
account of default values that are not explicitly defined, it disregards
the order of configuration items, and it sees any include-file content
after inlining has occurred.

Seemingly identical suites can differ slightly if they use default
configuration items, such as the default job submission log directory,
that are suite-specific (i.e. they include $CYLC_SUITE_REG_NAME etc.).

Files in the suite bin directory and other sub-directories of the
suite definition directory are not currently differenced.

Arguments:
   SUITE1, SUITE2  - suites to compare.

Options:
  -h, --help     show this help message and exit
  -n, --nested   print suite.rc section headings in nested form.
  -c, --central  target the central database.
```

B.2.11 dump

```
Usage: cylc [info] dump [options] SUITE

Print current suite state information (e.g. the state of each task). For
small suites 'watch cylc [info] dump SUITE' is an effective non-GUI real
time monitor (but see also 'cylc monitor').

For more information about a specific task, such as the current state of
its prerequisites and outputs, see 'cylc [info] show'.

Examples:
 Display the state of all running tasks, sorted by cycle time:
 % cylc [info] dump -ts SUITE | grep running

 Display the state of all tasks in a particular cycle:
 % cylc [info] dump -t SUITE | grep 2010082406
```



```
Arguments:
   SUITE                Target suite.

Options:
  -h, --help            show this help message and exit
  -o USER, --owner=USER
                        Owner of the target suite (defaults to $USER).
  --host=HOST           Cylc suite host (defaults to local host).
  -f, --force           (No effect; for consistency with interactive commands)
  --debug               Turn on exception tracebacks.
  -g, --global          Global information only.
  -t, --tasks           Task states only.
  -s, --sort            Task states only; sort by cycle time instead of name.
```

B.2.12 edit

```
Usage: 1/ cylc [prep] edit SUITE
       2/ cylc [prep] edit -i,--inline SUITE
       3/ cylc [prep] edit --cleanup SUITE

1/ Change to the suite definition directory and load the suite.rc file
in your $EDITOR. This is a convenience so that you can edit the suite
without having to remember the suite definition location.

2/ Edit suite.rc files WITH INCLUDE-FILES INLINED between special
markers.  The real suite.rc file is temporarily replaced so that
THE INLINED SUITE.RC FILE IS THE "LIVE" FILE DURING EDITING. The
inlined suite.rc file gets split up again when you exit the
editor. Include-files can be nested or multiply-included; in the
latter case only the first inclusion is inlined (this serves to
prevent conflicting changes being made to the same file).

3/ Remove backup files left by previous INLINED edit sessions.

INLINED EDITING SAFETY: The suite.rc file and any include-files used
by it are automatically backed up prior to an inlined editing session.
If the editor dies mid-session just invoke 'cylc edit -i' again to
recover from the last saved inlined file. On exiting the editor, if any
of the original include-files are found to have changed during editing
(due to external intervention) you will be warned and the affected files
will be written to new backups instead of overwriting the originals.
FOR IMPORTANT SUITES YOU SHOULD USE REVISION CONTROL IN ANY CASE,
regardless of inlined editing.

The edit process is spawned in the foreground as follows:
  $(G)EDITOR suite.rc
$GEDITOR or $EDITOR, and $TMDPIR, must be in your environment.

Examples:
   export EDITOR=vim
   export GEDITOR='gvim -f'      # -f: do not detach from parent shell!!
   export EDITOR='xterm -e vim'  # for gcylc, if gvim is not available
   export GEDITOR=emacs
   export EDITOR='emacs -nw'
You can set both $GEDITOR and $EDITOR to a GUI editor if you like, but
$GEDITOR at least *must* be a GUI editor, or an in-terminal invocation
of a non-GUI editor, if you want to spawn editing sessions via gcylc.

Arguments:
  SUITE         Target suite.

Options:
  -h, --help    show this help message and exit
  -i, --inline  Edit with include-files inlined as described above.
  --central     Target the central suite database.
  --cleanup     Remove backup files left by previous inlined edit sessions.
  -g, --gui     Use GUI editor $GEDITOR instead of $EDITOR. This option is
                automatically used when an editing session is spawned by
                gcylc.
```

B.2.13 email-alert

```
USAGE: cylc email-alert EVENT SUITE TASKID MESSAGE

This is a simple cylc event hook script that sends an email.
The command line arguments are supplied automatically by cylc.

For example, to get an email alert whenever any task fails:

# SUITE.RC
[cylc]
   [[environment]]
      MAIL_ADDRESS = foo@bar.baz.waz
[runtime]
   [[root]]
      [[[event hooks]]]
         events = failed
         script = cylc email-alert

See the Suite.rc Reference (Cylc User Guide) for more information
on event hooks and event handler scripts.
```

B.2.14 export

```
Usage: cylc [db] export [OPTIONS] SOURCE [TARGET]

Export suites or groups of suites to the central suite database.

Central registrations always start with the suite owner's username. If a
target name is not given ...

Arguments:
    SOURCE  - Suite to export to the central database
    TARGET  - Target central suite name, if different from SOURCE.

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Print extra information.
```

B.2.15 failed

```
Usage: cylc [task] failed [options] [REASON]

This is part of the cylc external task interface.

Release my lock to the lockserver, and report that I have failed.

Arguments:
    REASON        - message explaining why the task failed.

Options:
  -h, --help  show this help message and exit
```

B.2.16 gcylc

```
Note that you can invoke gcylc directly.
Usage: cylc gcylc [SUITE]
       gcylc [SUITE]

This is the cylc graphical user interface. It is functionally equivalent
to the command line interface ('cylc help') in most respects.

    1/ gcylc &
This invokes the gcylc main window, which displays suites in your
private registration database, and in the central database available to
all users. By right-clicking on suites or registration groups you can
gain access to all cylc functionality, from editing and graphing through
to suite control and monitoring.
```

```
    2/ gcylc [-g,--graph] SUITE &
This directly invokes a suite control and monitoring application for a
particular suite. Alternatively you can get this by right-clicking on
the suite in the main gcylc suite database viewer (1/ above). Without
the -g option you'll get a filtered text treeview GUI; with it, a
dependency graph based GUI.

NOTE: daemonize important suites with the POSIX nohup command:
    $ nohup gcylc [options] SUITE &

Arguments:
    SUITE      Target suite.

Options:
  -h, --help   show this help message and exit
  -g, --graph  With SUITE - invoked the new dependency graph based suite
               control and monitoring interface.
```

B.2.17 get-config

```
Usage: cylc [info] get-config SUITE [HIERARCHY]

Print configuration items parsed from a suite definition (including any
defaults not explicitly set in the suite.rc file). This enables scripts
to control suites or process their output without assuming directory
locations and so on.

Config items containing spaces must be quoted. If the requested config
item is not a single value the corresponding internal data structure (a
nested Python dict) will be printed verbatim.

#_________________________________________________________________EXAMPLE
# excerpt from a suite definition registered as foo.bar.baz:
title = local area implementation of modelX
[cylc]
    [[lockserver]]
        enable = True
[runtime]
    [[modelX]]
        command scripting = run-model.sh
        [[[environment]]]
            OUTPUT_DIR=/oper/live/modelX/output
#_______________________________________________________________________

$ cylc get-config foo.bar.baz title
$ local area implementation of modelX

$ cylc get-config foo.bar.baz cylc lockserver enable
$ True

$ cylc get-config foo.bar.baz runtime modelX environment OUTPUT_DIR
$ /oper/live/modelX/output

Arguments:
    SUITE - target suite.
    HIERARCHY - list of suite.rc item hierarchy elements.

Options:
  -h, --help         show this help message and exit
  -c, --central      Target the central suite database.
  -d, --directories  Print all configured suite directory paths.
```

B.2.18 get-directory

```
Usage: cylc [db] get-directory SUITE

Retrieve and print a suite definition directory path.
How to move to a suite definition directory, for the lazy:
  $ cd $(cylc get SUITE)
```

```
Arguments:
    SUITE - target suite.

Options:
  -h, --help     show this help message and exit
  -c, --central  Target the central suite database.
```

B.2.19 graph

```
Usage: 1/ cylc [prep] graph [options] SUITE [START [STOP]]
     Plot the suite.rc dependency graph for SUITE.
       2/ cylc [prep] graph [options] -f,--file FILE
     Plot the specified dot-language graph file.

Plot cylc dependency graphs in a pannable, zoomable viewer.

The viewer updates automatically when the suite.rc file is saved during
editing. By default the full cold start graph is plotted; you can omit
cold start tasks with the '-w,--warmstart' option.  Specify the optional
initial and final cycle time arguments to override the suite.rc defaults.
If you just override the intitial cycle, only that cycle will be plotted.

GRAPH VIEWER CONTROLS:
    * Left-click to center the graph on a node.
    * Left-drag to pan the view.
    * Zoom buttons, mouse-wheel, or ctrl-left-drag to zoom in and out.
    * Shift-left-drag to zoom in on a box.
    * Also: "Best Fit" and "Normal Size".
  Family (namespace) grouping controls:
    Toolbar:
    * "group" - group all families up to root.
    * "ungroup" - recursively ungroup all families.
    Right-click menu:
    * "group" - close this node's parent family.
    * "ungroup" - open this family node.
    * "recursive ungroup" - ungroup all families below this node.

Arguments:
  SUITE    - Target suite.
  START    - Initial cycle time to plot (default=2999010100)
  STOP     - Final cycle time to plot (default=2999010123)


Options:
  -h, --help            show this help message and exit
  -w, --warmstart       Plot the mid-stream warm start graph (technically this
                        is termed a 'raw start' in cylc).
  -f FILE, --file=FILE  View a specific dot-language graphfile.
  -c, --central         Target the central database.
  -o FILE, --output=FILE
                        Write an image file with format determined by file
                        extension. The image will be rewritten automatically,
                        for the configured (suite.rc) graph as you edit the
                        suite. Available formats may include png, svg, jpg,
                        gif, ps, ..., depending on your graphviz build; to see
                        what's available specify a non-existent format and
                        read the resulting error message.
```

B.2.20 hold

```
Usage: cylc [control] hold [options] SUITE [TASK]

Put a hold on a suite or a single waiting task. Putting a suite on hold
stops it from submitting any tasks that are ready to run, until it is
released.  Putting a waiting task on hold prevents it from running and
spawning successors, until it is released.

See also 'cylc [control] release'.
```

```
You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.
   TASK                 Task to hold (NAME%YYYYMMDDHH)

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.21 housekeeping

```
Usage: 1/ cylc [util] housekeeping [options] SOURCE MATCH OPER OFFSET [TARGET]
Usage: 2/ cylc [util] housekeeping [options] FILE

Parallel archiving and cleanup of files or directories with names
that contain a cycle time. Matched items are grouped into batches in
which members are processed in parallel, by spawned sub-processes.
Once all batch members have completed, the next batch is processed.

OPERATE ('delete', 'move', or 'copy') on items (files or directories)
matching a Python-style regular expression MATCH in directory SOURCE
whose names contain a cycle time (as YYYYMMDDHH, or YYYYMMDD and HH
separately) more than OFFSET (integer hours) earlier than a base cycle
time (which can be $CYLC_TASK_CYCLE_TIME if called by a cylc task, or
otherwise specified on the command line).

FILE is a housekeeping config file containing one or more of lines of:

   VARNAME=VALUE
   # comment
   SOURCE    MATCH    OPERATION   OFFSET   [TARGET]

(example: $CYLC_DIR/conf/housekeeping.eg)

MATCH must be a Python-style regular expression (NOT A SHELL GLOB
EXPRESSION!) to match the names of items to be operated on AND to
extract the cycle time from the names via one or two parenthesized
sub-expressions - '(\d{10})' for YYYYMMDDHH, '(\d{8})' and '(\d{2})'
for YYYYMMDD and HH in either order. Partial matching can be used
(partial: 'foo-(\d{10})'; full: '^foo-(\d{10})$'). Any additional
parenthesized sub-expressions, e.g. for either-or matching, MUST
be of the (?:...) type to avoid creating a new match group.

SOURCE and TARGET must be on the local filesystem and may contain
environment varables such as $HOME or ${FOO} (e.g. as defined in the
suite.rc file for suite housekeeping tasks). Variables defined in
the housekeeping file itself can also be used, as above.

TARGET may contain the strings YYYYMMDDHH, YYYY, MM, DD, HH; these
will be replaced with the extracted cycle time for each matched item,
e.g. $ARCHIVE/oper/YYYYMM/DD.

If TARGET is specified for the 'delete' operation, matched items in
SOURCE will not be deleted unless an identical item is found in
TARGET. This can be used to check that important files have been
successfully archived before deleting the originals.

The 'move' and 'copy' operations are aborted if the TARGET/item already
exists, but a warning is emitted if the source and target items are not
identical.

To implement a simple ROLLING ARCHIVE of cycle-time labelled files or
directories: just use 'delete' with OFFSET set to the archive length.

SAFE ARCHIVING: The 'move' operation is safe - it uses Python's
shutils.move() which renames files on the local disk partition and
otherwise copies before deleting the original. But for extra safety
```

```
consider two-step archiving and cleanup:
1/ copy files to archive, then
2/ delete the originals only if identicals are found in the archive.

Options:
  -h, --help            show this help message and exit
  --cycletime=YYYYMMDDHH
                        Cycle time, defaults to $CYLC_TASK_CYCLE_TIME
  --mode=MODE           Octal umask for creating new destination
                        directoriesE.g. 0775 for drwxrwxr-x
  -o LIST, --only=LIST  Only action config file lines matching any member of a
                        comma-separated list of regular expressions.
  -e LIST, --except=LIST
                        Only action config file lines NOT matching any member
                        of a comma-separated list of regular expressions.
  -v, --verbose         print the result of every action
  -d, --debug           print item matching output.
  -c, --cheapdiff       Assume source and target identical if the same size
  -b INT, --batchsize=INT
                        Batch size for parallel processing of matched files.
                        Members of each batch (matched items) are processed in
                        parallel; when a batch completes, the next batch
                        starts. Defaults to a batch size of 1, i.e. sequential
                        processing.
```

B.2.22 import

```
Usage: cylc [db] import [OPTIONS] SOURCE [TARGET] TOPDIR

Import suites or groups of suites from the central registration database
to your private database. Import semantics are the same as for the copy
command. The directory path of imported suite definitions under TOPDIR
will reflect the corresponding suite names, with '.' replaced by '/'.
See 'cylc db copy --help' for examples.

Arguments:
    SOURCE    - Central suite or group (e.g. alice.lam.test).
   [TARGET]   - Private suite or group (omit to use SOURCE: lam.test).
    TOPDIR    - Top directory for imported suite definitions.

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Print extra information.
```

B.2.23 inline

```
Usage: cylc [prep] inline [options] SUITE

View a temporary copy of a suite config (suite.rc) file in your $EDITOR
(or GUI $GEDITOR) exactly as the config file parser sees it: with
include-files inlined and continuation lines joined. This can be used to
get a quick global view of a suite that uses include-files, or to trace
parsing errors by line number (suite validation is sufficiently
informative, however, that you should not need to do this).

If you want to edit the suite, in inlined form or not, or to view it
without inlined include-files, use the 'cylc prep edit' command.

The read-only edit process is spawned in the foreground as follows:
  $(G)EDITOR <temporary-inlined-file-in-$TMPDIR>
$GEDITOR or $EDITOR, and $TMPDIR, must be defined in your environment.
Examples:
   export EDITOR=vim
   export GEDITOR='gvim -f'       # -f: do not detach from parent shell!!
   export EDITOR='xterm -e vim'  # for gcylc, if gvim is not available
   export GEDITOR=emacs
   export EDITOR='emacs -nw'
You can set both $GEDITOR and $EDITOR to a GUI editor if you like, but
$GEDITOR at least *must* be a GUI editor, or an in-terminal invocation
```

```
of a non-GUI editor, if you want to spawn editing sessions via gcylc.

See also: cylc prep edit

Arguments:
  SUITE   - suite registration

Options:
  -h, --help      show this help message and exit
  -c, --central   Target the central database.
  -m, --mark      Mark inclusions in the left margin (line numbers will still
                  correspond to those reported by the parser).
  -l, --label     Label file inclusion boundaries with the file name (line
                  numbers will not correspond to those reported by the parser).
  -n, --nojoin    Do not join continuation lines (line numbers will not
                  correspond to those reported by the parser).
  -s, --single    Inline and label just the first instance of any multiply-
                  included file (line numbers will not correspond to those
                  reported by the parser).
  -g, --gui       Use GUI editor $GEDITOR instead of $EDITOR. This option is
                  automatically used when an editing session is spawned by
                  gcylc.
```

B.2.24 insert

```
Usage: cylc [control] insert [options] SUITE TASK[%STOP]

Insert a task into a running suite. Inserted tasks will spawn successors
as normal unless they are 'one-off' tasks.
See also 'cylc [task] submit', for running single tasks without the scheduler.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                 Target suite.
   TASK                  Task to insert (NAME%TAG).
   STOP                  Optional stop TAG (e.g. a final cycle time).

Options:
  -h, --help    show this help message and exit
  --debug       Turn on exception tracebacks.
  -f, --force   Do not ask for confirmation before acting.
```

B.2.25 jobscript

```
USAGE: cylc [prep] jobscript [options] SUITE TASK

Generate a task job script and print it to stdout.

Here's how to capture the script in the vim editor:
  % cylc jobscript SUITE TASK | vim -
Emacs unfortunately cannot read from stdin:
  % cylc jobscript SUITE TASK > tmp.sh; emacs tmp.sh

This command wraps 'cylc [control] submit --dry-run'.

Options:
  -h,--help   - print this usage message.

Arguments:
  SUITE       - Target suite.
  TASK        - Task NAME or NAME%YYYYMMDDHH (if you omit
                the cycle time 2999010100 will be used).
```

B.2.26 list

```
Usage: cylc [info|prep] list|ls [OPTIONS] SUITE [FILTER]
```

```
Parse a suite and print its task list or inheritance hierarchy.

Arguments:
   SUITE    - Target suite
   FILTER   - regular expression filter on task names.

Options:
  -h, --help     show this help message and exit
  -c, --central  Target the central database.
  -t, --tree     Print the full runtime inheritance hierarchy.
  -p, --pretty   Use unicode box drawing characters when printing the suite
                 runtime inheritance hierarchy.
```

B.2.27 lockclient

```
Usage: cylc [control] lockclient|lc [options]

This is the command line client interface to the cylc lockserver daemon,
for server interrogation and manual lock management.

Use of the lockserver is optional (see suite.rc documentation)

Manual lock acquisition is mainly for testing purposes, but manual
release may be required to remove stale locks if a suite or task dies
without cleaning up after itself.

See also:
    cylc lockserver

Options:
  -h, --help            show this help message and exit
  --acquire-task=SUITE:TASK%CYCLE
                        Acquire a task lock.
  --release-task=SUITE:TASK%CYCLE
                        Release a task lock.
  --acquire-suite=SUITE
                        Acquire an exclusive suite lock.
  --acquire-suite-nonex=SUITE
                        Acquire a non-exclusive suite lock.
  --release-suite=SUITE
                        Release a suite and associated task locks
  -p, --print           Print all locks.
  -l, --list            List all locks (same as -p).
  -c, --clear           Release all locks.
  -f, --filenames       Print lockserver PID, log, and state filenames.
```

B.2.28 lockserver

```
Usage: cylc [control] lockserver [-f CONFIG] ACTION

The cylc lockserver daemon brokers suite and task locks for a single
user. These locks are analogous to traditional lock files, but they work
even for tasks that start and finish executing on different hosts. Suite
locks prevent multiple instances of the same suite from running at the
same time (even if registered under different names) unless the suite
allows that. Task locks do the same for individual tasks (even if
submitted outside of their suite using 'cylc submit').

The command line user interface for interrogating the daemon, and
for manual lock management, is 'cylc lockclient'.

Use of the lockserver is optional (see suite.rc documentation).

The lockserver reads a config file that specifies the location of the
daemon's process ID, state, and log files. The default config file
is '$CYLC_DIR/conf/lockserver.conf'. You can specify an alternative
config file on the command line, but then all subsequent interaction
with the daemon via the lockclient command must also specify the same
file (this is really only for testing purposes). The default process ID,
```

```
state, and log files paths are relative to $HOME so this should be
sufficient for all users.

The state file records currently held locks and, if it exists at
startup, is used to initialize the lockserver (i.e. suite and task locks
are not lost if the lockserver is killed and restarted). All locking
activitiy is recorded in the log file.

Arguments:
  ACTION   -  'start', 'stop', 'status', 'restart', or 'debug'
               In debug mode the server does not daemonize so its
               the stdout and stderr streams are not lost.

Options:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --config-file=CONFIGFILE
                        Config file (default
                        /home/oliverh/cylc/conf/lockserver.conf)
```

B.2.29 log

```
Usage: cylc [info] log [options] SUITE
Print, or view in $EDITOR, local suite log files, with optional
filtering. This command is a convenience: you don't need to
know the log file location on disk.

Arguments:
  SUITE - Target suite.

Options:
  -h, --help            show this help message and exit
  -p, --print           Print the suite log file location and exit. This is
                        equivalent to 'cylc get-config SUITE cylc logging
                        directory'.
  -t TASK, --task=TASK  Filter the log for messages from a specific task
  -f RE, --filter=RE    Filter the log with a Python-style regular expression
                        e.g. '\[(foo|bar).*(started|succeeded)'
  -r INT, --rotation=INT
                        Rotation number (to view older, rotated logs)
  -e, --edit            View a temporary copy of the (filtered) log file in
                        $EDITOR instead of printing it to stdout.
```

B.2.30 maxrunahead

```
Usage: cylc [control] maxrunahead [options] SUITE [HOURS]

Change the suite runahead limit in a running suite. This is the number of
hours that the fastest task is allowed to get ahead of the slowest. If a
task spawns beyond that limit it will be held back from running until the
slowest tasks catch up enough. WARNING: if you omit HOURS no limit will
be used!

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.
   HOURS                New runahead limit (no limit if omitted).

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.31 message

```
Usage: cylc [task] message [options]

This is part of the cylc external task interface.
```

```
Report completed outputs, progress, or any other messages.

Options:
  -h, --help            show this help message and exit
  -p PRIORITY           message priority: NORMAL, WARNING, or CRITICAL;
                        default NORMAL.
  --next-restart-completed
                        Report next restart file(s) completed
  --all-restart-outputs-completed
                        Report all restart outputs completed at once.
  --all-outputs-completed
                        Report all internal outputs completed at once.
```

B.2.32 monitor

```
Usage: cylc [info] monitor [options] SUITE

A terminal-based suite monitor that updates the current state of all
tasks in real time. It is effective even for quite large suites if
'--align' is not used. Being a passive monitor that cannot intervene in
a suite's operation, it is allowed to monitor suites owned by others
and/or running on remote hosts.

Arguments:
   SUITE                Target suite.

Options:
  -h, --help            show this help message and exit
  -o USER, --owner=USER
                        Owner of the target suite (defaults to $USER).
  --host=HOST           Cylc suite host (defaults to local host).
  -f, --force           (No effect; for consistency with interactive commands)
  --debug               Turn on exception tracebacks.
  -a, --align           Align columns by task name. This option is only useful
                        for small suites.
```

B.2.33 nudge

```
Usage: cylc [control] nudge [options] SUITE

Cause the cylc task processing loop to be invoked in a running suite.

This happens automatically when the state of any task changes such that
task processing (dependency negotation etc.) is required, or if a
clock-triggered task is ready to run.

The main reason to use this command is to update the "estimated time till
completion" intervals shown in the tree-view suite control GUI, during
periods when nothing else is happening.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.

Options:
  -h, --help   show this help message and exit
  -f, --force  (No effect; for consistency with interactive commands)
  --debug      Turn on exception tracebacks.
```

B.2.34 ping

```
Usage: cylc [discover] ping [options] SUITE

Check that SUITE is running.

Arguments:
SUITE             Target suite.
```

```
Arguments:
   SUITE                Target suite.

Options:
  -h, --help            show this help message and exit
  -o USER, --owner=USER
                        Owner of the target suite (defaults to $USER).
  --host=HOST           Cylc suite host (defaults to local host).
  -f, --force           (No effect; for consistency with interactive commands)
  --debug               Turn on exception tracebacks.
  --print-ports         Print cylc's configured port range.
```

B.2.35 print

```
Usage: cylc [db] print [--central] [FILTER]

Print private or central suite registrations.

FILTERING:
   (a) The filter patterns are Regular Expressions, not shell globs, so
the general wildcard is '.*' (match zero or more of anything), NOT '*'.
   (b) For our purposes there is an implicit wildcard at the end of each
pattern ('foo' is the same as 'foo.*'); use the string end marker to
stop this ('foo$' matches only literal 'foo').

Options:
  -h, --help     show this help message and exit
  -c, --central  Target the the central database.
  -t, --tree     Print registrations in nested tree form.
  -p, --pretty   Use unicode box drawing characters in tree views.
  -a, --align    Align columns.
  -x             don't print suite definition directory paths.
  -y             Don't print suite titles.
  --fail         Fail (exit 1) if no matching suites are found.
```

B.2.36 purge

```
Usage: cylc [control] purge [options] SUITE TASK STOP

Remove an entire tree of dependent tasks from a running suite. The root
task will be forced to spawn and will then be removed, then so will
every task that depends on it, and every task that depends on those, and
so on until the given stop cycle time.

WARNING: THIS COMMAND IS DANGEROUS but in case of disaster you can
restart the suite from the automatic pre-purge state dump (the filename
will be logged by cylc before the purge is actioned.)

UNDERSTANDING HOW PURGE WORKS: cylc identifies tasks that depend on
the root task, and then on its downstream dependents, and then on
theirs, etc., by simulating what would happen if the root task were to
trigger: it artificially sets the root task to the "succeeded" state
then negotatiates dependencies and artificially sets any tasks whose
prerequisites get satisfied to "succeeded"; then it negotiates
dependencies again, and so on until the stop cycle is reached or nothing
new triggers. Finally it marks "virtually triggered" tasks for removal.
Consequently:
 * Dependent tasks will only be identified as such if they have already
   spawned into the root cycle, otherwise they will be missed by the
   purge. To avoid this, wait until all tasks that depend on the root
   have caught up to it before purging.
 * If you purge a task that has already finished, only it and its own
   successors will be purged (other downstream tasks will already have
   triggered if they were able to).
[development note: post cylc-3.0 we should be able to identify tasks
that depend on the purge root by using the suite dependency graph, even
if they have not spawned into the cycle time of the root yet.]

You must be the owner of the target suite to use this command.
```

```
Arguments:
   SUITE                 Target suite.
   TASK                  Task (NAME%CYCLE) at which to start the purge.
   STOP                  Cycle time (inclusive!) at which to stop purging.

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.37 refresh

```
Usage: cycl [db] refresh [options] SUITE

Check a suite database for invalid registrations (no suite definition
directory or suite.rc file) and refresh suite titles in case they have
changed since the suite was registered (for the central database this
also updates the titles of suites owned by others).

Arguments:
    SUITE      - suite name, or match pattern

Options:
  -h, --help        show this help message and exit
  -c, --central     Print suite registrations from the central database.
  -u, --unregister  Automatically unregister invalid registrations.
  -v, --verbose     Print extra information.
```

B.2.38 register

```
Usage: cycl [db] register [options] SUITE PATH

Register a suite in your private suite database.

Cylc commands target suites via names registered in a suite database so
that you don't need to continually re-type the actual location of the
suite definition directory on disk. Suite names are hierarchical like
directory paths but delimited by '.' (foo.bar.baz), allowing suites to
be grouped and organised into tree-like structures. Groups are virtual
and do not need to be explicitly created before use.

Legal name characters are letters, digits, underscore, and hyphen. The
period '.' is the registration hierarchy delimiter. Colon cannot be used
because of the potential for trouble with PATH variables if the suite
name is used in certain directory paths.

EXAMPLES, for suite definition directories $TMPDIR/(one,two,three,four):

% cylc db reg bob         $TMPDIR/one
% cylc db reg foo.bag     $TMPDIR/two
% cylc db reg foo.bar.baz $TMPDIR/three
% cylc db reg foo.bar.waz $TMPDIR/four

% cylc db pr '^foo'             # print in flat form
  bob         | "Test Suite One"   | /tmp/oliverh/one
  foo.bag     | "Test Suite Two"   | /tmp/oliverh/two
  foo.bar.baz | "Test Suite Four"  | /tmp/oliverh/three
  foo.bar.waz | "Test Suite Three" | /tmp/oliverh/four

% cylc db pr -t '^foo'          # print in tree form
  bob        "Test Suite One"   | /tmp/oliverh/one
  foo
   |-bag     "Test Suite Two"   | /tmp/oliverh/two
   `-bar
     |-baz   "Test Suite Three" | /tmp/oliverh/three
     `-waz   "Test Suite Four"  | /tmp/oliverh/four

Arguments:
```

```
     SUITE   -  The new hierarchical suite registration name.
     PATH    -  A cylc suite definition directory.

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Print extra information.
```

B.2.39 release

```
Usage: cylc [control] release|unhold [options] SUITE [TASK]

Release a suite or a single task from a hold, allowing it run as normal.
Putting a suite on hold stops it from submitting any tasks that are
ready to run, until it is released. Putting a waiting task on hold
prevents it from running and spawning successors, until it is released.

See also 'cylc [control] hold'.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.
   TASK                 Task to release (NAME%YYYYMMDDHH)

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.40 remove

```
Usage: cylc [control] remove|kill [options] SUITE TARGET

Remove a single task, or all tasks with a common TAG (cycle time or
asynchronous 'a:INT'), from a running suite.

Target tasks will be forced to spawn successors before being removed if
they have not done so already, unless you use '--no-spawn'.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.
   TARGET      NAME%CYCLE or NAME%a:TAG to remove a single task;
               CYCLE or a:TAG to remove all tasks with the same tag.

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
  --no-spawn   Do not spawn successors before removal.
```

B.2.41 reregister

```
Usage: cylc [db] reregister|rename [options] SOURCE TARGET

Change the hierarchical registration name of a suite or group.
Example:
  cylc db rereg foo.bar.baz test.baz

Arguments:
    SOURCE, TARGET  - suite or group names.

Options:
  -h, --help     show this help message and exit
  -c, --central  Target the central suite database.
  -v, --verbose  Print extra information.
```

B.2.42 reset

```
Usage: cylc [control] reset [options] SUITE TASK

Force a task's state to:
 1/ 'ready' .... (default)   ...... all prerequisites satisfied (default)
 2/ 'waiting' .. (--waiting) ...... prerequisites not satisfied yet
 3/ 'succeeded'  (--succeeded) .... all outputs completed
 4/ 'failed' ... (--failed) ....... use to test failure recovery

Resetting a task to 'ready' will cause it to trigger immediately unless
the suite is held, in which case the task will trigger when normal
operation is resumed.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                 Target suite.
   TASK                  The target task.

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
  --ready      Force task to the 'ready' state.
  --waiting    Force task to the 'waiting' state.
  --succeeded  Force task to 'succeeded' state.
  --failed     Force task to 'failed' state.
```

B.2.43 restart

```
Usage: cylc [control] restart [options] SUITE [FILE]

Restart a cylc suite from a previous state. Tasks in the 'submitted',
'running', or 'failed' states will immediately be resubmitted at
start up unless you specify '--no-reset'. Any 'held' tasks will be
released unless you specify the '--no-release' option. A final cycle
time that was set prior to shutdown will be ignored on restart unless
you specify '--keep-finalcycle'

By default the suite will restart from the suite state dump file, which is
updated whenever a task changes state and thus records the most recent
previous state of the suite. However, cylc also records a special named
state dump, and logs its filename, before actioning any intervention
command, and you can choose to restart from one of these (just
cut-and-paste the filename from the log to the command line).

NOTE: daemonize important suites with the POSIX nohup command:
    nohup cylc [con] restart SUITE YYYYMMDDHH > suite.out 2> suite.err &

Arguments:
   SUITE                 Target suite.
   FILE                  Optional non-default state dump file (assumed to
                         reside in the suite state dump directory unless
                         you give the full path).

Options:
  -h, --help             show this help message and exit
  --no-reset             Do not reset failed tasks to ready on restarting
  --no-release           Do not release held tasks on restarting.
  --keep-finalcycle      Do not ignore a previously set final cycle time on
                         restarting
  --until=YYYYMMDDHH     Shut down after all tasks have PASSED this cycle time.
  --hold                 Hold (don't run tasks) immediately on starting.
  --hold-after=YYYYMMDDHH
                         Hold (don't run tasks) AFTER this cycle time.
  -s, --simulation-mode
                         Use dummy tasks that masquerade as the real thing, and
                         accelerate the wall clock: get the scheduling right
                         without having to run the real suite tasks.
  --fail=NAME%YYYYMMDDHH
                         (SIMULATION MODE) get the specified task to report
```

```
                        failure and then abort.
  --debug               Turn on 'debug' logging and full exception tracebacks.
  --timing              Turn on main task processing loop timing, which may be
                        useful for testing very large suites of 1000+ tasks.
  --gcylc               (DO NOT USE THIS OPTION).
```

B.2.44 run

```
Usage: cylc [control] run|start [options] SUITE [START]

Start a suite running at a specified initial cycle time. To restart a
suite from a previous state (which may contain tasks at multiple cycle
times), see 'cylc restart SUITE'.

There are three start up modes that differ in how initial intercycle
dependencies are handled (at start up there is no previous cycle for
tasks, such as warm cycled forecast models, that normally depend on
tasks in previous cycles):
  1/ Cold start (the default) -- start from scratch with special tasks
  2/ Warm start (-w,--warm)   -- assume a previous cycle
  3/ Raw  start (-r,--raw)    -- assume nothing
These are all the same for a suite that has no intercycle dependencies.
Otherwise, cold start tasks should be defined in the suite.rc file:
# SUITE.RC:
[scheduling]->[[special tasks]]->cold-start = task1, task2, ...

1/ COLD START -- start from scratch, with special one off "cold start
tasks" to satisfy the initial dependencies of any tasks that normally
depend on tasks from a previous cycle (most notably the restart
dependencies of warm cycled forecast models). Cold start tasks
can run real processes (e.g. a special forecast to generate the initial
restart files for a warm cycled model) or, with no task command defined,
just act as a dummy proxy for some external process that has to occur
before the suite is started (e.g. a spinup run of some kind). For a
suite with no intercycle dependencies there is no distinction between
the cold, warm, and raw start methods. For a cold start, each task,
including designated cold start tasks, starts in the 'waiting' state
(i.e. prerequisites not satisfied) at the initial cycle time or at the
next valid cycle time thereafter for the task.

2/ WARM START -- this assumes that there was a previous cycle (from a
previous suite run - in which case a 'cylc restart' may be an option
too) or that files required from previous cycles have been put in place
by some external means. Start up is as for a cold start, except that
designated cold start tasks are inserted in the 'succeeded' state (i.e.
outputs completed).

3/ RAW START -- this starts the suite as if in mid run without any
special handling - any tasks that depend on previous cycles will have to
be triggered manually. Start up is as for a cold start, except that
designated cold start tasks are excluded from the suite.

NOTE: daemonize important suites with the POSIX nohup command:
  $ nohup cylc [con] run SUITE [START] > suite.out 2> suite.err &

Arguments:
   SUITE                Target suite.
   START                Initial cycle time. This is optional if defined in the
                        suite.rc file, in which case the command line
                        takes priority (and a suite.rc final cycle time
                        will be ignored), and is not required if the
                        suite contains no cycling tasks.

Options:
  -h, --help            show this help message and exit
  -w, --warm            Warm start the suite
  -r, --raw             Raw start the suite
  --until=YYYYMMDDHH    Shut down after all tasks have PASSED this cycle time.
  --hold                Hold (don't run tasks) immediately on starting.
  --hold-after=YYYYMMDDHH
```



```
                        Hold (don't run tasks) AFTER this cycle time.
  -s, --simulation-mode
                        Use dummy tasks that masquerade as the real thing, and
                        accelerate the wall clock: get the scheduling right
                        without having to run the real suite tasks.
  --fail=NAME%YYYYMMDDHH
                        (SIMULATION MODE) get the specified task to report
                        failure and then abort.
  --debug               Turn on 'debug' logging and full exception tracebacks.
  --timing              Turn on main task processing loop timing, which may be
                        useful for testing very large suites of 1000+ tasks.
  --gcylc               (DO NOT USE THIS OPTION).
```

B.2.45 scan

```
Usage: cylc [discover] scan [options]

Scan cylc ports for running suites and lockservers. Connection Denied
indicates a secure suite owned by somebody else.

Options:
  -h, --help      show this help message and exit
  --host=HOST     Cylc suite host (defaults to localhost).
  --print-ports   Print cylc's configured port range.
```

B.2.46 scp-transfer

```
Usage: cylc [util] scp-transfer [options]

An scp wrapper for transferring a list of files and/or directories
at once. The source and target scp URLs can be local or remote (scp
can transfer files between two remote hosts). Passwordless ssh must
be configured between the source and target hosts.

Environment inputs:
$SRCE  - list of sources (files or directories) as scp URLs.
$DEST  - parallel list of targets as scp URLs.

We let scp determine the validity of source and target URLs.
Target directories are created pre-copy if they don't exist.

Options:
 -v      - verbose: print scp stdout.
 --help - print this usage message.
```

B.2.47 search

```
Usage: cylc [prep] search|grep [options] PATTERN SUITE

Find matches to PATTERN in the suite.rc file, and any contained
include-files, and in any files in the suite bin directory. For the
suite.rc and include-files, matches are reported by file line number and
suite section (even if include files are nested).

An unquoted list of PATTERNs will be converted to an OR'd pattern.

Note that the order of command line arguments conforms to grep command usage
('grep PATTERN FILE') not normal cylc command usage ('command SUITE ARGS').
However, if the second argument is not found to be a registered suite, an
attempt will be made to action the command with the arguments reversed.

Note that this command performs a text search on the suite definition,
rather than search all item names and values in the data structure that
results from parsing the suite definition. Both types of search may be
implemented in the future.

For case insenstive matching use '(?i)PATTERN'.

Arguments:
```

```
  SUITE   - Suite database registration
  PATTERN - (list of) Python-style regular expression(s).


Options:
  -h, --help     show this help message and exit
  -x             Do not search in the suite bin directory
  -c, --central  Target the central database.
```

B.2.48 show

```
Usage: cylc [info] show [options] SUITE [NAME[%TAG]]

Print global or task-specific information from a running suite: title
and task list, task descriptions, current state of task prerequisites
and outputs and, for clock-triggered tasks, whether the delayed start
time is up yet.

Arguments:
   SUITE                    Target suite.
   NAME                     Name of a task type.
   TAG                      Cycle time, or asynchronous tag 'a:INT'.

Options:
  -h, --help                show this help message and exit
  -o USER, --owner=USER
                            Owner of the target suite (defaults to $USER).
  --host=HOST               Cylc suite host (defaults to local host).
  -f, --force               (No effect; for consistency with interactive commands)
  --debug                   Turn on exception tracebacks.
```

B.2.49 started

```
Usage: cylc [task] started [options]

This is part of the cylc external task interface.

Acquire a lock from the lockserver, and report that I have started.

Options:
  -h, --help  show this help message and exit
```

B.2.50 stop

```
Usage: cylc [control] stop|shutdown [options] SUITE [STOP]

1/ Shut down a suite when all currently running tasks have finished.
   No other tasks will be submitted to run in the meantime.

2/ With [STOP], shut down a suite AFTER on of the following events:
    a/ all tasks have passed the TAG STOP (cycle time or async tag)
    b/ the clock time has reached STOP (YYYY/MM/DD-HH:mm)
    c/ the task STOP (TASK%TAG) has finished

3/ With [--now], shut down immediately, regardless of tasks still running.
   WARNING: beware of orphaning tasks that are still running at shutdown;
   these may need to be killed manually, and they will (by default) be
   resubmitted if the suite is restarted.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                    Target suite.
   STOP                     a/ task TAG (cycle time or 'a:INTEGER'), or
                            b/ YYYY/MM/DD-HH:mm (clock time), or
                            c/ TASK (task ID).

Options:
  -h, --help    show this help message and exit
```

```
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
  --now        Shut down immediately; see WARNING above.
```

B.2.51 submit

```
Usage: cylc [task] submit|single [options] SUITE TASK

Submit a single task to run exactly as it would be submitted by its
parent suite, in terms of both execution environment and job submission
method. This can be used as an easy way to run single tasks for any
reason, but it is particularly useful during suite development.

If the parent suite is running at the same time and it has acquired an
exclusive suite lock (which means you cannot running multiple instances
of the suite at once, even under different registrations) then the
lockserver will let you 'submit' a task from the suite only under the
same registration, and only if the task is not locked (which would mean
the same task, NAME%CYCLE, is currently running in the suite).

Arguments:
   SUITE            Registered name of the target suite.
   TASK             Identity of the task to run (NAME%YYYYMMDDHH).

Options:
  -h, --help      show this help message and exit
  -d, --dry-run   Generate the cylc task execution file for the task and report
                  how it would be submitted to run.
  --scheduler     (EXPERIMENTAL) tell the task to run as a scheduler task, i.e.
                  to attempt to communicate with a task proxy in a running cylc
                  suite (you probably do not want to do this).
```

B.2.52 succeeded

```
Usage: cylc [task] succeeded [options]

This is part of the cylc external task interface.

Release my lock to the lockserver, and report that I have succeeded.

Options:
  -h, --help  show this help message and exit
```

B.2.53 template

```
Usage: cylc [util] template [options] STRING [CYCLE]

Compute cycle time-dependent file names based on template strings, and
print the result to stdout. The base cycle time can be taken from the
environment or the command line, and offsets can be computed a la
'cylc cycletime'. Suite file naming conventions can thus be encoded
in template variables defined in the suite config file.

STRING can be the name of an environment variable containing a template
string OR the template string itself. In the template, 'YYYY', 'MM',
'DD', or 'HH' will be replaced with computed cycle time components.

   $ export CYLC_TASK_CYCLE_TIME=2010082318

   $ cylc template -s 6 fooHH-YYYYMMDD.nc #explicit name convention
  foo12-20100823.nc

   $ export NCONV=fooHH-YYYYMMDD.nc    # (as if from system config file)

   $ cylc template NCONV #............implicit filename convention
  foo18-20100823.nc
   $ cylc template -s 6 NCONV_FOO #..same, with current cycle offset
  foo12-20100823.nc
```



```
Note that 'cylc template NCONV' and 'cylc template $NCONV' will
generate the same result, but the former is preferred so that we can
detect accidental use of undefined environment variables.

Arguments:
   STRING       A template string, or the name of an env variable that
                contains a template.
   CYCLE        (YYYYMMDDHH) defaults to $CYLC_TASK_CYCLE_TIME.

Options:
  -h, --help            show this help message and exit
  -s HOURS, --subtract=HOURS
                        Subtract HOURS from CYCLE
  -a HOURS, --add=HOURS
                        Add HOURS to CYCLE
  -o HOURS, --offset=HOURS
                        Apply an offset of +/-HOURS to CYCLE
```

B.2.54 test-db

```
USAGE: cylc [admin] test-db [--help]
This is a thorough test of cylc suite registration
database functionality (private and central).
Options:
  --help   Print this usage message.
```

B.2.55 test-suite

```
USAGE: cylc [admin] test-suite [--help]

Run an automated test of core cylc functionality using a new copy of
'/home/oliverh/cylc/examples/admin/test'. This should be used to check that new
developments in the cylc codebase have not introduced serious bugs.
The test runs a suite registered as 'test'; to watch its progress
use 'cylc view'. Aside from timing differences results should be the
same in live or simulation mode.

Currently the test does the following:
  - Copies the 'intro' example suite definition directory;
  - Registers the new suite as ;
  - Starts the suite at T0=06Z, with task X set to fail at 12Z;
  - Unlocks the running suite;
  - Sets a stop time at 12Z (i.e. T0+30 hours);
  - Waits for the suite to stall as result of X failing;
  - Inserts a new coldstart task at 06Z (T0+24 hours);
  - Purges the failed X and dependants through to 00Z (T0+18 hours)
    inclusive, which allows the suite to get going again at 06Z;
  - Waits for the suite to shut itself down at the 12Z stop time.
  - Run a single task (called prep) from the suite with submit.

Options:
  -h, --help   Print this help message and exit.
```

B.2.56 trigger

```
Usage: cylc [control] trigger [options] SUITE TASK

Force a task to trigger immediately (unless the suite is paused,
in which case it will trigger when normal operation is resumed).
This is just a special case of the 'reset' command.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.
   TASK                 The target task.

Options:
  -h, --help   show this help message and exit
```

```
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.57 unblock

```
Usage: cylc [control] unblock [options] SUITE

A blocked suite refuses to comply with intervention commands until
deliberately unblocked. This is a crude security measure to guard
against accidental intervention in your own suites. It may be a
useful aid when running multiple suites at once.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.58 unregister

```
Usage: cylc [db] unregister [OPTIONS] SUITE

Unregister suites from your private database, or remove your suites from
the central suite database. This does not delete the corresponding suite
definition directories unless you use the '--delete' option.

If you accidentally unregister a running suite, just register it again
under the original name to regain access to it.

Arguments:
    SUITE - suite or group name, or filter expression.

Options:
  -h, --help     show this help message and exit
  -d, --delete   Delete the suite definition directory too (!DANGEROUS!).
  -f, --force    Don't ask for confirmation before deleting suites.
  -c, --central  Target the central suite database.
  --dry-run      Just show what I would do.
  --filter       A filtering regular expression. This must be explicitly
                 indicated for unregistering, for safety reasons.
  -v, --verbose  Print extra information.
```

B.2.59 validate

```
Usage: cylc [prep] validate [options] SUITE

(a) Parse and validate a suite config (suite.rc) file to check that
all entries conform to the $CYLC_DIR/conf/suiterc.spec specification.
(b) Attempt to instantiate a proxy object for each task in the suite.

IF THE SUITE.RC FILE USES INCLUDE-FILES: line numbers reported with
validation errors will be wrong because the parser sees an inlined
version of the file. You can use 'cylc prep inline SUITE' to trace errors to
the correct source line, although the extra information reported by the
validator should be sufficient to make this unnecessary.

Arguments:
   SUITE    - Suite database registration.

Options:
  -h, --help     show this help message and exit
  -c, --central  target the central database.
  -p, --pretty   Use unicode box drawing characters when printing the suite
                 runtime inheritance hierarchy.
  -v, --verbose  Print extra information from the validation process.
```

B.2.60 verbosity

```
Usage: cylc [control] verbosity [options] SUITE LEVEL

Change the logging priority level of a running suite.  Only messages at
or above the chosen priority level will be logged; for example, if you
choose 'warning', only warning, error, and critical messages will be
logged. The 'info' level is appropriate under most circumstances.

You must be the owner of the target suite to use this command.

Arguments:
   SUITE                Target suite.
   LEVEL                debug, info, warning, error, or critical

Options:
  -h, --help   show this help message and exit
  --debug      Turn on exception tracebacks.
  -f, --force  Do not ask for confirmation before acting.
```

B.2.61 warranty

```
USAGE: cylc [license] warranty [--help]
   Cylc is released under the GNU General Public License v3.0
This command prints the GPL v3.0 disclaimer of warranty.
Options:
  --help   Print this usage message.
```

C The Cylc Lockserver

Each cylc user can optionally run his/her own lockserver to prevent accidental invocation of multiple instances of the same suite or task at the same time. The suite and task locks brokered by the lockserver are analogous to traditional lock files, but they work across a network, even for distributed suites containing tasks that start executing on one host and finish on another.

Accidental invocation of multiple instances of the same suite or task at the same time can have serious consequences, so use of the lockserver should be considered for important operational suites, but it may be considered an unnecessary complication for general less critical usage, so it is currently disabled by default.

To enable the lockserver:

```
# SUITE.RC
use lockserver = True
```

The suite will now abort at start-up if it cannot connect to the lockserver. To start your lockserver daemon,

```
% cylc lockserver start
```

To check that it is running,

```
% cylc lockserver status
```

For detailed usage information,

```
% cylc lockserver --help
```

There is a command line client interface,

```
% cylc lockclient --help
```

for interrogating the lockserver and managing locks manually (e.g. releasing locks if a suite was killed before it could clean up after itself).

To watch suite locks being acquired and released as a suite runs,

```
% watch cylc lockclient --print
```

D The Graph-Based Suite Control GUI

The graph-based suite control GUI has the advantage that it shows the structure of a suite very clearly as it evolves. It works remarkably well even for very large suites, on the order of one hundred tasks or more *but* on the downside, the graphviz engine does a new global optimization every time the graph changes, so the layout is often not very stable. This may or may not be a solvable problem - it seems likely that making continual incremental changes to an existing graph without redoing the global layout would inevitably result in some kind of horrible mess.

The following features of the graph-based control GUI go a long way toward mitigating the changing layout problem:

- The disconnect button can be used to temporarily prevent the graph from changing as the content of the suite changes (and in real time operation suites evolve quite slowly anyway)
- Right-click on a task and choose the "Focus" option to restrict the graph display to that task's cycle time. Anything interesting happening in other cycles will show up as disconnected rectangular nodes to the right of the graph (and you can click on those to intantly refocus to their cycles).
- Task filtering is the ultimate quick route to temporarily focusing on just the tasks you're interested in (but this will destroy the graph structure, to state the obvious).

In future cylc releases we plan to keep the graph centered, after layout changes, on the most recently clicked-on task.

E Simulation Mode

If you start a suite in simulation mode (a command line option for the cold-, warm-, raw-, and re-start commands, and a checkbutton in the gcylc suite start panel) then cylc will run on an accelerated clock and submit dummy programs instead of the real tasks. These masquerade as the real tasks by reporting the correct outputs complete after a short interval, reporting success, and then exiting. This is essentially indistinguishable, to cylc, from real operation. Simulation mode was, and remains, an important aid to cylc development because it allows testing of every aspect of scheduling without having to run real tasks in real time. Prior to cylc-3 it was also a useful aid to suite development - a simulation run would quickly identify any mismatch between the user-defined prerequisites and outputs across the suite, so you could get the scheduling right without running the real tasks. Post cylc-3.0 this is less important because task prerequisites and outputs are implicitly defined by the dependency graph and, short of a bug in cylc, the suite will run according to the graph.

E.1 Clock Rate and Offset

Simulation mode suites run on an accelerated clock so that you can test things very quickly. You can set the clock rate and offset with respect to the initial cycle time with options to the `cylc run` command. An offset of 10 hours, say, means that the simulation mode clock starts at 10 hours prior to the suite's initial cycle time. You can thus simulate the behaviour of the suite as it catches up from a delay and transitions to real time operation. By default, the clock runs at a rate of 10 seconds real time to 1 hour suite time, and with an initial offset of 10 hours.

E.2 Switching A Suite Between Simulation And Live Modes?

The scheduler mode (simulation or live) is recorded in the suite state dump file. *Cylc will not let you restart a simulation mode suite in live, or a live mode suite in simulation mode* - any attempt to do the former must certainly be a mistake, and doing the latter, while feasible, would corrupt your live suite state dump and turn it over to simulation mode. Note, however, that if you really want to run the current state of a live suite forward in simulation mode, all you need to do is this:

1. back up the live state dump (or take note of the filename of the relevant automatic dump when you do the final suite shutdown intervention).
2. edit the mode line in the state dump file, and restart the suite in simulation mode.
3. later, restart the live suite from the pre simulation mode state dump backup

Cylc can also create an instant dummy mode clone of the current state of any running or stopped suite, but this "practice mode" has been disabled in the current release pending testing.

F Cylc Development History

F.1 Pre-3.0

Early versions of cylc were focused on developing and testing the new scheduling algorithm, and the suite design interface at the time was essentially the quickest route to that end. A suite was a collection of "task definition files" that encoded the prerequisites and outputs of each task in a direct reflection of cylc's internal task proxies. This way of defining suites exposed cylc's self-organising nature to the user, and it did have some nice properties. For instance a group of tasks could be transferred directly from one suite to another by simply copying the taskdef files over (and checking that prerequisite and output messages were consistent with the new suite). However, ensuring consistency of prerequisites and outputs across a large suite could be tedious; a few edge cases associated with suite start-up and forecast model restart dependencies were, arguably, difficult to understand; and the global structure of a suite was not readily apparent until run time (although to counter this cylc 2.x could generate run-time resolved dependency graphs very quickly in simulation mode).

F.2 Version 3.0

Version 3.0 implemented an entirely new suite design interface in which one defines the suite dependency graph, execution environment, and command scripting for each task, in a single structured, validated, configuration file - the suite.rc file. This *really* makes suite structure apparent at a glance, and task prerequisites and outputs (and some other important parameters besides) no longer need to be specified by the user because they are implied by the graph.

F.3 Version 4.0

Version 4.0 has the following major improvements over cylc-3.x, along with many refinements:

- suite registration has been generalized from GROUP:NAME to a hierarchy of arbitrary depth, e.g foo.bar.baz, allowing suites to be organized in a tree-like structure.
- the suite.rc file has undergone some major housekeeping and, in particular, now defines a *namespace hierarchy* of task families and tasks, to allow common run time properties to be grouped naturally among related tasks.

G Pyro

Pyro (Python Remote Objects) is a widely used open source objected oriented Remote Procedure Call technology developed by Irmen de Jong.

Earlier versions of cylc used the Pyro Nameserver to handle marshalling of communication between client programs (tasks, commands, viewers, etc.) and their target suites. This worked well, but in principle it provided a route for one suite or user on the subnet to bring down all running suites by killing the nameserver. Consequently cylc now uses Pyro simply as a lightweight object oriented wrapper for direct network socket communication between client programs and their target suites - all suites are thus entirely isolated from one another.

H Acknowledgements

Bernard Miville and Phil Andrews (NIWA) for discussion, bug finding, and many suggestions that have improved cylc's usability and functionality; David Matthews and Matthew Shin (Met Office UK) for the same, and much code development as well.

I GNU GENERAL PUBLIC LICENSE v3.0

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

   The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
   (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
   (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
   (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
   (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

   A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

   A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or

(2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.