The Cylc Forecast Suite Metascheduler User Guide

Cylc Version 3.4.0-a

Hilary Oliver

August 5, 2011

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.

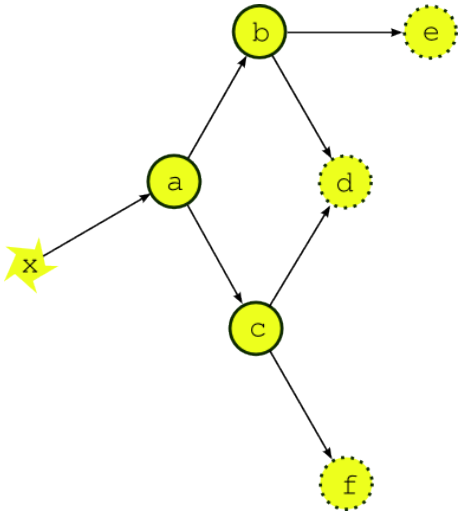

Figure 1: The dependency graph for a single forecast cycle of a simple example suite. Tasks *a, b,* and *c* represent forecast models, *d, e* and *f* are post processing or product generation tasks, and *x* represents external data that the upstream forecast model depends on.

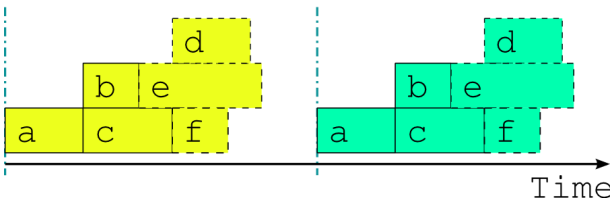

Figure 2: The optimal job schedule for two consecutive cycles of our example suite during real time operation, assuming that all tasks trigger off upstream tasks finishing completely. The horizontal extent of a task bar represents its execution time, and the vertical blue lines show when the external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

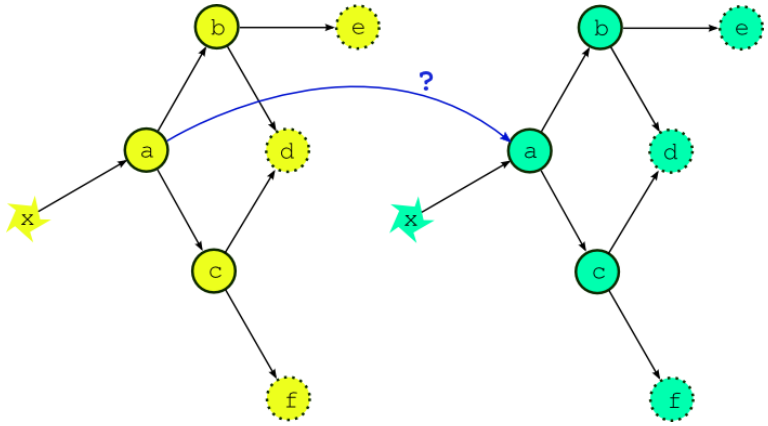

Figure 3: If the external driving data is available in advance, can we start running the next cycle early?

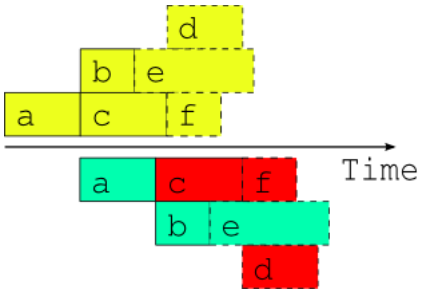

Figure 4: A naive attempt to overlap two consecutive cycles using the single-cycle dependency graph. The red shaded tasks will fail because of dependency violations (or will not be able to run because of upstream dependency violations).

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

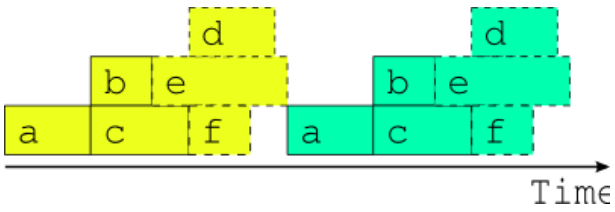

Figure 5: The best that can be done *in general* when intercycle dependence is ignored.

- imagine that task b in the first cycle is delayed for any reason *after* the second cycle has been 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 dependencies can be ignored because they are 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

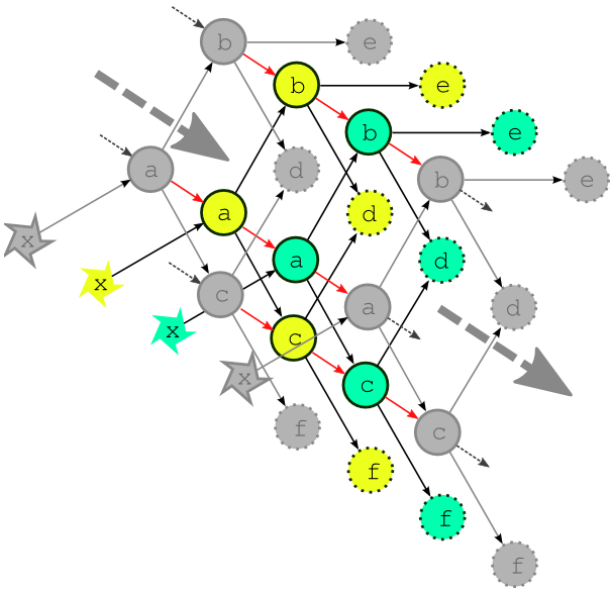

Figure 6: The complete dependency graph for the example suite, assuming the least possible intercycle dependence: the forecast models (a, b, and c) depend on their own previous instances. The dashed arrows show connections to previous and subsequent forecast cycles.

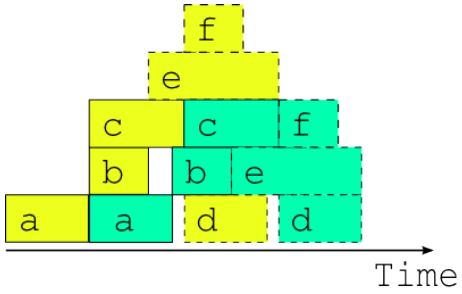

Figure 7: The optimal two cycle job schedule when the next cycle's driving data is available in advance, possible in principle when intercycle dependence is handled explicitly.

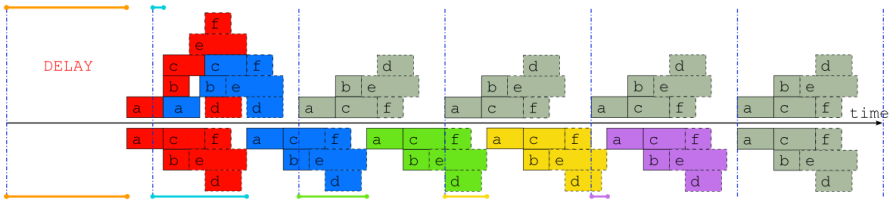

Figure 8: Job schedules for the example suite after a delay of almost one whole forecast cycle, when intercycle dependence is taken into account (above the time axis), and when they are not (below the time axis). The colored lines indicate the time that each cycle is delayed, and normal "caught up" cycles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

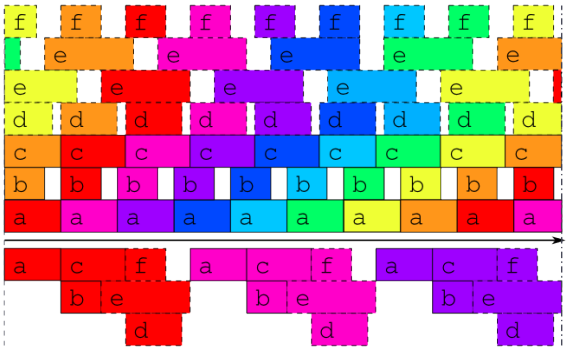

Figure 9: Job schedules for the example suite in case study mode, or after a long delay, when the external driving data are available many cycles in advance. Above the time axis is the optimal schedule obtained when the suite is constrained only by its true dependencies, as in Figure 3, and underneath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

cycles in advance. Task *a*, which as the most upstream forecast model is likely to be a resource 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

[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.

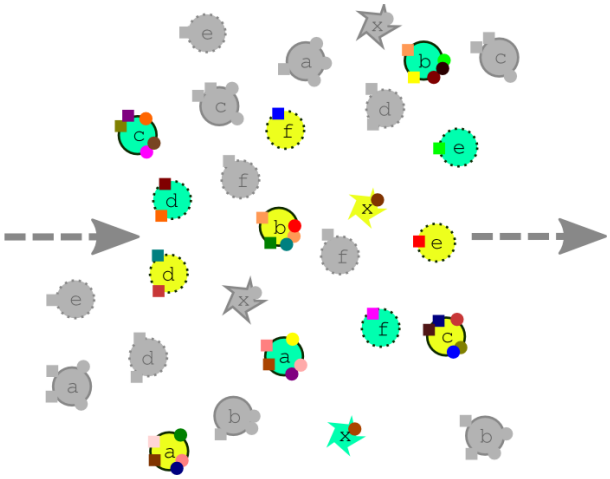

Figure 10: How cylc sees a suite, in contrast to the multicycle dependency graph of Figure 6. Task colors represent different cycle times, and the small squares and circles represent different prerequisites and outputs. A task can run when its prerequisites are satisfied by the outputs of other tasks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

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://www.xs4all.nl/ irmen/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)

[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.

- ConfigObj and Validate (http://www.voidspace.org.uk/python, BSD license)

2.2 Unpacking The Cylc Tarball

Cylc installs into a normal user account; just unpack the release tarball in the desired location.

2.3 Configuring Your Environment For Cylc

To gain access to cylc type the following at the command prompt:

```
$ export CYLC_DIR=/path/to/cylc/installation/
$ . $CYLC_DIR/environment.sh
```

Put this into your `.profile` login script to configure cylc access automatically. The variable `$CYLC_DIR` is required inside the environment script, so don't skip the export step. You should now be able to run cylc commands:

```
$ cylc --version
3.x.y
```

2.4 Creating 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 several 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 --central` on the command line.

```
$ cylc db print --central
 admin:examples:CUG1    "Cylc User Guide Example 1"  ~admin/cylc/examples/CUG1
 admin:examples:CUG73   "Cylc User Guide <SNIP>"     ~admin/cylc/examples/CUG7.3
 admin:examples:CUG74   "A two-task test suite"      ~admin/cylc/examples/CUG7.4
 admin:examples:CUG_QS1 "Cylc User Guide <SNIP> "    ~/dmin/cylc/examples/CUG5/one
 admin:examples:CUG_QS2 "Cylc User Guide <SNIP>"     ~admin/cylc/examples/CUG5/two
 admin:examples:CUG_QS3 "Cylc User Guide <SNIP>"     ~admin/cylc/examples/CUG5/three
 admin:examples:FFHook  "family failure hook <SNIP>" ~admin/cylc/examples/<SNIP>
 admin:examples:FFTask  "family failure clean <SNIP>"~admin/cylc/examples/<SNIP>
DONE
```

(some strings above have been truncated to reduce the line length)

2.5 Running The Automated Database Test

The command `cylc admin test-db` gives suite database functionality a work out - it registers the User Guide example suite under a new name and then manipulates it (by copying 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 Running The Automated Scheduler Test

The command `cylc admin test-suite` tests the cylc scheduler itself by running a suite registered as *examples:test* in the central database (by `cylc admin create-cdb`, above) 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 private database window, and opening up a suite control GUI.

The test-suite command does not currently detect failure of the running suite, it will just keep waiting. To see if this has happened (it shouldn't!) take a look at the file `test-suite.out` *that is generated by the test-suite script.* This will be addressed in a future cylc release.

2.7 Installing Everything Under Your Home Directory

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.

2.7.1 Cylc

Cylc is already designed to be installed into a normal user account - just unpack the cylc release tarball in the desired location, which we shall refer to below as `$CYLC_DIR`.

If you try to use cylc commands now you will get a warning that Pyro is not installed.

2.7.2 Download Pyro, Graphviz, And Pygraphviz

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).

2.7.3 Pyro

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

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

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

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

(The exact path will depend on your local Python environment). Add this to your `PYTHONPATH` by editing `$CYLC_DIR/environment.sh`, uncommenting the paragraph that begins with:

```
# FOR LOCAL INSTALLATION OF PYRO, GRAPHVIZ, AND PYGRAPHVIZ
```

and altering the paths under it to suit your local installation.

Now you should be able to source the cylc environment and get cylc to print its release version:

```
$ export CYLC_DIR=$HOME/cylc-3.3.3   # or whichever version you have
$ . $CYLC_DIR environment.sh
CONFIGURING THIS SHELL FOR /wdata/hilary/cylc-3.3.3/bin/cylc
Cylc release version: 3.3.3
$ cylc -v
3.3.3
```

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 script since updating its PYTHONPATH.

2.7.4 Create The Suite Database And Load The Example Suites

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.

2.7.5 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 your cylc environment script (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=.

2.7.6 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  # (if not done already)
$ 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 the PYTHONPATH variable in your cylc environment script.

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 script 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 script since updating its PYTHONPATH. Finally, if you have installed pygraphviz and configured your

PYTHONPATH properly 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.

2.7.7 What Next?

You should now have access to all cylc functionality. Create the central suite database and upload the example suites now, 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 5).

2.8 Upgrading To A New Cylc Version

This is as simple as unpacking the new cylc release and replacing the old example suites with the new ones in the central suite database.

In a future release we will make the central suite database installation-independent. Currently though it is held in the cylc installation directory under `$CYLC_DIR/CDB`, i.e. if you have multiple versions of cylc on the one host each will have its own central database. The following procedure takes the old central database over to a new installation, replacing the old example suites in it with the new ones:

1. Before upgrading (under the old cylc version), unregister the *examples* group from the central suite database.
2. Download the new cylc release tarball and unpack it.
3. Copy (via the shell) the central database from the old installation to the new.
4. Source the new cylc environment script.
5. Run `cylc admin create-cdb` to upload the example suites for the new release.

3 Cylc Screenshots

The following screenshots provide a glimpse of what the cylc user interface looks like.

- 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.

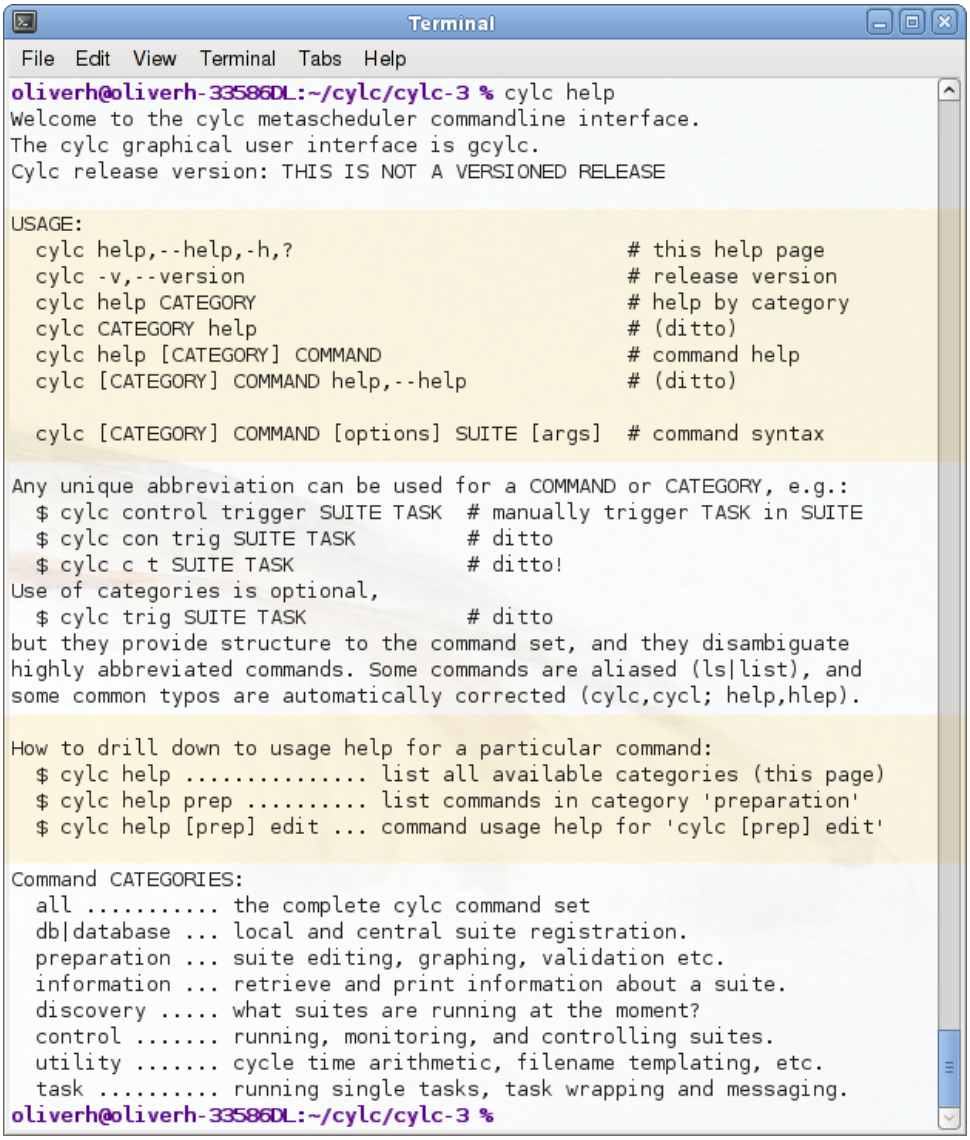

Figure 11: The cylc command line interface

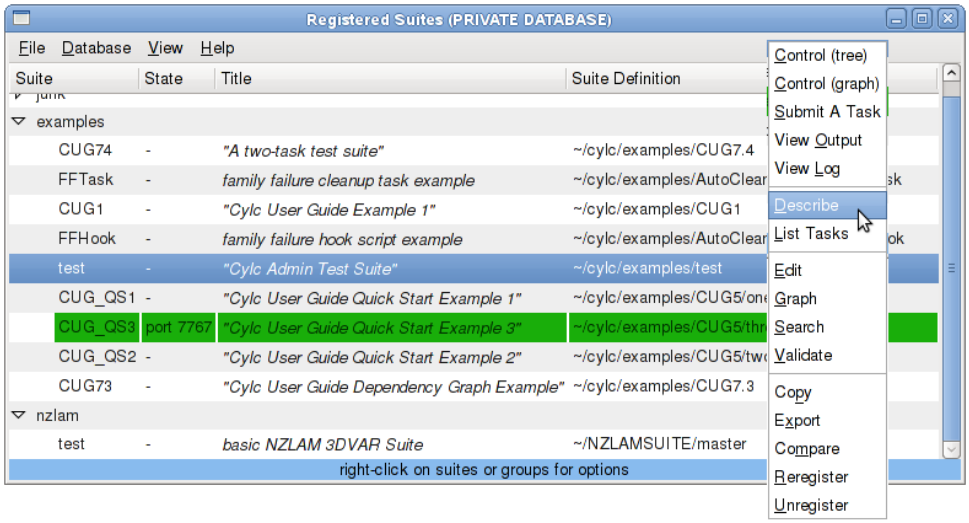

Figure 12: gcylc showing a private suite database.

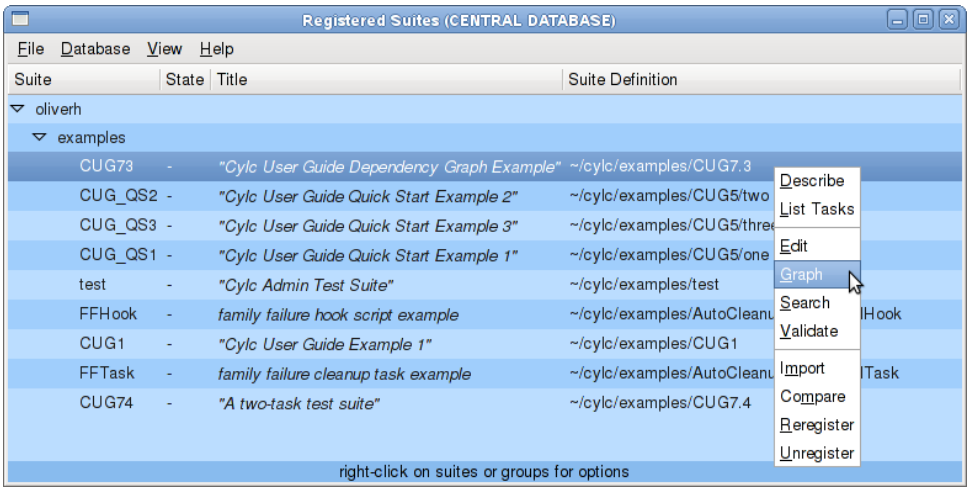

Figure 13: gcylc showing the central suite database.

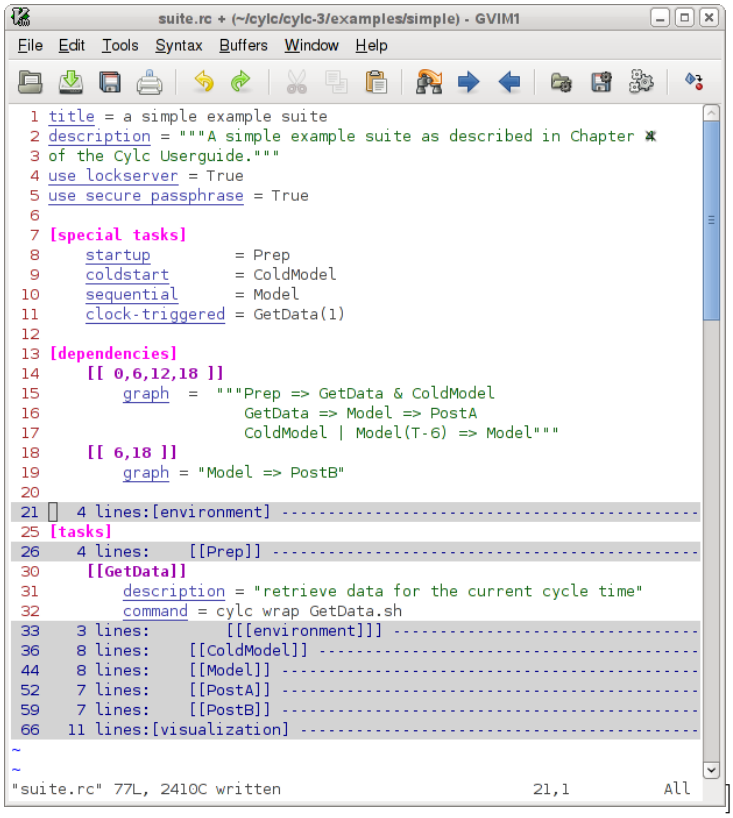

Figure 14: A simple cylc suite definition edited in *vim*

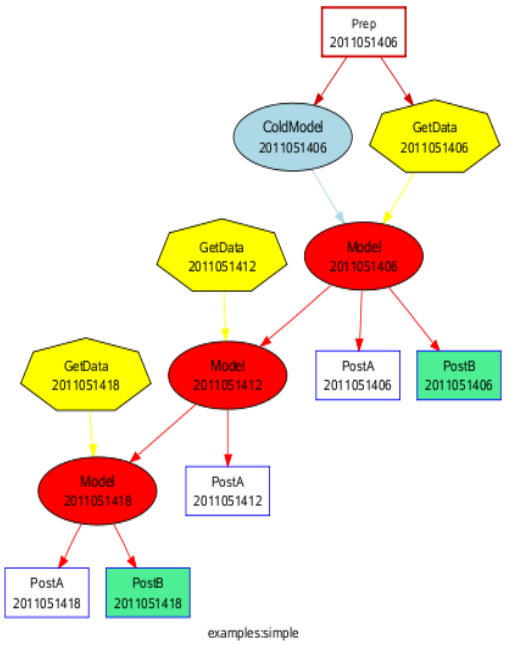

Figure 15: The suite definition of Figure 14 graphed by cylc.

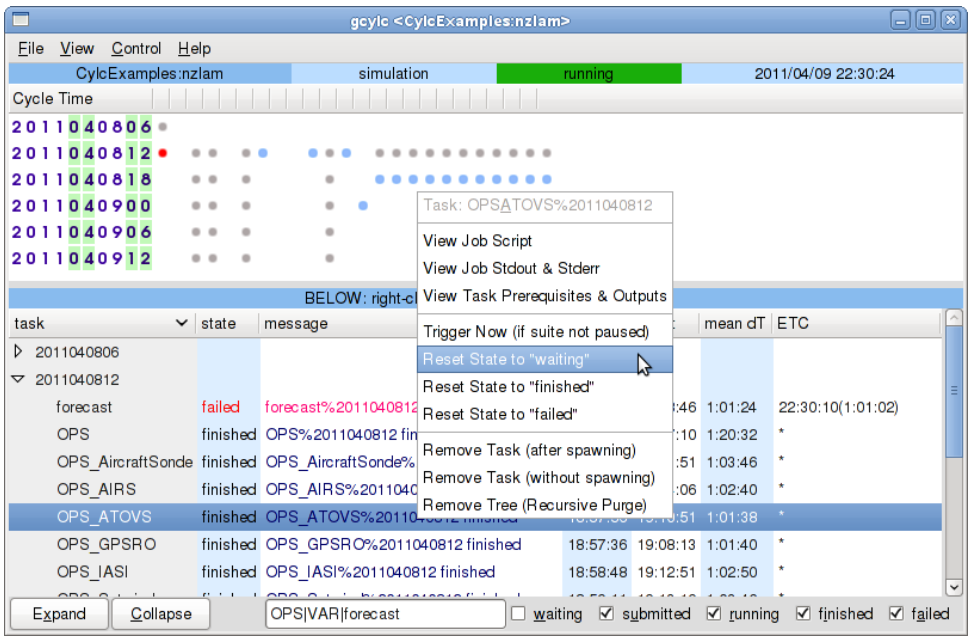

Figure 16: The suite control GUI, treeview interface.

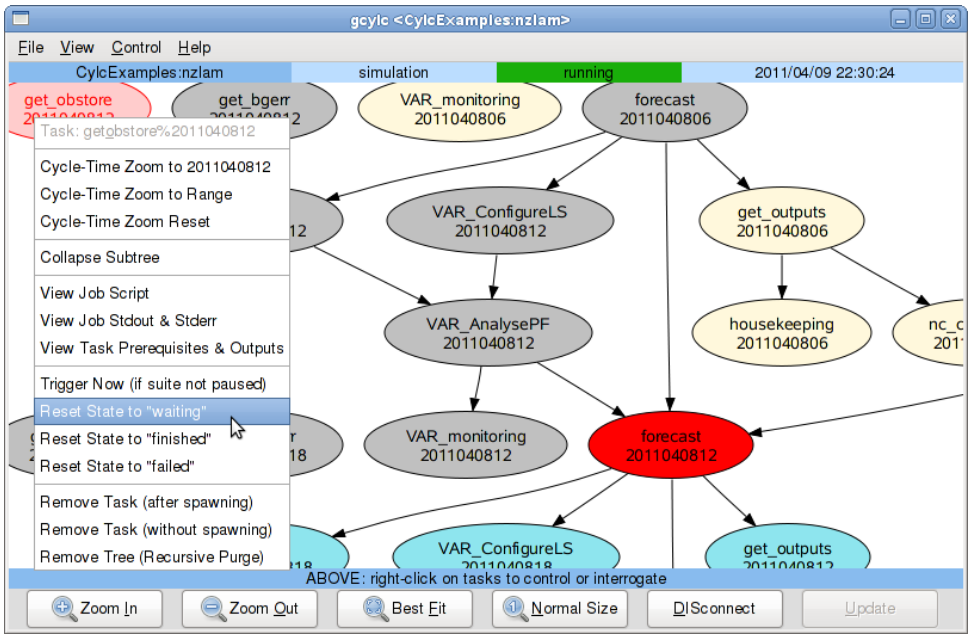

Figure 17: The suite control GUI, graph interface.

Figure 18: A large operational suite graphed by cylc.

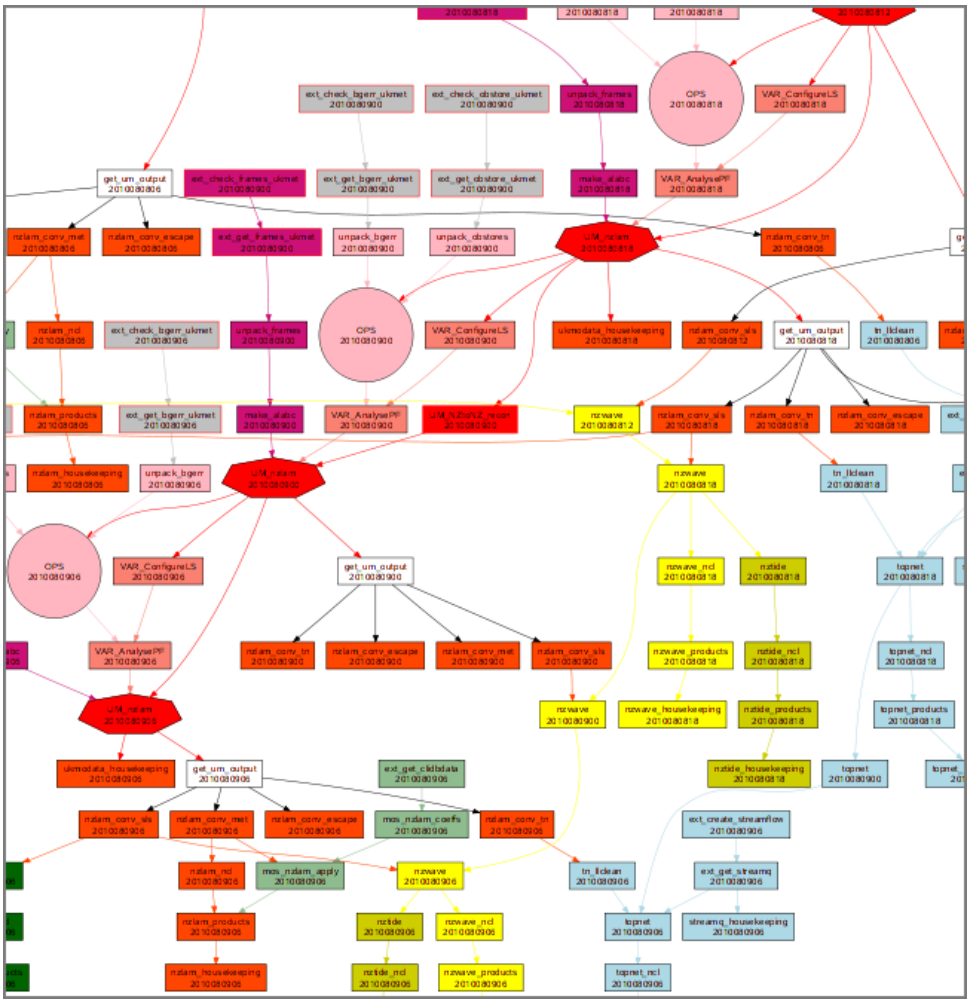

Figure 19: Another view of the large operational suite of Figure 18.

4 On The Meaning Of *Cycle Time* In Cylc

As explained in *Introduction: How Cylc Works* (Section 1): **cylc has no concept of global 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. **Cylc suites advance by means of each task spawning a successor at the next valid cycle time for the task**, not by incrementing a suite-wide forecast cycle.

However, it may still be convenient sometimes 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 a group may not all exist in the running suite at the same time. In other words, some tasks may pass through the "current cycle" (etc.) at different times as the suite evolves, particularly in delayed (catch up) operation.

5 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 during cylc installation,

```
$ cylc database print --central | grep Quick
 admin:examples:CUG_QS1 "<SNIP>Quick Start Example 1" ~admin/cylc/examples/CUG5/one
 admin:examples:CUG_QS2 "<SNIP>Quick Start Example 2" ~admin/cylc/examples/CUG5/two
 admin:examples:CUG_QS3 "<SNIP>Quick Start Example 3" ~admin/cylc/examples/CUG5/three
DONE
```

(suite titles have been truncated above to reduce line length).

5.1 New Features Not Yet Covered In the Quick Start Guide

- One-off asynchronous tasks (no associated cycle time).
- Repeating asynchronous tasks for parallel processing of satellite passes.

Asynchronous tasks, and their new example suites, are partially documented in Section 11.

- Suite graphing now has default start and end cycles, and you may define your own default start and end cycles for graphing in the suite.rc file.
- Default initial and final cycle times for running a suite can now optionally be defined in the suite.rc file, so you don't have to keep typing them on the command line, or GUI panel, for repetitive testing.

5.2 Configuring Your Environment For Cylc

To gain access to cylc you just need to source the cylc environment script. Put the following code in your login script, or do the same at the terminal prompt before using cylc.[9]

```
# .profile
export CYLC_DIR=/path/to/cylc/installation
. $CYLC_DIR/environment.sh
```

The variable `$CYLC_DIR` is required inside the environment script, so don't skip the export step. You should now be able to run cylc commands:

```
$ cylc --version
3.x.y
```

[9] To switch between different cylc installations just source the appropriate environment script.

You should also specify the graphical and terminal editors you want to use to edit suites, by setting the following environment variables in your `.profile`:

```
# .profile
export GEDITOR='gvim -f'  # GUI editor launched by gcylc
export EDITOR=vim         # terminal editor launched by 'cylc edit'
```

See `cylc edit help` (Appendix B.2.11) for other editor examples. And finally, you must ensure that `$TMPDIR` is defined in your environment, for example:

```
# .profile
export TMPDIR=/tmp/$USER
```

5.3 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, in a treeview organized by owner, group, and name (see Figure 13).

5.3.1 Central Suite Database Actions

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 definition in your editor,
4. plot the suite dependency graph,
5. search the suite definition and bin scripts,
6. validate the suite definition,
7. compare (difference) the suite with another suite,
8. import the suite or group to your private database,
9. unreregister or delete the suite or group (if you own it),
10. reregister the suite or group (if you own it).

But you can only *run* suites that are registered in your private database.

5.4 Importing The QuickStart Suites To Your Private Database

You can register new suites directly in your private database using gcylc or `cylc db register`, or you can import suites from the central database.

Find the suites registered under the *admin:examples* group in the central database (replace 'admin' with the name of the user account under which cylc was installed). If there are a lot of suites in the central database, use View → Filter to restrict the view (you can leave the dialog open and refilter as often as you like).

Now right click on the *examples* group and choose 'Import' to copy the whole group of suites to your private database. A dialog box will pop up allowing you to register your copy under a different group (leave this as it is) and a destination for the suite definition directories in the group (choose something like `$HOME/suites/examples`).

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 use the Database menu to switch back to your private database, and confirm that you now have a copy of the example suites.

5.4.1 By The Command Line

Here's how to do the same thing on the command line:

```
$ cylc db import admin:examples: $HOME/suites/examples
Matched:
   admin:examples:CUG1
   admin:examples:CUG73
   admin:examples:CUG74
   admin:examples:CUG_QS1
   admin:examples:CUG_QS2
   admin:examples:CUG_QS3
   admin:examples:FFHook
   admin:examples:FFTask
Locking database /home/oliverh/.cylc/LDB/db
REGISTERED examples:CUG1 "<SNIP> Example 1" ~oliverh/suites/examples/CUG1
Copying suite definition
REGISTERED examples:CUG73 "<SNIP> Graph Example" ~oliverh/suites/examples/CUG73
Copying suite definition
REGISTERED examples:CUG74 "A two-task test suite" ~oliverh/suites/examples/CUG74
Copying suite definition
REGISTERED examples:CUG_QS1 "<SNIP> Example 1" ~oliverh/suites/examples/CUG_QS1
Copying suite definition
REGISTERED examples:CUG_QS2 "<SNIP> Example 2" ~oliverh/suites/examples/CUG_QS2
Copying suite definition
REGISTERED examples:CUG_QS3 "<SNIP> Example 3" ~oliverh/suites/examples/CUG_QS3
Copying suite definition
REGISTERED examples:FFHook  "<SNIP> script example"  ~oliverh/suites/examples/FFHook
Copying suite definition
REGISTERED examples:FFTask  "<SNIP> task example"  ~oliverh/suites/examples/FFTask
Copying suite definition
Unlocking database /home/oliverh/.cylc/LDB/db
DONE
```

(suite title strings have been truncated again to reduce line length).

5.4.2 Private Suite Database Actions

By right-clicking on a suite in your private database, or using the cylc command line, 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 cylc stdout and stderr streams for the suite,
4. view the cylc log for the suite (which records all events and messages),
5. retrieve the suite description,
6. list tasks in the suite,
7. edit the suite definition in your editor,
8. plot the suite dependency graph,
9. search the suite definition and bin scripts,
10. validate the suite definition,
11. copy the suite or group,
12. compare (difference) the suite with another suite,
13. export the suite or group to the central database,
14. unreregister the suite or group,
15. reregister the suite or group.

5.5 Viewing The examples:CUG_QS1 Suite Definition

Right-click on the suite and choose 'Edit', or use the edit command,

```
$ cylc edit examples:CUG_QS1
```

to view the suite definition (suite.rc file) in your editor.

This moves to the suite definition directory (so that you can easily open other suite files in the same edit session) and opens the suite.rc file in your chosen editor. You can of course do this manually, but the automated way is quick and convenient, you don't have to remember suite definition directory locations, and for suite.rc files that contain include files you can optionally *edit the file inlined - it will be split back into its constituent include-files when you save the file and exit the editor.*

If you use the vim editor you can have nice cylc-specific syntax highlighting and section folding; see Section 7.2.3 for a screenshot and instructions.

You should see the following suite.rc file in your editor:

```
title = "Cylc User Guide Quick Start Example 1"
description = "(See the Cylc User Guide)"

runahead limit in hours = 12

[special tasks]
    startup         = 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"
```

This defines a complete, valid, runnable suite. To understand suite.rc files in detail, read *Suite Definition* (Section 7). Here's how to interpret this one:

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 happen to be satisfied before the previous instance is finished). Most tasks should be capable of this (see Section 12.5) but if necessary you can force particular tasks to run sequentially without using an explicit trigger on their own previous instance, e.g.:

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

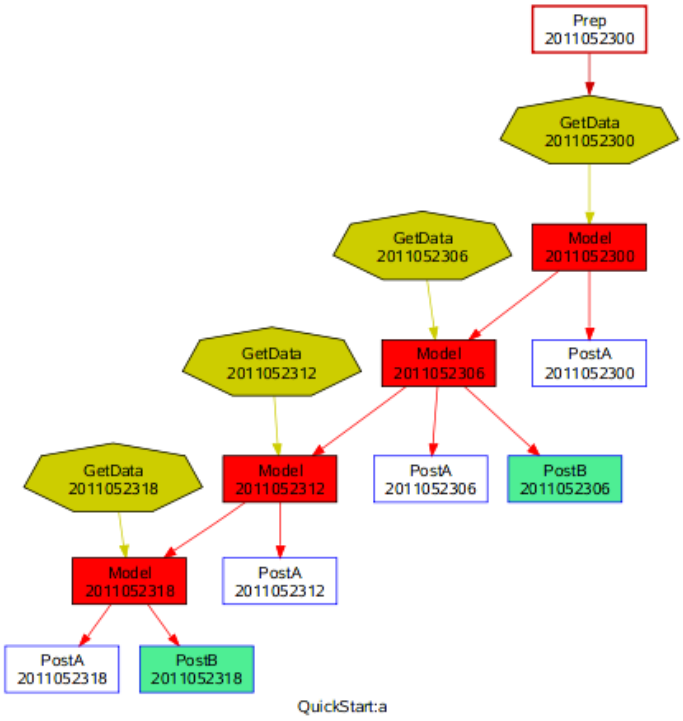

Figure 20: The *examples:CUG_QS1* dependency graph.

```
3     sequential = GetData
```

Finally, when the suite is first *cold started* from scratch it is made to wait on a special *startup* task called Prep. Startup tasks are one off tasks (non-spawning) that are only used at suite startup, and any dependence on a startup task in the suite graph only applies at suite startup.

The optional visualization section, fairly obviously, just defines properties used to plot the suite graph, which we'll do next.

5.6 Plotting The examples:CUG_QS1 Dependency Graph

Right-click on the *examples:CUG_QS1* suite in gcylc and choose Graph; or on the command line,

```
1 $ cylc graph examples:CUG_QS1 2011052300 2011052318 &
```

This should pop up a zoomable, pannable, graph viewer showing the graph of Figure 20.

If you edit the suite.rc file while viewing the graph, the viewer will update whenever you save the file.[10]

5.7 Running The examples:CUG_QS1 Suite

Each cylc task has command scripting that is called to invoke task processing when the task is ready to run. No task commands have been defined in our example suite so the tasks all default to the *dummy task* command line:

[10] Unless you're editing a temporarily inlined suite.rc file, in which case it will update when you save the file and exit the editor.

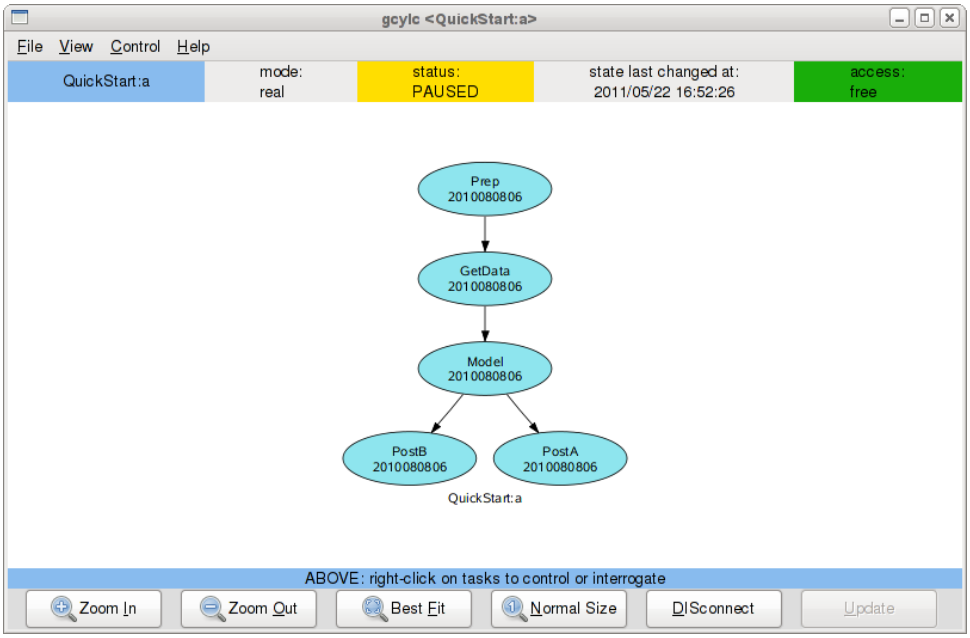

Figure 21: Suite *examples:CUG_QS1* at startup with an initial cycle time ending in 06 or 18 hours. Blue nodes indicate tasks in the waiting state.

```
# SUITE.RC
[tasks]
    [[GetData]]
        command = "echo DUMMY $TASK_ID; sleep $CYLC_SIMULATION_SLEEP"
```

where `$TASK_ID` is `NAME%YYYYMMDDHH`) and `$CYLC_SIMULATION_SLEEP` defaults to 10 seconds. This just writes an identifying message to stdout and then sleeps for a few seconds before exiting.

Now start a suite control GUI by right-clicking on the suite in gcylc and choosing 'Control (graph)'. You can also open a 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. *See* The Graph-Based Suite Control GUI*, Section D.*

In the control GUI click on Control → Run, enter an initial cold start cycle time (e.g. 2011052306), and select "Hold (pause) on startup" 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 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).

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 , and then the suite will stall, as shown in Figures 23 and 24.

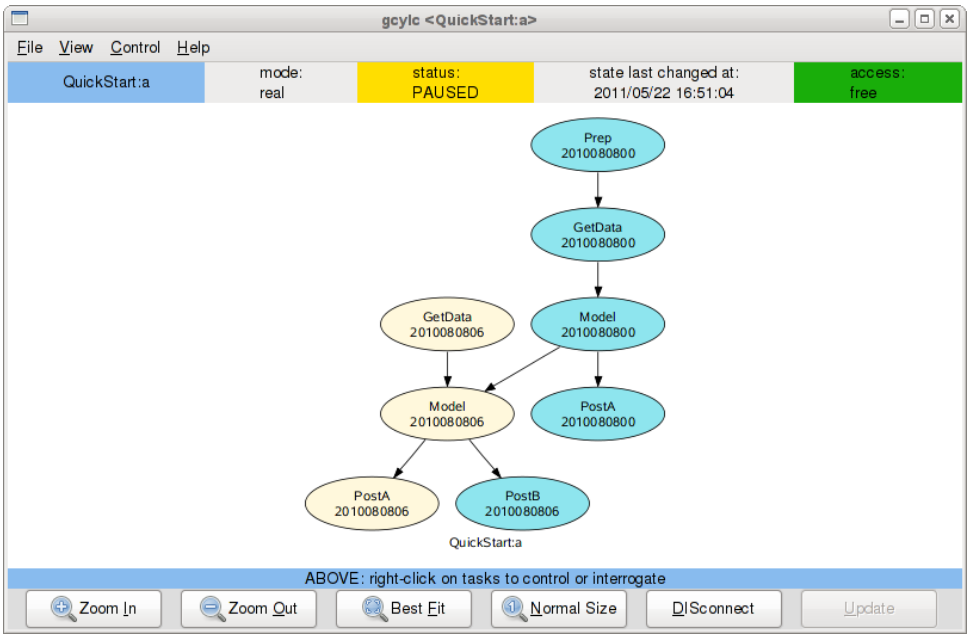

Figure 22: Suite *examples:CUG_QS1* at startup with an initial cycle time ending in 00 or 12 hours. Blue nodes represent tasks in the waiting state and off-white nodes are tasks from the base graph, defined in the suite.rc file, that aren't currently live in the suite.

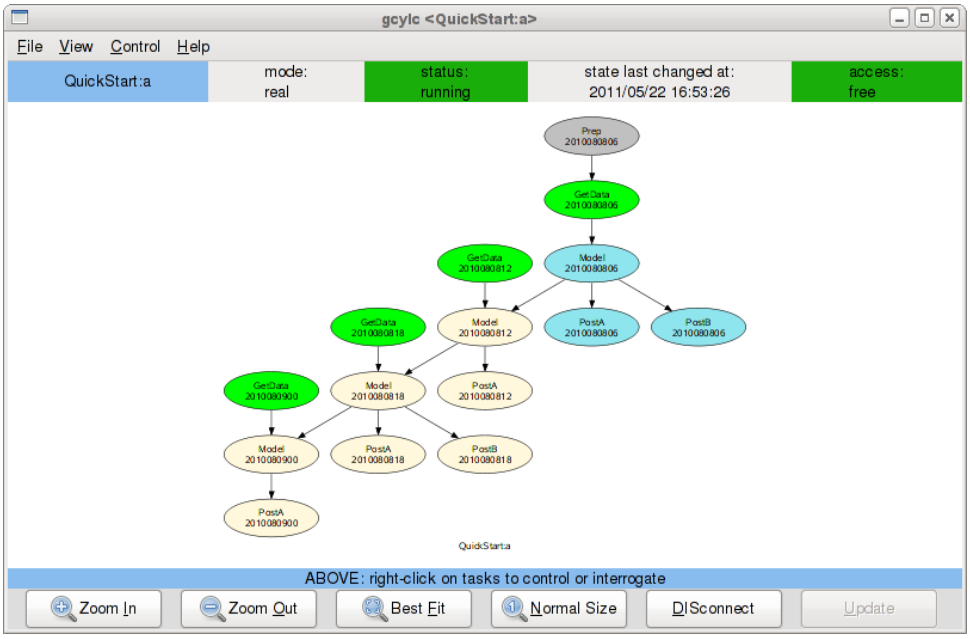

Figure 23: Suite *examples:CUG_QS1* running, showing multiple instances of the clock-triggered GetData task running at once.

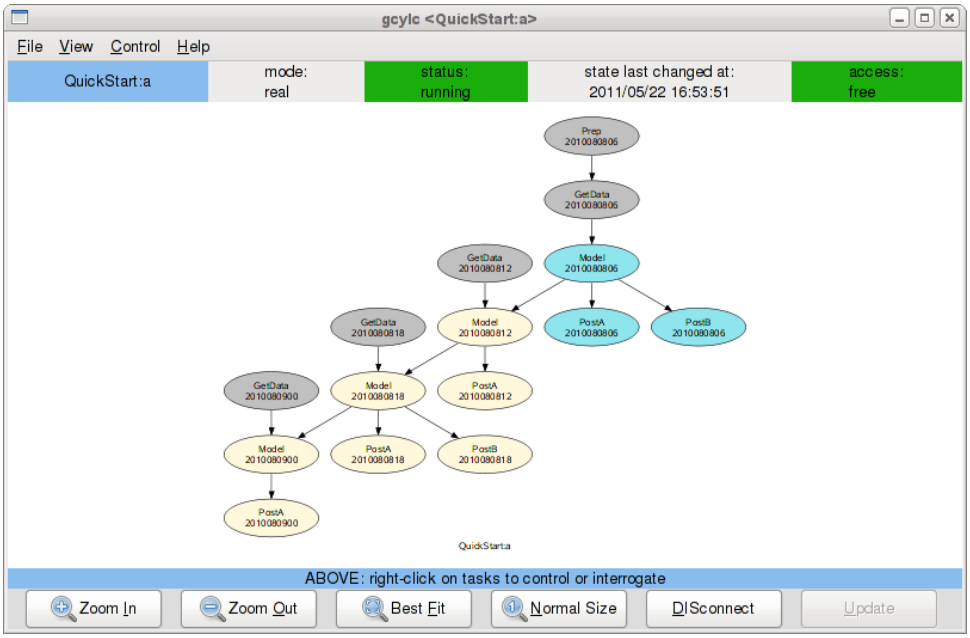

Figure 24: Suite *examples:CUG_QS1* stalled after the clock-triggered GetData tasks have finished, because of Model's previous-cycle dependence and the suite runahead limit (see main text).

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). They don't spawn beyond four cycles ahead though, thanks to the suite "runahead limit" which is set to 12 hours in the suite.rc file. The runahead limit is designed to stop free tasks like this from running off too far into the future. It is of no conseqence in normal real time operation[11] because the clock triggered tasks are then constrained by the wall clock, and the other tasks have to wait on them.

5.7.1 Viewing Task States

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, to get a small window like Figure 25.

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

5.7.2 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

[11] So long as it 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.

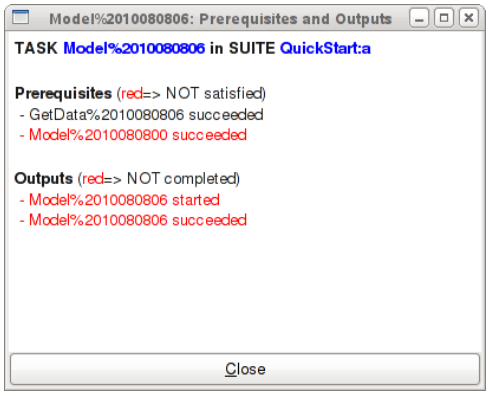

Figure 25: Viewing current task state after right-clicking on a task in gcylc. The same information is available from the `cylc show` command.

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.

5.7.3 Shutting Down And Restarting A Suite

Having watched the *examples:CUG_QS1* 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, in case you make a mistake.*

5.8 examples:CUG_QS2 - Handling Model Cold Starts Properly

Now take a look at *examples:CUG_QS2*, which is a minor modification of *examples:CUG_QS1*. Its suite.rc file has a new *cold start* task called ColdModel,

```
# SUITE.RC
[special tasks]
    cold start = ColdModel
```

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

```
# SUITE.RC
[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 can trigger off *either* its previous-cycle self *or* ColdModel in the same cycle.

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 startup, when there is no previous cycle. Unlike startup

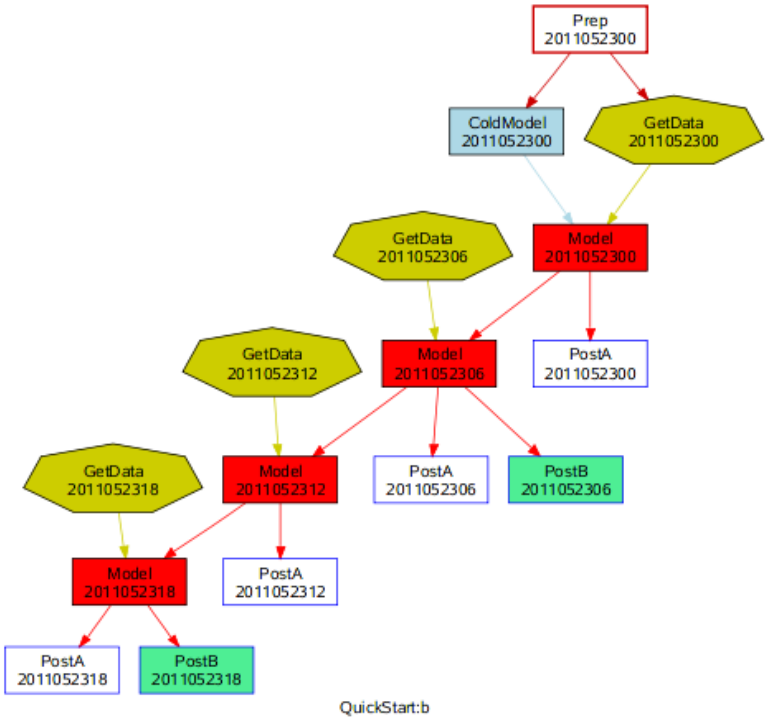

Figure 26: The *examples:CUG_QS2* dependency graph showing a model cold start task.

tasks though, cold start dependencies are not restricted to suite startup 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 often submits a real "cold start forecast" to generate the previous-cycle input files required by its associated model. Or it could just stand in for some external spinup process, or similar, that has to be completed before the suite starts - in this 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.

If you run *examples:CUG_QS2* you'll see that no manually triggering is required to get the suite started this time.

5.9 examples:CUG_QS3 - With Task Implementation

The suite *examples:CUG_QS3* is the same as *examples:CUG_QS2* except that it has real task implementations - scripts located in the suite bin directory that generate output files and consume input 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.

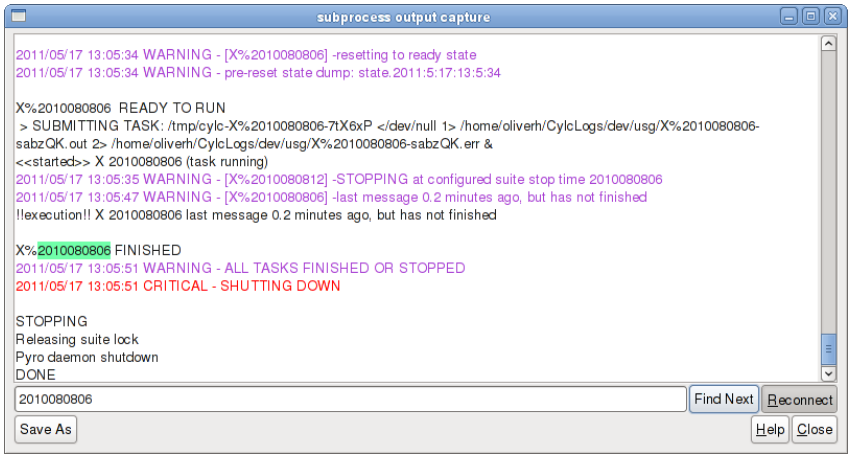

Figure 27: Cylc suite stdout/stderr example.

5.10 Following What's Happening In A Running Suite

5.10.1 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, what happens to this output is entirely up to you. 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.

5.10.2 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 5 most recent back ups are automatically 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.

5.10.3 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/CylcJobLogs/GROUP/NAME/`. 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 final location of the internal output of detaching tasks, whose initiating processes submit internal jobs before exiting early, is up to the task implementation.

5.11 Searching A Suite

The cylc suite search tool reports matches in the suite.rc file by line number, suite section, and file, even if nested include-files are used, and by file and line number for matches in the suite bin directory. The following output listing is from a search of the *examples:CUG3* suite, which has some task scripts in the suite bin directory.

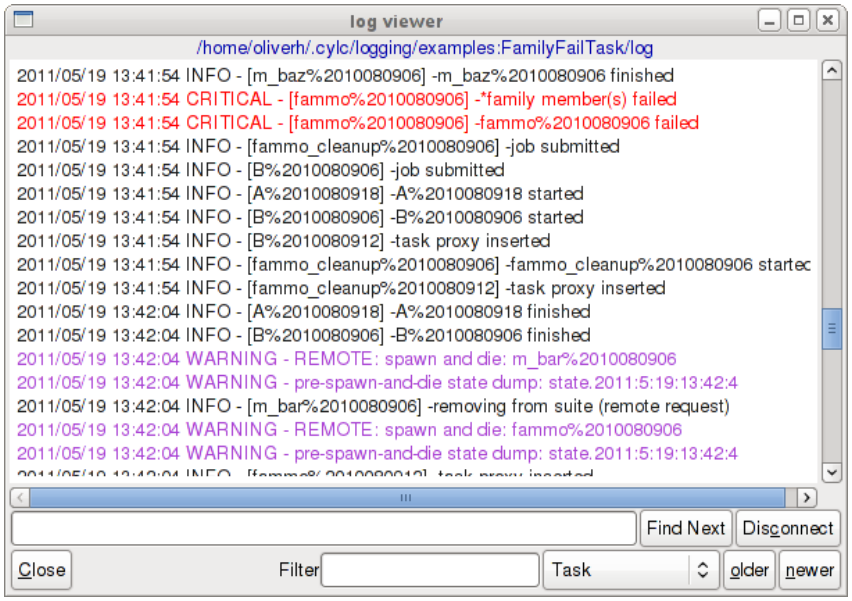

Figure 28: A cylc suite log viewed via gcylc.

```
$ cylc grep OUTPUT_DIR examples:CUG3

SUITE: examples:CUG_QS3
PATTERN: OUTPUT_DIR

FILE: /home/admin/cylc/examples/CUG5/three/suite.rc
   SECTION: [tasks]->[[GetData]]->[[[environment]]]
      [47]:             GETDATA_OUTPUT_DIR = $WORKSPACE
   SECTION: [tasks]->[[ColdModel]]->[[[environment]]]
      [54]:             MODEL_OUTPUT_DIR = $WORKSPACE
   SECTION: [tasks]->[[Model]]->[[[environment]]]
      [62]:             MODEL_OUTPUT_DIR = $WORKSPACE
   SECTION: [tasks]->[[PostA]]->[[[environment]]]
      [70]:             POSTA_OUTPUT_DIR = $WORKSPACE
   SECTION: [tasks]->[[PostB]]->[[[environment]]]
      [77]:             POSTB_OUTPUT_DIR = $WORKSPACE

FILE: /home/admin/cylc/examples/CUG5/three/bin/PostA.sh
   [7]: cylc checkvars -c POSTA_OUTPUT_DIR
   [21]: touch $POSTA_OUTPUT_DIR/surface-wind.products

FILE: /home/admin/cylc/examples/CUG5/three/bin/GetData.sh
   [6]: cylc checkvars -c GETDATA_OUTPUT_DIR
   [11]: touch $GETDATA_OUTPUT_DIR/obs-${CYCLE_TIME}.nc

FILE: /home/admin/cylc/examples/CUG5/three/bin/PostB.sh
   [7]: cylc checkvars -c POSTB_OUTPUT_DIR
   [21]: touch $POSTB_OUTPUT_DIR/precip.products

FILE: /home/admin/cylc/examples/CUG5/three/bin/Model.sh
   [11]: cylc checkvars -c MODEL_OUTPUT_DIR MODEL_RUNNING_DIR
   [54]: touch $MODEL_OUTPUT_DIR/surface-winds-${CYCLE_TIME}.nc
   [55]: touch $MODEL_OUTPUT_DIR/precipitation-${CYCLE_TIME}.nc
DONE
```

(The same thing can of course be done via the gcylc right-click menu).

5.12 Comparing Suites

Cylc can also compare suites and report differences by suite.rc section and item. For instance, comparing the example suites `CUG_QS1` and `CUG_QS2` by GUI or command line results in:

```
$ cylc diff examples:CUG_QS1 examples:CUG_QS2

Suite definitions examples:CUG_QS1 and examples:CUG_QS2 differ.

1 items only in examples:CUG_QS1 (<)

   [visualization] [[node attributes]]
 <   Model = ['style=filled', 'fillcolor=red']

3 items only in examples:CUG_QS2 (>)

   [visualization] [[node attributes]]
 >   models = ['style=filled', 'fillcolor=red']
 >   ColdModel = ['fillcolor=lightblue']

   [visualization] [[node groups]]
 >   models = ['ColdModel', 'Model']

7 common items differ examples:CUG_QS1(<) examples:CUG_QS2(>)
# (some have been omitted here for inclusion in the User Guide!)

   (top)
 <   suite log directory = /home/oliverh/CylcSuiteLogs/examples/CUG_QS1
 >   suite log directory = /home/oliverh/CylcSuiteLogs/examples/CUG_QS2

   [special tasks]
 <   cold start = []
 >   cold start = ['ColdModel']

   [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
DONE
```

(some output has been omitted for brevity). Note that the *suite log directory* items differ even though they are not explicitly defined in either suite. That is because their default values (see the *Suite.rc Reference*, Appendix A) are suite-specific.

5.13 Validating A Suite

After editing a suite always validate it to check for errors, via gcylc or the `cylc validate SUITE` command. Figure 29 shows validation of *examples:CUG73* (unfortunately the suite was registered under a different name when the screenshot was taken). Note the warnings that the tasks will be dummied out, as discussed above.

For more information on suite validation see Section 7.5.

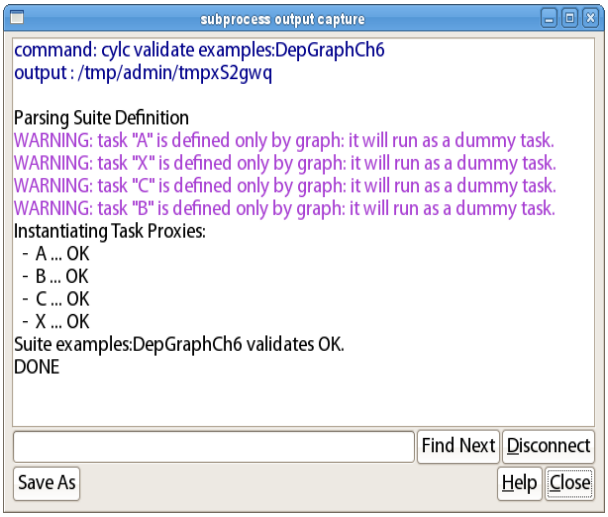

Figure 29: Suite examples:CUG73 Validation.

5.14 Using Example Suites To Understand Cylc

Cylc has been designed from the ground up to make testing and prototyping very easy. Simply drawing (in text) a dependency graph in a new suite.rc file creates a valid suite that you can run. The tasks will 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 (to fail, for example) by supplying appropriate task command lines to replace the default "dummy" one.

Cylc's example suites run quickly and are portable: you can copy and run them immediately without modification, even those with real task implementations that generate real (albeit zero-sized) output files. You can even run multiple copies of the same example suite at once (even multiple *instances* of the exact same suite, in fact), under different registrations, because the suite group and name are used in all I/O paths (without being hardwired anywhere).

You can copy an example suite in an instant, using the cylc commands or GUI, make a few changes according to your needs, and then run it to see what happens.

You can also run real suites in simulation mode, wherein each real 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 you can watch how any suite catches up and transitions from delayed to real time operation.

The example suites all use registration specific I/O paths so that you can run multiple instances of them at once without interference.

6 Suite Registration

How to construct a cylc suite is described in following sections of the User Guide. The cylc command set includes tools to aid in suite construction as well as suite control, however, and before you can do anything to a suite with cylc commands you must register it in your *private suite database*.

6.1 Private Suite Databases

Your private suite registration database simply maps suites, organised by GROUP and NAME, to their suite definition locations (see *Suite Definition*, Section 7). You can then refer to your suites concisely "by name" without having to specify, or even remember, where the suite definition resides. For example this command lists all the tasks in suite 'foo' of the 'oper' group.

```
1 $ cylc info list oper:foo
```

By right-clicking on a suite in the 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 cylc stdout and stderr streams for the suite,
4. view the cylc log for the suite (which records all events and messages),
5. retrieve the suite description,
6. list tasks in the suite,
7. edit the suite definition in your editor,
8. plot the suite dependency graph,
9. search the suite definition and bin scripts,
10. validate the suite definition,
11. copy the suite or group,
12. compare (difference) the suite with another suite,
13. export the suite or group to the central database,
14. unreregister the suite or group,
15. 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 without cluttering the public namespace with suites that are partially completed, broken, or not generally useful. It works similarly to a private suite database except that suites are organised by *owner* as well as group and name. Users can export suites to the central database to make them available to others. During export you can choose to have the suite definition copied to the database (the suite definition itself isn't "stored in the database" but the new registration will refer to the new centrally located suite definition) or you can have to central database reference the original suite.

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 definition in your editor,
4. plot the suite dependency graph,
5. search the suite definition and bin scripts,

6. validate the suite definition,
7. compare (difference) the suite with another suite,
8. import the suite or group to your private database,
9. unreregister or delete the suite or group (if you own it),
10. reregister the suite or group (if you own it).

6.2.1 Can Suites Be Shared Across The Network?

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 public suite database so that users can easily share suites across the network. The required server functionality is essentially identical to that of the cylc lockserver (see Appendix C) 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:
  gcylc ............... The cylc Graphical User Interface
  register ............ Add a suite to your private registration database
  reregister .......... Change private or central suite registrations
  unregister|delete ... Remove suites from the private or central database
  copy|cp ............. Copy a suite or group of suites.
  print ............... Print private or central suite registrations
  get-dir ............. Print a suite definition directory path
  export .............. Export private registrations to the central database
  import .............. Import central registrations to your private database
  refresh|check ....... Check for invalid registrations and update titles
```

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

7 Suite Definition

A cylc suite is defined entirely by a single structured, validated, *suite.rc file* 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 8.

7.1 The Suite Definition Directory

A cylc *suite definition directory* contains:

- **a suite.rc file**: this is *the* suite definition
- any include-files used by the suite.rc file (see Appendix A.1)
- **a bin directory** for scripts and programs that implement, or are used by, suite tasks
  - technically optional - tasks can call external commands, scripts, or programs
  - tasks get automatic access to their own suite bin directory
- any other suite-related documentation, control files, etc.
  - whole suite definition directories are copied if you copy a suite
  - files in the suite definition directory can be accessed portably by tasks at run time through the environment variable `$CYLC_SUITE_DIR` supplied by the suite (see *Task Execution Environment* (Section 7.4.3)
  - holding everything in one place makes revision control easy

Here's an imaginary example,

```
/path/to/my/suite   # suite definition directory
    suite.rc           # <-- SUITE DEFINITION FILE
    bin/               # bin directory (scripts, programs)
        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
    ...
```

7.2 The suite.rc File

Cylc suite.rc files parse directly into a nested data structure inside cylc, which makes it very easy to add and use new configuration items.

Suite.rc files are validated against a specification file (`$CYLC_DIR/conf/suiterc.spec`) that defines all legal entries, values, options, and defaults; these are documented in detail in the *Suite.rc Reference* (Appendix A).

7.2.1 Syntax

- **Entries** take the form `item = value`.
- **Comments** follow a hash character (#) to the end of the line.
- **Single Line Strings** are quoted, but quotes can be omitted if the string contains no commas (which indicate a list value).
- **Multiline Strings** are triple-quoted.
- **List Values** are comma separated.
- **White Space** is ignored but you can indent for clarity.
- **Continuation Lines** follow a trailing backslash.

- **Sections**
  - nesting is determined by the number of square brackets around the section heading.
  - sections are closed by the next section heading, so within a section all top level items must be defined before any subsections (and similarly for lower levels of nesting).
- **Include-files** can be multiply-included and nested. Paths are specified, portably, relative to the suite definition directory. Inclusions can span section boundaries.
- Duplicate items are allowed in environment and directives sections (this is good for overriding defaults held in an include-file).

The following pseudo-listing illustrates the file structure:

```
# comment
item = value # trailing comment
a string item = the quick brown fox
another string item = "the quick brown fox"
yet another string item = """the quick brown fox
jumped over the lazy dog"""
a list item = foo, bar, baz
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]]
        # ...
    # ...
# ...
```

7.2.2 An Example

A typical suite.rc file might contain the following information:

- suite title
- suite description
- a default job submission method for the suite
- lists of tasks with special behaviour (e.g. clock-triggered tasks)
- a dependency graph defining the relationships between tasks
- a global environment section (variables available to all tasks)
- and for each task,
  - task description
  - task-specific environment section (variables available to just this task)
  - the command scripting to execute when the task is ready to run
- optionally, a visualization section to configure graph plotting for the suite

Here's an example:

```
# GLOBAL SETTINGS
title = suite foo
```

```
description = """Run Model on real time input data and postprocess its
output with PostA, four times daily; twice daily do additional
postprocessing with PostB."""

job submission method = loadleveler

[special tasks]
    # TASKS WITH UNUSUAL BEHAVIOUR
    cold start       = ColdModel
    clock-triggered = GetData(1)

[dependencies]
    # THE SUITE DEPENDENCY GRAPH
    [[ 0,6,12,18 ]]  # four times daily
        graph  =  """
            GetData => Model => PostA
            ColdModel | Model(T-6) => ModelA  # model cold start or restart
                  """
    [[ 6,18 ]]  # additional postprocessing at 6 and 18 UTC
        graph = "Model => PostB"

[environment]
    # ENVIRONMENT VARIABLES AVAILABLE TO ALL TASKS
    WORKSPACE = /$TMPDIR/$CYLC_SUITE_GROUP/$CYLC_SUITE_NAME

[tasks]
    # COMMAND AND EXECUTION ENVIRONMENT FOR EACH TASK
    [[GetData]]
        description = "retrieve data for the current cycle time"
        command = GetData.sh
        [[[environment]]]
            # ENVIRONMENT VARIABLES AVAILABLE TO THIS TASK
            GETDATA_OUTPUT_DIR = $WORKSPACE
    [[Model]]
        # ...
    [[PostA]]
        # ...

[visualization]
    # NODE AND EDGE PROPERTIES FOR DEPENDENCY GRAPH PLOTTING
    # ...
```

This is in fact the *examples:CUG_QS3* suite from the Quick Start Guide, which you can copy, study, and run. Dependency graphs plotted from this (by gcylc or `cylc graph`) are shown in Figures 30 and 31.

Subsequent sections of the User Guide explain important parts of the suite.rc file in detail. To see what else can go in a suite definition, study other example suites and refer to the *Suite.rc Reference* (Appendix A).

Figure 30: Dependency graphs (cold and warm start) plotted from a simple suite.rc file.

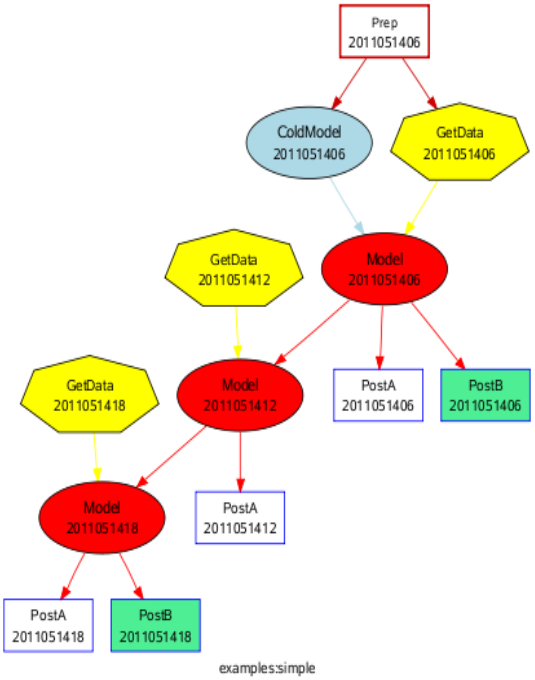

Figure 31: Three cycle cold start dependency graph plotted by 'cylc graph' from the simple example suite.rc file, showing variation between cycles.

```
# SUITE.RC
title = "Dependency Graph Example (Ch.6)"
[dependencies]
    [[0,6,12,18]] # 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"
```

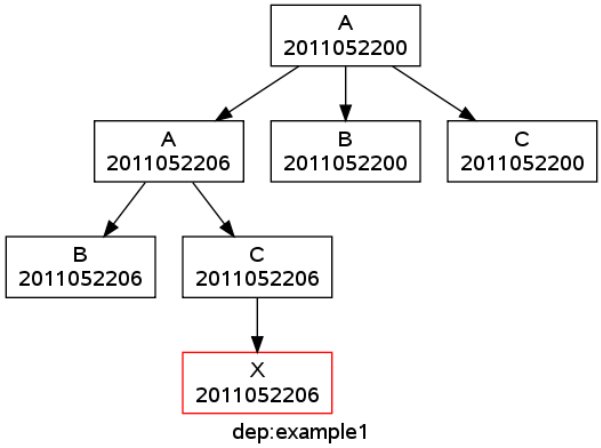

Figure 32: Dependency Graph Example 1.

7.2.3 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.

7.3 Dependency Graphs

Dependency graphs define the relationships between tasks in a suite. The cylc suite.rc graph notation makes clear textual representations of actual graphs, but it is 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.

Figure 32 shows an example alongside its graphical representation, plotted with,

```
$ cylc graph examples:CUG73 2011052200 2011052206
# (or use right-click Graph in gcylc)
```

Incidentally, the suite.rc file listed in Figure 32 is a complete valid suite definition that you can run (just import a copy from the central suite database and run it): cylc will run dummy tasks, which sleep for a few seconds then emit an identifying message, because task command scripting has not been specified. If you do try to run Example 1, be aware that you'll need to trigger task A manually to get the suite started. That's because A(T) depends on its own previous instance A(T-6), and at startup there is no previous cycle to satisfy that prerequisite. How to handle a *cold start* properly is described below in *Satisfying Intercycle Dependencies At Startup* (Section 7.3.9) and in the *Quick Start Guide* (Section 5).

7.3.1 Graph Syntax

Multiline graph strings may contain:

- **blank lines**
- **arbitrary white space**
- **task names**
- **arrows** (`=>`) for dependencies (triggers)
- **comments** - following the # character

- **conditional operators**: `&` (AND) and `|` (OR)
- **explicit task output labels** - `foo:fail`, `foo:out1`
- **cycle time offsets** - `A(T-6)`

7.3.2 Valid Cycle Times For Each Task

The set of hours at which a task can run is defined by the hour list headings of any graph sections that the task appears in. This is how cylc knows what "the next cycle time" for a particular task is, when it comes time to spawn a successor.

7.3.3 Partitioning The Graph

Suite dependency graphs can be broken down into *dependent pairs* where the left side of pair (which may be a single task, or several that are conditionally related) defines a trigger for the task on the right, for cycle times matching the list of hours specified for the graph section.

For instance, the simple "word graph" *C(T) triggers off B(T) which triggers off A(T)*, where T represents task cycle time, can be deconstructed into the pairs *C(T) triggers of B(T)* and *B(T) triggers off A(T)*. Also, T should be left implicit except when the left side has an offset, i.e:

- `A(T)` must be written as just `A`

This make graphs look clean and concise, because the vast majority of tasks in a typical suite will only depend on others with the same cycle time.

Also, because dependent pairs define a trigger for the right side task at cycle time T, where T matches one of the hours listed for the graph section,

- cycle time offsets such as `A(T-6)` can only appear on the left side of a pair.

Now, having explained that dependency graphs are interpreted in a pairwise manner, you can optionally, but intuitively, 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 (and can be written like) this:

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

Note that because cycle time offsets can only appear on the left of an arrow, any such offset task can only be leftmost in the chain, 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!
```


(Of course `A => B(T-6)` would not make sense in a real suite in any case - if this kind of relationship seems necessary, it probably means that B should be "reassigned" to the next cycle - remember that except for the actual forecast models in a suite, 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 (and can be written like) this:

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

The branching tree structure of a dependency graph can now 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
               B => C
               B => D"""
# and it can even be written like this:
    graph = """A => B # blank line follows:

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

7.3.4 Task Succeeded Triggers

This is the default behaviour - to trigger off another that succeeds (i.e. *finishes successfully*):

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

7.3.5 Task Failed Triggers

To get a task to trigger off another's failure:

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

This can be used for certain kinds of failure recovery. If you don't care whether a task succeeds or fails so long as it finishes,

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

7.3.6 Internal Triggers

These are only needed if you want a task to trigger early off an output that the upstream task completes before it finishes.

```
# SUITE.RC
[dependencies]
    [[6,18]]
        # B triggers off specific OUTPUT "out1" of task A:
        graph = A:out1 => B
[tasks]
    [[A]]
        [[[outputs]]]
            out1 = "NWP products uploaded for $(CYCLE_TIME)"
```

Task A must emit this message when the actual output has been completed - see *Reporting Internal Outputs Completed*, Section 8.4.2.

7.3.7 Intercycle Triggers

Most tasks in a typical suite will trigger off other cotemporal (same cycle time) tasks, 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 more general intercycle dependence is not uncommon.[12] 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"
```

7.3.8 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
```

[12] 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.

```
            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 true "conditional triggers", can only appear on the left,[13]

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

7.3.8.1 Which Task Will Be Plotted?

In a list of alternative triggers separated by OR operators, the full conditional expression determines run time behaviour, but one of the alternatives has to be singled out in order to plot the graph: the rightmost task is plotted by default; to change this change the task order or attach an asterisk to the chosen task:

```
# SUITE.RC
# this will plot as C => D:
    graph = "A | B | C => D"
# but this will plot as A => D:
    graph = "A* | B | C => D"
```

In future cylc releases we may allow all tasks in a conditional trigger to be plotted, with dotted arrows, say, to indicate the conditional nature of the relationship.

7.3.8.2 Complex Conditional Triggers With AND And OR

You can use AND and OR operators in the same expression,

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

KNOWN BUG ALERT: complex conditional triggers work as expected in cylc suites at run time, but they are currently not graphed properly.

7.3.8.3 Parenthesized Complex Conditional Triggers

Parenthesized conditional expressions are currently illegal. Internally cylc can handle this perfectly well, because it transforms prerequisite conditions directly into Python code and gets the interpreter to evaluate the result, *but* the cylc-3 suite.rc graph parser currently cannot handle general conditional expressions.

This restriction should not be an impediment because forecasting suites tend to have simple requirements for conditional triggers (plus you can rely on operator precedence - AND before OR - and split large expressions into multiple lines). However, if there is any call for more complicated conditional triggers we will certainly address this problem.

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

7.3.9 Satisfying Intercycle Dependencies At Startup

Considering the intercycle dependence shown in *Intercycle Triggers* (Section 7.3.7) it is clear that some kind of bootsrapping process will be required to get the suite going initially, because in the very first cycle there is no previous instance of A to satisfy B's prerequisites.

7.3.9.1 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 its counterpart's previous-cycle trigger.

A cold start task may invoke real processing (to generate the files that are normally generated by the cycling task that it masquerades as) or it could be a dummy task that represents some external spinup process (resulting in the same files) that has to be completed before the suite is started (in this case the cold start task in cylc will just report itself successfully completed, thereby satisfying the aforementioned dependencies).

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

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

In other words, Foo(T) can trigger off *either* Bar(T-6) *or* ColdFoo(T). At startup ColdFoo will do the job, before being be eliminated from the suite (because cold start tasks are non-spawning), and thereafter Bar(T-6) will do it.

A cold task can also be inserted into the suite at run time to cold start just the task that it is associated with, if a problem of some kind prevents continued normal cycling.

7.3.9.2 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.

Warm starting a previously shut down suite at a particular cycle time is an alternative to *restarting* it from a previous state (which in cylc is likely to include tasks with several different cycle times). Warm starts assume the existence of a previous cycle (i.e. that any files from the previous cycle that are required by the new cycle will be in place already). So no cold start tasks need to run *but* cylc itself doesn't know the details of the previous cycle (it would if you did a restart from the previous state rather than 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 already in the succeeded state. In other words, the cold start tasks stand for the previous finished cycle, rather than actually running processes that masquerade as the previous cycle.

7.3.10 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
[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 its cold start task, *or* off its own previous instance, *or* 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'.*[14] 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 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 in 'models with explicit restart outputs', instead of 'sequential' tasks, and you use explicit labeled restart outputs *containing 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 the situation, of course.

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

[14] 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.


```
            r18 = restart files completed for $(CYCLE_TIME+18)
```

7.3.11 Task Families

A task family is a named group of tasks that appears as a single task in the suite dependency graph. Instead of having each individual member task trigger off one or more upstream tasks, you can just trigger the family; and instead of triggering downstream processing off each member task, you can trigger off the family as a whole.

- a family enters the 'running' state when its prerequisites are satisfied
  - this causes each member of the family to trigger
- a family's final state is not determined until all of its members have succeeded or failed:
  - succeeded, if all members succeeded
  - failed, if one or more members failed

Cylc task families are implemented as special pseudo-tasks that do not actually submit anything to run when they enter the 'running' state.

Cylc task families can have internal dependencies, and members can also participate in dependent relationships outside of the family.

Task families can be declared 'sequential' in the suite.rc [special tasks] section, in which case they will run sequentially even if their members are not sequential.

To trigger downstream processing off a family even if one or more of its members failed (e.g. a family of obs processing tasks wherein you want to use the obs that were successfully processed even if some members fail) use a conditional trigger on the family:

```
# SUITE.RC
[dependencies]
    [[ 0,6,12,18 ]]
        graph  =  "Family | Family:fail => PostProc"
```

Task families are listed in the top level of the suite.rc file, and then used in the suite graph just like ordinary tasks.

```
# SUITE.RC
[task families]
    Family = one, two, three, four
[dependencies]
    [[ 0,6,12,18 ]]
        graph  =  "PreProc => Family => PostProc"
[visualization]
    show family members = True
    default node attributes = "shape=box", "color=black"
```

This simple suite is plotted, with and without family members, in Figure 33.

7.3.11.1 Recovering From Task Family Failures

If a family member fails it will enter the 'failed' state, and once all other members have finished (succeeded or failed), the family itself will enter the failed state. To recover from this, assuming the problem that caused the original member to fail is fixable,

1. fix the problem that caused the failure
2. reset the failed member task to the 'waiting' state
3. retrigger the family itself

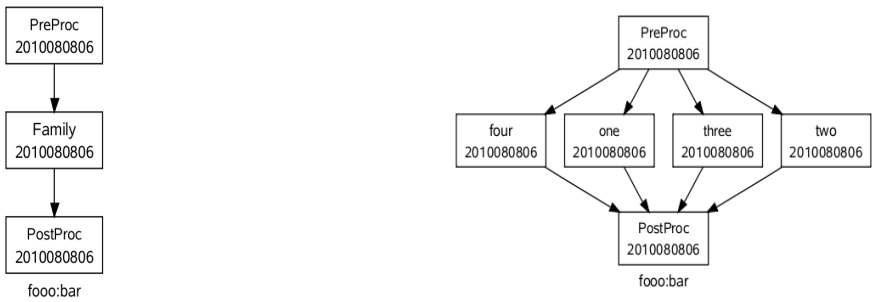

Figure 33: A very small suite with a simple task family, showing the family itself on the left, and the effective dependencies on the right (plotted with `SUITE.RC: [visualization] -> show family members = True`).

When the family triggers, the reset member task will follow suit. If the member completes successfully this time, so will the family, and downstream processing will proceed as normal.

7.3.11.2 Use Task Families Sparingly

Task families should really only be used as a convenient simplification for groups of tasks that would naturally trigger at the same time anyway (e.g. 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 use of task families may unnecessarily constrain cylc's ability to achieve maximum functional parallelism, because every member has to wait on its parent family starting before it can run, and all members have to finish before downstream processing can continue.[15]

7.4 Task Definition

A task's position in the suite dependency graph determines its relationship to other tasks and its valid cycle times. Other properties such as execution environment and the command scripting used to invoke task processing are defined under the suite.rc `[tasks]` section.

Note that there is not necessarily a one-to-one mapping between tasks defined in the suite.rc file and the external commands, scripts, or programs that implement them. Multiple tasks can call the same script (etc.) with varying inputs supplied by their respective command lines and execution environments.

7.4.1 A Task Definition Example

The following rather trivial suite of two tasks illustrates the important aspects of task definition:

```
# SUITE.RC
title = "A two task test suite"
description = "Used as an example in Section 7.4 of the User Guide"
[dependencies]
    [[0,12]]              # For cycle times ending in 00 and 12 hours
        graph = "A => B"  # B triggers when A successfully finishes
[environment]             # GLOBAL ENVIRONMENT
    NEXT_CYCLE = $( cylc util cycletime -a 12 )
    PREV_CYCLE = $( cylc util cycletime -p 12 )
```

[15] Actually downstream tasks can trigger off individual family members as if they weren't family members - but extensive use of this would probably defeat the purpose of using a family in the first place.

```
[tasks]
    [[A]]                  # TASK A PROPERTIES
        description = "Task A, does something"
        command = A.sh
        [[[environment]]] # Task A environment
            INPUT = foo-${NEXT_CYCLE}.txt
    [[B]]                  # TASK B PROPERTIES
        description = "Task B, does something else"
        command = B.sh -f
        [[[environment]]] # Task B environment
            INPUT = bar-${NEXT_CYCLE}.txt
```

Note the task names in the suite dependency graph, the command scripting for each task, and the global and task-specific environment sections.

7.4.2 Task Names

Task names can contain letters, digits, and underscores. Task names should not be hard wired into task implementations (scripts, models, etc.) because task and suite identity can be extracted portably from the task execution environment supplied by the suite (Section 7.4.3) - then to rename a task, just change the name in the suite.rc file.

7.4.3 Task Execution Environment

The task execution environment comprises variables automatically defined by cylc, and global and task-specific variables defined in the suite.rc file. These variables are explicitly exported prior to executing the task command scripting (to see how this is implemented go to Section 9, *Task Execution*).

7.4.3.1 Suite And Task Identify

The task's name and cycle time are defined before the global variables, so global variables can actually be made task- and cycle-specific by referencing them.

An executing task will almost certainly need to use its own cycle time:

```
$CYCLE_TIME
```

It may also need the location of its suite definition directory, to access files stored there,

```
$CYCLE_SUITE_DIR
```

And it may need to use the task and suite identity variables (e.g. to make portable suites that write all output to suite-specific locations):

```
$CYLC_SUITE             # e.g. foo:bar
$CYLC_SUITE_GROUP       # foo
$CYLC_SUITE_NAME        # bar
$TASK_NAME              # X
$TASK_ID                # X%2011051118
```

These variables are also used automatically by the cylc task messaging commands, to target the right task proxy object in the right suite.

7.4.3.2 Task Access To Cylc

Task execution environments are automatically configured to give the executing task access to cylc commands and utilities. This is done prior to exporting user-defined variables so that cylc utility commands can be used in variable assignment expressions. For example:

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

7.4.3.3 Global And Task-Specific Environment Variables

Generally speaking, parameters common to several tasks should be defined once in the global environment section of the suite.rc file; and those needed by a single task, in the task-specific environment section. Task-specific variables can reference global variables.

Order of definition is preserved for user-defined global and task-specific environment variables, so you can safely refer to previously defined variables.

7.4.3.4 When Are Environment Variables Evaluated?

Environment variables are *not* evaluated by cylc prior to executing the task. They are written, in unevaluated form, to the job script that is submitted by cylc to run the task (this is shown in Section 9.1). Thus `$HOME`, for instance, will evaluate, at run time, to the home directory of the account under which the task executes (which isn't necessarily the same account, or even the same host, that is running the suite).

7.5 Suite Validation

Suite validation is designed to catch most suite definition errors before run time. First the suite.rc file is validated against the spec file `$CYLC_DIR/conf/suiterc.spec`, which is the ultimate arbiter of what's legal in a cylc suite definition (see the *Suite.rc Reference*, Appendix A). Validation detects any formatting errors, typographic errors, illegal items, and illegal values; then some global consistency checking is done; and finally the validator attempts to create each suite task proxy according to information parsed from the suite.rc file.

Here's an example of a successful validation,

```
$ cylc validate examples:simple
Parsing Suite Config File
Instantiating Task Proxies:
  - ColdModel ... OK
  - GetData ... OK
  - Model ... OK
  - PostA ... OK
  - PostB ... OK
  - Prep ... OK
Suite examples:simple validates OK.
DONE
```

7.5.1 Validation Errors

The validator reports the line numbers of detected errors.

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

If the suite.rc file contains include-files, use the `cylc inline SUITE` command (or the gcylc right-click 'Edit' option) to view an inlined copy with correct line numbers.

7.5.2 Validation Warnings

Warnings are emitted by the validator if any tasks in the dependency graph are not defined in the [tasks] section, or if any tasks defined in the [tasks] section are not used in the dependency graph:

```
Parsing Suite Config File
WARNING: task "PostA" is defined only by graph: it will run as a dummy task.
WARNING: task "Prep" is defined in [tasks] but not used in the graph.
Instantiating Task Proxies:
  - ColdModel ... OK
  - GetData ... OK
  - Model ... OK
  - PostA ... OK
  - PostB ... OK
  - Prep ...WARNING: no hours in graph or [tasks][[Prep]]; task can be
                     'submit'ed but not inserted into the suite.
 OK
Suite examples:simple validates OK.
DONE
```

These are not necessarily errors - a task defined by graph alone will run as a dummy task, and you can temporarily disable a defined task by commenting it out of the dependency graph.

8 Task Implementation

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

8.1 Most Tasks Require No Modification For Cylc

Any existing command, script, or executable can function as a cylc task if the following conditions are met:

- The task must have no *internal outputs* that others trigger off - otherwise it must be modified to report when those internal outputs are completed.
- The script or process that invokes the task must not exit before all task processing is finished - otherwise it must be modified to report final success or failure itself.
- Task scripting must return non-zero exit status on error, to allow automatic detection of successful completion or failure.

If these requirements are not met see *Modifying Scripts For Cylc*, Section 8.4. Reporting internal outputs is easy. Manual completion messaging is more involved, but is typically only needed for large models with internally complex native job submission processes.

The following suite.rc listing shows several tasks that run external scripts, `foo.sh` and `bar.sh`, which do not know about cylc:

```
# SUITE.RC
[tasks]
    [[foo]]
        description = a task that runs foo.sh
        command = foo.sh OPTIONS ARGUMENTS

    [[bar]]
        description = a task that runs bar.sh
        command = """echo HELLO
                     bar.sh
```

```
                    echo BYE"""

    [[baz]]
        description = "a task that runs baz.sh, and retries on failure"
        command = """echo attempt No.1
                    baz.sh""",
                  """echo attempt No.2  # only invoked if try No.1 fails
                    baz.sh --retry"""
```

8.2 Suite.rc Inlined Tasks

Simple tasks can be entirely implemented within the suite.rc file because the task *command* string can contain multiple lines of scripting (or a list of such strings, to retry on failure). The command string is written verbatim to the task job script after configuring the execution environment - see *Task Execution*, Section 9.

8.3 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 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
```

8.4 Modifying Scripts For Cylc

8.4.1 Voluntary Messaging

If you like you can modify task scripts to log explanatory messages with the suite if certain events occur. 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 can be examined for evidence of what went wrong.

8.4.2 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 $CYCLE_TIME"
```

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

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

otherwise it will just be logged as a progress report or similar.

8.4.3 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 9.1) that otherwise provides automatic completion messaging cannot know when the task is really finished. To disable automatic completion messaging:

```
# SUITE.RC
manual task completion messaging = True  # global setting
[tasks]
    [ [foo]]
        manual task completion messaging = True  # 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, as explained in *Task Execution Environment* (Section 7.4.3), 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.[16]

8.4.4 A Custom Task Wrapper Example

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

```
#!/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.
```

[16] 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.

```
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."""

[special tasks]
    sequential = model

[dependencies]
    [[0,6,12,18]]
        graph = "model"

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

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:detach.
# See documentation in the Cylc User Guide, Section 8.4.3.

# 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}@${CYLC_SUITE_ENVIRONMENT}\n${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 ${PREFIX}-${CYCLE_TIME}-$$.out ${PREFIX}-${CYCLE_TIME}-$$.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.

8.5 Running Local Tasks Under Other User Accounts

If a task declares an owner other than the suite owner, and does not declare a remote host:

```
# SUITE.RC
owned task execution method = sudo  # or ssh
[tasks]
    [[foo]]
        owner = bob
        job submission method = loadleveler
```

cylc will attempt to execute the task, by the configured *job submission method* (e.g. loadleveler), as the specified owner, using the configured *owned task execution method*. To use sudo with llsubmit (loadleveler), for example, `/etc/sudoers` must be configured to allow the suite owner to execute the llsubmit command as the task owner. For ssh, passwordless ssh must be configured between the suite owner and task owner accounts.

A task owner can also be declared globally, if all or most of your tasks need to run under a different username than the suite owner.

8.6 Running Tasks On A Remote Host

You should 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.

If a task declares a remote host and owner in its suite.rc task section, cylc will copy the task job script to the remote host by `scp`, and then run it on the remote host (using the right job submission method for the task) by `ssh`.

Remote host, task owner, and directories can be declared globally and/or by task. Use the global settings if all or most of your tasks need to run on the same remote host.

```
# SUITE.RC
```

```
# GLOBAL remote host settings
owner = alice
remote host = foo.niwa.co.nz
remote cylc directory = /path/to/remote/cylc/installation/on/foo
remote suite directory = /path/to/remote/suite/definition/on/foo
job submission method = loadleveler
[tasks]
    [[TaskA]]
        # (no task specific remote host settings: this task runs on foo)
        # ...
    [[TaskB]]
        # TASK remote host settings:
        owner = bob
        remote host = bar.niwa.co.nz  # but this one runs on bar
        remote cylc directory = /path/to/remote/cylc/installation/on/bar
        remote suite directory = /path/to/remote/suite/definition/on/bar
        job submission method = at_now
```

Specifying the remote cylc directory gives remote tasks access to cylc commands; and the remote suite directory gives them access to suite files via `$CYLC_SUITE_DIR`, and to the suite bin directory via `$PATH` (automatically configured by the cylc environment script).

- 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 its parent suite.
- 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.

Note that you can easily run the cylc example suites on a remote host by simply setting the global remote hosting suite.rc configuration items (if cylc is installed on the remote host). For the example suites with real task implementations that work with "real" files, you'll have to run *all* tasks on the same remote host so that they can all access their required 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.

8.6.1 Remote Task 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 9.3.

Remote task stdout and stderr, however, currently ends up in the remote task owner's home directory, ignoring any suite.rc job log directory settings. This is simply because at cylc's home institution we now have cross platform job submission via loadleveler so we don't need cylc's remote task functionality . . . and consequently cylc needs updating slightly in this respect.

9 Task Execution

Task Implementation (Section 8) describes what requirements a command, script, or program, must fulfill in order to function as a cylc task.

This section explains how tasks are executed when they are ready to run, and how to define new task job submission methods.

9.1 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 then submitted to run by means of the *job submission method* specified for the task. Different tasks can have different job submission methods. You can set a default for the suite and override it, if necessary, per task:

```
# SUITE.RC
job submission method = loadleveler  # suite default
[tasks]
   [[foo]]
        job submission method = at_now  # just for task foo
```

The actual command line used to submit the job script is written to suite stdout by cylc. You can see this using `cylc submit`, which runs a single task exactly as the suite would. The `--dry-run` option causes it to generate the job script for inspection and show how it would be executed with the chosen job submission method. The following listing illustrates using the two task example suite of Section 7.4.1 (also registered as *examples:CUG74* in the central suite database):

```
% cylc submit --dry-run examples:CUG74 A%2011052400
DRY RUN: create the task job script and show how it would be executed.
 > TASK JOB SCRIPT: /tmp/cylc-A%2011052400-jfHtqy
 > JOB SUBMISSION: /tmp/cylc-A%2011052400-jfHtqy </dev/null 1>
     /home/oliverh/CylcJobLogs/examples/CUG74/A%2011052400-DuaPUt.out 2>
     /home/oliverh/CylcJobLogs/examples/CUG74/A%2011052400-DuaPUt.err &
```

Here is the generated job script:

```
% cat /tmp/cylc-A%2011052400

#!/bin/bash

# ++++ THIS IS A CYLC TASK JOB SCRIPT ++++
# Task: A%2011052400
# 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-33191VL.greta.niwa.co.nz"
export CYLC_SUITE_PORT="NONE"
export CYLC_SUITE_DIR="/home/oliverh/cylc/examples/CUG7.4"
export CYLC_SUITE="examples:CUG74"
export CYLC_SUITE_GROUP="examples"
export CYLC_SUITE_NAME="CUG74"
export CYLC_SUITE_OWNER="oliverh"
export CYLC_USE_LOCKSERVER="True"
export CYLC_LOCKSERVER_PORT="7767"
export CYLC_SIMULATION_SLEEP="10"

# TASK IDENTITY:
export TASK_ID=A%2011052400
export TASK_NAME=A
export CYCLE_TIME=2011052400

# ACCESS TO CYLC:
. $CYLC_DIR/environment.sh
```



```
# 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 any signals which could cause the 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

# GLOBAL VARIABLES:
NEXT_CYCLE="$( cylc util cycletime -a 12 )"
PREV_CYCLE="$( cylc util cycletime -p 12 )"
export NEXT_CYCLE PREV_CYCLE

# LOCAL VARIABLES:
INPUT="foo-${NEXT_CYCLE}.txt"
export INPUT

# TASK COMMAND SCRIPTING:
A.sh

# SEND TASK SUCCEEDED MESSAGE:
cylc task succeeded
trap "" EXIT

echo "JOB SCRIPT EXITING (TASK SUCCEEDED)"

#EOF
```

9.2 Available Methods

There are two basic 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.

9.3 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 random component so that rerunning the task won't overwrite the old output, e.g.:

```
ColdB%2011101300-N3pBYv.out
ColdB%2011101300-N3pBYv.err
```

You can set the location for these task output logs in the suite.rc file; the default is,

```
# SUITE.RC
job submission log directory = $HOME/CylcJobLogs/$CYLC_SUITE_GROUP/$CYLC_SUITE_NAME
```

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. 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 resubmit sub-jobs and direct the output thereof to other files). For less internally complex tasks, however, these job logs will capture all stdout and stderr for your tasks, and *they can be viewed updating in real time in the suite control GUI.*

9.4 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/src/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, 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 job script directly in a background shell.
    """
    def construct_jobfile_submission_command( self ):
        # stdin redirection allows background execution on remote hosts
        self.command = self.jobfile_path + " </dev/null" + \
                " 1> " + self.stdout_file + " 2> " + self.stderr_file + " &"
# EOF
```

Here is the `at_now` method:

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

from job_submit import job_submit

class at_now( job_submit ):
    """
Submit the job script to the 'at' scheduler, to run 'now'.
    """
    def construct_jobfile_submission_command( self ):
        self.command = 'at now <<EOF\n' + self.jobfile_path + \
                ' 1> ' + self.stdout_file + ' 2> ' + self.stderr_file + '\nEOF'
# EOF
```

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.
    """
    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 ):
        self.command = 'llsubmit ' + self.jobfile_path
# EOF
```

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/src/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.

10 Getting More Information

Some of the following topics are yet to be properly documented in the User Guide. For the moment, 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 examples in this document (particularly the *Quick Start Guide*, Section 5).

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

- running single tasks outside or inside suites (see `cylc insert help` and `cylc submit 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`)
- where to look for output - suite log, suite stdout/stderr, task stdout/stderr (this is documented in the Quick Start Guide)
- 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.

10.1 Currently Undocumented Advanced Topics

- Spin-up processes via temporary tasks and adding prerequisites on-the-fly.
- How to recover from certain kinds of task failure.
- Sub-suites - running another suite inside a task:

```
[tasks]
    [[foo]]
        command = "cylc run SUITE $CYCLE_TIME --until=$CYCLE_TIME"
```

11 Asynchronous Tasks (No Cycle Time)

Documentation of this new feature (post version 3.4.0) will eventually be integrated into the preceding sections of the User Guide.

Asynchronous tasks have an arbitrary integer tag instead of a cycle time. Cylc can manage both asynchronous and cycling tasks within the same suite.

11.1 One-off Asynchronous Tasks

These tasks are defined with no sub-heading under [dependencies] in the suite.rc file:

```
# SUITE.RC
[dependencies]
    graph = "foo => bar => baz"
    [[0,12]]
        graph  =  "baz => CyclingA => CyclingB"
```

In this listing two cycling tasks are made to wait on a chain of three one-off asynchronous tasks (they can be used in entirely non-cycling suites, or for initial processing prior to starting the cycling tasks).

11.1.1 Comparison with One-off Non-asynchronous "startup" Tasks

One-off non-asynchronous startup 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 starting is only meaningful for cycling tasks, however, so *one-off asynchronous tasks run every time a suite is started* (restarts just resurrect the prior suite state, as usual). Thus, while one-off asynchronous tasks can precede cycling tasks in the same suite, as shown above, it may be that non-asynchronous startup tasks are more appropriate for this purpose, and that one-off asynchronous tasks are best used in constructing entirely non-cycling suites.

11.1.2 One-off Asynchronous Example Suite

See suite `[admin]:examples:async_oneoff`

11.2 Repeating Asynchronous Tasks

These tasks can be used to construct suites for processing asynchronous satellite datasets, wherein the data arrives at (effectively) random intervals. The first task in the suite should be of the *daemon* type, which runs indefinitely watching for incoming data. When new data is detected it sends a message that matches a registered pattern containing some unique data ID. The daemon task proxy extracts the ID and dynamically registers a new output, containing the actual ID, that downstream tasks can trigger off. The downstream tasks likewise have prerequisites containing the pattern, and when these are satisfied the actual ID is substituted into their registered outputs. Each asynchronous task proxy passes the ID to its real task as `$ASYNCID` so that it can process the correct dataset. Repeating asynchronous tasks are defined under special sub-headings in the [dependencies] section of the suite.rc graph, e.g.:

```
# SUITE.RC
[dependencies]
    [[ASYNCID:SATID-\d+]]
        graph  =  "watcher:a => foo:a & bar:a => baz"
        daemon = watcher
[tasks]
    [[watcher]]
        # ...
        [[[outputs]]]
            a = "New dataset $(ASYNCID) ready for processing"
    [[foo]]
        # ...
        [[[outputs]]]
            a = "Products generated from dataset $(ASYNCID)"
# ...
```

If the asynchronous datasets happen to become available in quick succession, entire trees of asynchronous processing can run in parallel.

11.2.1 Repeating Asynchronous Example Suite

See suite `[admin]:examples:async_repeat`

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 Use Include-Files For Groups Of Related Tasks

Suite.rc include-files can be used just to help organise tasks into convenient groups in very large suites (and with `cylc edit` you can edit a temporarily inlined file to get a global view). But they are most useful for handling the repetitive definition of groups of similar tasks, because the same inclusion can be used multiple times in the same suite.rc file. See *Include Files* (Section A.1).

12.3 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.

12.4 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.

12.5 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*.

12.6 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.

12.6.1 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.

12.7 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.

12.8 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.

12.9 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_GROUP/$CYLC_SUITE_NAME/
```

(these variables are automatically exported to the task execution environment by cylc - see *Task Execution Environment*, Section 7.4.3). 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.

12.10 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.[17] 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 #...
```

12.11 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;[18] other files there will be ignored by cylc but suite tasks can access them via the `$CYLC_SUITE_DIR` 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.

12.12 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.

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

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

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 so that successive instances cannot run out of sequence, or in parallel, even if the opportunity arises.

12.13 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).

12.14 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.

A Suite.rc Reference

This section documents all legal entries in a suite.rc file. The information is extracted from the specification file `$CYLC_DIR/conf/suiterc.spec` during document processing, so that the User Guide is automatically kept up to date. Section headings are generated with the same nesting structure as the file itself to make it easy to get at the documentation you need by clicking on hyperlinks in the PDF document contents page.

Most suites will only need a few of these settings - many configuration items have default values that should be sufficient, some are probably only needed for critical operational suites (e.g. secure passphrases and task event hooks), and some are primarily used for cylc development (e.g. simulation mode configuration). In general your suite.rc files shouldn't be a lot more complicated than those of the cylc example suites.

A.1 Include Files

Include-files can be used to help organize the task definition sections of large suites, or to group common environment variable settings into one file that can be included in multiple task environment sections (instead of polluting the global namespace for *all* tasks). Include-file boundaries are arbitrary (they can cross suite.rc section boundaries). They can be multiply included and nested.

```
%include path/to/myfile.rc
```

Include-file paths should be specified portably[19] relative to the suite definition directory. The `cylc edit` command can optionally provide an inlined version of a suite.rc file that is automatically split back into its constituent include-files when you save the file and exit the editor.

A.2 Suite Level Items

A.2.1 title

The suite title is displayed in the gcylc suite database window, and can also be retrieved from a suite at run time using the `cylc show` command.

- *section:* (top level)
- *type:* string
- *default:* "No title supplied"
- *example:* `title = "Suite Foo"`

A.2.2 description

The suite description can be retrieved by gcylc right-click menu and the `cylc show` command.

- *section:* (top level)
- *type:* string
- *default:* "No description supplied"
- *example:*

```
description = """
Here's what this suite does ...
... on a good day"""
```

[19] If the suite is copied to another location you shouldn't have to change hardwired paths.

A.2.3 initial cycle time

Initial suite cycle time. At startup each cycling task will be inserted into the suite with this cycle time, or with the closest subsequent valid cycle time for the task (unless excluded by the *tasks to include—exclude at startup* items; and note that how cold start tasks are inserted, or not, depends on the start up method - cold, warm, or raw). Alternatively you can provide, or override, the initial cycle time on the command line or suite start GUI panel.

- *section:* (top level)
- *type:* integer
- *default:* None
- *example:* `initial cycle time = 2011052318`

A.2.4 final cycle time

Final suite cycle time. Cycling tasks will be held (i.e. not allowed to spawn a successor) after passing this cycle time. When all tasks have reached this time the suite will shut down (unless it also contains still-running asynchronous tasks). Alternatively you can provide, or override, the final cycle time on the command line or suite start GUI panel.

- *section:* (top level)
- *type:* integer
- *default:* None
- *example:* `final cycle time = 2011052318`

A.2.5 job submission method

The default job submission method for the suite. This determines how cylc job scripts are executed when a task is ready to run - see Section 9.

- *section:* (top level)
- *type:* string
- *legal values:*
  - `background` - direct subshell execution in the background
  - `at_now` - the rudimentary Unix 'at' scheduler
  - `loadleveler` - loadleveler, generic
  - `ll_ecox` - loadleveler, customized for EcoConnect triplicate environment at NIWA
  - `ll_raw` - loadleveler, for existing job scripts
- *default:* `background`
- *example:* `job submission method = at_now`

A.2.6 use lockserver

The cylc lockserver generalizes traditional *lock files* to the network. It prevents prevents invocation of multiple instances of the same suite at the same time, or invocation of a task (using `cylc submit`) if the same task is already running (in its suite or by `cylc submit`). It will allow multiple instances of a suite to run under different registrations *only if* the suite declares itself capable of that (see `suite.rc` item "allow multiple simultaneous instances"). The lockserver cannot prevent you from running distinct *copies* of a suite simultaneously. See `cylc lockserver --help` for how to run the lockserver, and `cylc lockclient --help` for occasional manual lock management requirements. The lockserver is currently disabled by default.

- *section:* (top level)
- *type:* boolean
- *default:* False
- *example:* `use lockserver = True`

A.2.7 remote host

If a suite level remote host is specified cylc will attempt to run every task on that host, except for particular tasks that override the host setting, by passwordless ssh. Use this if all of your tasks, or at least the bulk of them, run on the same remote host, otherwise define remote hosts at task level. The relevant suite task scripts and executables, and cylc itself, must be installed on the remote host. The items *remote cylc directory* and *remote suite directory* must also be specified at suite and/or task level, and *owner* must be defined (at suite and/or task level) if the task owner's username on the remote host is not the same as the local suite owner's. Passwordless ssh must be configured between the local suite owner and remote task owner accounts.

- *section:* (top level)
- *type:* string
- *legal values:* a valid hostname on your network
- *default:* None
- *example:* `remote host = foo.niwa.co.nz`

A.2.8 remote cylc directory

For tasks that declare a remote host at suite level, this defines the path to the remote cylc installation (i.e. `$CYLC_DIR`). Use this if all of your tasks, or at least the bulk of them, run on the same remote host, otherwise define the remote cylc directory at task level. on the remote host.

- *section:* (top level)
- *type:* string
- *legal values:* a valid directory path on the remote host
- *default:* None
- *example:* `remote cylc directory = /path/to/cylc/on/remote/host`

This item is compulsory for remotely hosted tasks.

A.2.9 remote suite directory

For tasks that declare a remote host at suite level, this specifies the path to the suite definition directory on the remote host, in order to give remote tasks access to files stored there (via `$CYLC_SUITE_DIR`) and to the suite bin directory (via `$PATH`). Use this suite level item if all of your tasks, or at least the bulk of them, run on the same remote host, otherwise define the remote suite directory at task level.

- *section:* (top level)
- *type:* string
- *legal values:* a valid directory path on the remote host
- *default:* None
- *example:* `remote suite directory = /path/to/suite/on/remote/host`

This item is not compulsory for remotely hosted tasks, because some tasks may not require access to files in the suite definition directory.

A.2.10 owner

If a task has a defined owner, cylc will attempt to execute the task as that user, according to the suite level *owned task execution method*. Use this if all of your tasks, or at least the bulk of them, run under the same username, otherwise define task owners at task level (or not at all, if all tasks run as the suite owner, which is the usual situation).

- *section:* (top level)
- *type:* string
- *legal values:* a valid username on the task host
- *default:* None
- *example:* `owner = bob`

A.2.11 use secure passphrase

If True, any intervention in a running suite will require a special passphrase to be present, with secure permissions (as for ssh keys) in the file `$HOME/.cylc/security/GROUP:NAME`. The passphrase file must be present in any user account that needs access to the suite (remotely hosted tasks for instance). The passphrase itself is never transferred across the network (a secure MD5 checksum is). This guarantees suite security so long as your user account isn't breached.

- *section:* (top level)
- *type:* boolean
- *default:* False
- *example:* `use secure passphrase = True`

A.2.12 tasks to exclude at startup

Any task listed here will be excluded from the initial task pool (this goes for 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 may be required).

- *section:* (top level)
- *type:* list of task names
- *default:* empty
- *example:* `tasks to exclude at startup = TaskA, TaskB, TaskC`

A.2.13 tasks to include at startup

If this list is not empty, any task NOT listed in it will be excluded from the initial task pool (this goes for 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 may be required).

- *section:* (top level)
- *type:* list of task names
- *default:* empty
- *example:* `tasks to include at startup = TaskA, TaskB, TaskC`

A.2.14 runahead limit in hours

If a task's cycle time is ahead of the oldest non-failed task in the suite by more than this limit, it will be prevented from spawning a successor until the slowest tasks catch up sufficiently. Failed tasks (which are not automatically removed from a suite) do not activate the runahead limit (but downstream dependencies that can't run because of them will). In real time operation the runahead limit is of little consequence because the suite will be constrained by its real time clock-triggered tasks (however, it must be long enough to cover the range of tasks present in the suite; for example a task that only runs once per day needs to spawn 24 hours ahead). The runahead limit is intended to stop fast tasks from running off far into the future in historical case studies.

- *section:* (top level)
- *type:* integer
- *legal values:* >= 0
- *default:* 24
- *example:* `runahead limit in hours = 48`

A.2.15 suite log directory

Cylc logs all events to a suite log file. The main log and its automatic backups are stored under this directory. *You must ensure the directory is suite-specific; this can be done without hard wiring by using suite identity environment variables as the default value does.*

- *section:* (top level)
- *type:* string
- *legal values:* absolute path, may contain environment variables such as `$HOME`.
- *default:* `$HOME/CylcSuiteLogs/$CYLC_SUITE_GROUP/$CYLC_SUITE_NAME`
- *example:* `suite log directory = $HOME/CSL/$CYLC_SUITE_GROUP/$CYLC_SUITE_NAME`

A.2.16 roll log at startup

Roll the cylc log for the suite (i.e. relabel ordered backups and start the main log anew) when the suite is started or restarted.

- *section:* (top level)
- *type:* boolean
- *default:* True
- *example:* `roll log at startup = False`

A.2.17 state dump directory

Suite state dump files allow cylc to restart suites from previous states. The default state dump and its backups, and special pre-intervention state dumps are all stored under this directory. *You must ensure the directory is suite-specific; this can be done without hard wiring by using suite identity environment variables as the default value does.*

- *section:* (top level)
- *type:* string
- *legal values:* absolute path, may contain environment variables such as `$HOME`.
- *default:* `$HOME/CylcStateDumps/$CYLC_SUITE_GROUP/$CYLC_SUITE_NAME`
- *example:* `state dump directory = $HOME/CSD/$CYLC_SUITE_GROUP/$CYLC_SUITE_NAME`

A.2.18 number of state dump backups

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 that can also be used to restart the suite are written out prior to actioning any suite intervention.*

- *section:* (top level)
- *type:* integer
- *legal values:* >= 1
- *default:* 10
- *example:* `number of state dump backups = 20`

A.2.19 job submission log directory

The directory in which to put the stdout and stderr log files for the job scripts submitted by cylc when tasks are ready to run. For monolithic tasks (which don't resubmit sub-jobs themselves) these will be the complete job logs for the task. For owned tasks, the suite owner's home directory will be replaced by the task owner's.

- *section:* (top level)
- *type:* string
- *legal values:* absolute path, may contain environment variables such as `$HOME`.
- *default:* `$HOME/CylcJobLogs/$CYLC_SUITE_GROUP/$CYLC_SUITE_NAME`
- *example:* `job submission log directory = $HOME/Logs/$CYLC_SUITE`

For remotely hosted tasks this configuration item is currently ignored - task output logs are written to the remote task owner's home directory. (This will be addressed in a future cylc release).

A.2.20 task EVENT hook scripts

Task event hooks facilitate centralized alerting for critical events. The following hooks are available:

- **task submitted hook script**
- **task submission failed hook script**
- **task started hook script**
- **task succeeded hook script**
- **task failed hook script**
- **task warning hook script**
- **task timeout hook script**

These suite level defaults can be overridden by specific tasks, or you can omit the defaults and just attach hook scripts for critical tasks. Cylc provides a hook script that sends emails: cylc email-alert. Your own hook scripts can be located in suite bin directories. Hook scripts are passed the following arguments:

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


where MESSAGE describes what has happened; EVENT is either 'submitted', 'started', 'succeeded', 'failed', 'timeout', or 'submission_failed'; and TASKID is the unique task identifier (e.g. NAME%CYCLE_TIME for cycling tasks). Note that *hook scripts are called by cylc, not by tasks,* so if you wish to pass in additional information via the environment, use the [cylc local environment] section, not [environment].

- *section:* (top level)
- *type:* string
- *default:* None
- *example:* `task failed hook script = cylc email-alert`

A.2.21 task EVENT timeout in minutes

You can set timeout intervals for task submission or execution with the following items:

- **task submission timeout in minutes**
- **task execution timeout in minutes**

If a task has not started (or finished) this number of minutes after it was submitted (or started), the task timeout hook script will be called by cylc with the following arguments:

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

where message describes what has happened; EVENT is 'submission' or 'execution'; and TASKID is the unique task identifier (e.g. NAME%CYCLE_TIME for cycling tasks). Like the hook scripts themselves, these suite level settings can be overridden by specific tasks.

- *section:* (top level)
- *type:* float (minutes)
- *default:* None
- *example:* `task execution timeout in minutes = 10`

A.2.22 reset execution timeout on incoming messages

If True, and you have set an execution timeout, the timer will reset to zero every time a message is received from a running task. Otherwise, the task will timeout if it does not finish in time, even if it last sent a message (and was, by implication, still alive) within the timeout interval.

- *section:* (top level)
- *type:* boolean
- *default:* True
- *example:* `reset execution timeout on incoming messages = False`

A.2.23 pre-command scripting

Scripting to be executed verbatim in the task execution environment, before the task command, for every task.

- *section:* (top level)
- *type:* multiline string
- *default:* empty
- *example:*

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

A.2.24 post-command scripting

Scripting to be executed verbatim in the task execution environment, immediately after the task command, for every task.

- *section:* (top level)
- *type:* multiline string
- *default:* empty
- *example:*

```
post-command scripting = """
  . $HOME/.profile
  echo Goodbye from suite $CYLC_SUITE!"""
```

A.2.25 owned task execution method

This specifies the means by which the chosen job submission method is invoked for tasks that are owned by a user other than the suite owner.[20]

- *section:* (top level)
- *type:* string
- *legal values:*
  - sudo
  - ssh
- *default:* sudo
- *example:* `owned task execution method = ssh`

To use sudo with loadleveler, for example, `/etc/sudoers` must be configured to allow the suite owner to execute the `llsubmit` command as the designated task owner. To use ssh, passwordless ssh must be configured between the accounts of the suite and task owners.

A.2.26 ignore task owners

This item allows you to turn off the special treatment of owned tasks (namely invocation of the task job submission method via sudo or ssh as task owner), which can be useful when testing parts of a suite containing owned tasks outside of its normal operational environment.

- *section:* (top level)
- *type:* boolean
- *default:* False
- *example:* `ignore task owners = True`

A.2.27 use quick task elimination

If quick task elimination is switched on (it is by default) cylc will remove spent tasks from the suite sooner if they are known to have no downstream dependents in subsequent forecast cycles. Otherwise the generic spent task elimination algorithm will be used for all tasks. (Mainly used in cylc development).

[20] Why would you want to do this? At NIWA, parts of our complex multi-model operational suite are deployed into, and run from, role accounts that (at least in the development and test environments) are managed by the experts on the associated scientific model or subsystem.

- *section:* (top level)
- *type:* boolean
- *default:* True
- *example:* `use quick task elimination = False`

A.2.28 simulation mode only

If True, cylc will abort cleanly if you try to run the suite in real mode. This can be used for demo suites that can't run in real mode because they've been copied out of their normal operating environment.

- *section:* (top level)
- *type:* boolean
- *default:* False
- *example:* `simulation mode only = True`

A.2.29 allow multiple simultaneous instances

If True, the lockserver will allow multiple instances of this suite to run at the same time under different registrations. You can do this if the I/O paths of every task in the suite are dynamically configured to be suite specific (i.e. they all contain the suite registration group and name).

- *section:* (top level)
- *type:* boolean
- *default:* False
- *example:* `allow multiple simultaneous instances = True`

A.2.30 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.* Global pre- and post-command scripting, and the content of the task commands themselves, must be valid in the job submission shell. The suite environment sections must be converted similarly; this is currently hardwired into cylc as `export item=value` (which works for both bash and ksh because `value` is entirely user-defined) so cylc would have to be modified slightly if other shells are needed.

- *section:* (top level)
- *type:* string
- *legal values:*
  - `/bin/bash`
  - `/bin/ksh`
  - `/usr/bin/bash`
  - `/usr/bin/ksh`
- *default:* `/bin/bash`
- *example:* `job submission shell = /bin/ksh`

A.2.31 manual task completion messaging

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 insert some minimal cylc messaging in appropriate places in the task implementation. Use this global setting in the unlikely event that all, or most, of your tasks are in this category; otherwise you can set it on a per task basis.

- *section:* (top level)
- *type:* boolean
- *default:* `False`
- *example:* `manual task completion messaging = True`

A.3 [special tasks]

This section identifies any tasks with special behaviour. By default tasks:

- start running as soon as their prerequisites are satisfied
- spawns a successor at its next valid cycle time as soon as it enters the running state[21]

A.3.1 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.

- *section:* [special tasks]
- *type:* list of tasknames followed by parenthesized offsets, in hours
- *default:* None
- *legal values:* The offset, in hours, can be positive or negative.
- *example:* `clock-triggered = foo(1.5), bar(2.25)`

Clock-triggered tasks currently can't be triggered manually prior to their trigger time. This will change in a future cylc release. In the meantime if you need to do this you can kill the task with `cylc remove`, run it manually outside of the suite with `cylc submit`, and then manually trigger any direct downstream dependencies of the killed task.

A.3.2 startup

Startup tasks are one off tasks that are only used when *cold starting a suite*, i.e. when starting up without assuming any previous cycle. A startup task can be used to clean out or prepare a suite workspace, for example, before other tasks run.

- *section:* [special tasks]
- *type:* list of task names
- *default:* empty list
- *example:* `startup = TaskA, TaskB`

A.3.3 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

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

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 a serious problem (e.g. critical inputs not available). Cold start tasks can invoke real cold start processing, or they can just be dummy tasks (which don't specify a command) that stand in for some external process that has to be completed before the suite is started.

- *section:* [special tasks]
- *type:* list of task names
- *default:* empty list
- *example:* `cold start = ColdA, TaskF`

A.3.4 sequential

By default, a cylc task spawns a successor when 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 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 "models with explicit restart outputs" below).[22]

- *section:* [special tasks]
- *type:* list of task names
- *default:* empty list
- *example:* `sequential = ModelA, PostProcB`

A.3.5 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. *Startup* and *cold start* tasks are automatically one off tasks and do not need to be listed here.

- *section:* [special tasks]
- *type:* list of task names
- *default:* empty list
- *example:* `one off = TaskA, TaskB`

A.3.6 models with explicit restart outputs

This is only required in the unlikely event that you want a warm cycled forecast model to be able to start at the instant its restart files are ready (if other prerequisites are satisfied) *before its previous instance has finished*. If so, the task has to depend on a special output message 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, *which must contain the word "restart"*, in [tasks] → [[TASK]] → [[[outputs]]] - see Section A.8.1.16.

[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.

- *section:* [special tasks]
- *type:* list of task names
- *default:* empty list
- *example:* `models with explicit restart outputs = A, B`

A.4 [task families]

A task family is a named group of tasks that appears as a single task in the suite dependency graph. Thus the entire family triggers as a group, and downstream tasks can trigger off the entire family finishing. Task families can have internal dependencies, and family members can also appear in the graph as non-family tasks. See Section 7.3.11 for more information on task families.

A.4.1 <MANY>

Replace MANY with each task family definition.

- *section:* [task families]
- *type:* list of task names (the family members)
- *default:* None
- *example:* `ObsProc = ObsSurface, ObsSonde, ObsAircraft, ObsSat`

A.5 [dependencies]

The suite dependency graph should be defined under this section.

A.5.1 graph

Define the graph of any one-off asynchronous tasks (no cycle time) here. See Section A.5.2.1 below for details.

A.5.2 [[<MANY>]]

Replace MANY with each list of hours preceding a section of the suite dependency graph, as required for differing dependencies at different hours, *and* with any repeated asynchronous graph sections for satellite data processing or similar.

- *section:* [dependencies]
- *type:* list of integer hours, *or* string (for repeated asynchronous)
- *legal values:* $0 \leq hour \leq 23$, *or* string
- *default:* None
- *example:* `[[0,6,12,18]]`, *or* `[[ASYNID:SAT-\d+]]`

A.5.2.1 graph

Define the dependency graph valid for specified list of hours or asynchronous ID pattern. You can use the `cylc graph` command, or right click Graph in gcylc, to plot the dependency graph as you work on it. See Section 7.3 for details.

- *section:* [dependencies] → [[HOURS]]
- *type:* multiline string
- *legal values: refer to section 7.3*

- *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 | X:fail => Z            # Z triggers if X finishes or fails
   """
```

- *default:* None

A.5.2.2 daemon

For [[ASYNCID:pattern]] graph sections only, list any *asynchronous daemon* tasks by name.

- *section:* [tasks] → [[ASYNCID:pattern]]
- *type:* list of task names
- *default:* empty list
- *example:* `daemon = A, B`

A.6 [environment]

Use this section to define the global task execution environment, i.e. variables made available to all tasks. Order of definition is preserved. Even global variables can reference task name and cycle time because suite and task identity are exported in the cylc job script prior to the user-defined environment. Cylc utility commands can be used in variable assignment expressions because the cylc environment is exported prior to user-defined variables. See *Task Execution Environment*, Section 7.4.3, for more information.

A.6.1 <MANY>

Replace MANY with each global environment variable definition

- *section:* [environment]
- *type:* string
- *default:* None
- *legal values:* any environment variable assignment expression valid in the *job submission shell* (Appendix A.2.30). White space around the '=' is allowed (the `suite.rc` file is not itself 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}"`

  Variable expansion expressions containing the hash character must be quoted because hash is the suite.rc comment delimiter.

A.7 [directives]

Use this section to define batch queue scheduler directives, or similar, for all tasks in the suite. These are written to the top of the job script that cylc submits when a task is ready to run. Whether or not directives defined here are used depends on the task's job submission method, which should also define the directive comment prefix ('`# @`' for loadleveler) and final directive ('`# @ queue`').

A.7.1 <MANY>

Replace MANY with each batch queue scheduler directive, e.g.

```
    class = parallel
```

- *section:* [directives]
- *type:* string
- *legal values:* any legal directive for your batch scheduler
- *default:* None

A.8 [tasks]

A.8.1 [[<MANY>]]

Replace MANY with each task name, followed by configuration items for that task.

- *section:* [tasks]
- *example:* `[[TaskF]]`

A.8.1.1 description

A description of the task, retrievable at run time by the `cylc show` command.

- *section:* [tasks] → [[TASK]]
- *type:* string
- *default:* "No description supplied"
- *example:*

```
description = """
Here's what this task does ...
... on a good day"""
```

A.8.1.2 command

This is the scripting to execute when the task is ready to run. If omitted the task will run as a dummy task (see the default command below). It can be a single command line or verbatim scripting inside a multiline string. If a list of command lines (or of mulitline scripting strings) is provided, the task will automatically resubmit with the second command/script if the first fails, and so on - this can be used for automated error recovery.

- *section:* [tasks] → [[TASK]]
- *type:* string
- *default:* `"echo DUMMY $TASK_ID; sleep $CYLC_SIMULATION_SLEEP"`
- *example:* `GetData.sh OPTIONS ARGUMENTS`

A.8.1.3 job submission method

Set the job submission method for this task, overriding the suite default (if there is one). This determines how cylc job scripts are executed when a task is ready to run. See *Task Execution*, Section 9.

- *section:* [tasks] → [[TASK]]
- *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`
- *example:* `job submission method = at_now`

A.8.1.4 job submission log directory

Set a job submission log directory for this task, overriding the suite default, for the stdout and stderr logs from the job scripts submitted by cylc when tasks are ready to run. For monolithic tasks (which don't resubmit sub-jobs themselves) these will be the complete job logs for the task. For owned tasks, the suite owner's home directory will be replaced by the task owner's.

- *section:* [tasks] → [[TASK]]
- *type:* string
- *legal values:* absolute path, may contain environment variables such as `$HOME`.
- *default:* None (see suite level default)
- *example:* `job submission log directory = $HOME/TaskXLogs/$CYLC_SUITE`

For remotely hosted tasks this configuration item is currently ignored - task output logs are written to the remote task owner's home directory. (This will be addressed in a future cylc release).

A.8.1.5 owner

If a task has a defined owner, cylc will attempt to execute the task as that user, according to the suite level *owned task execution method* for local tasks, or by passwordless ssh for remote tasks.

- *section:* [tasks] → [[TASK]]
- *type:* string
- *legal values:* a valid username on the task host
- *default:* None
- *example:* `owner = alice`

A.8.1.6 remote host

If a task specifies a remote host, cylc will attempt to execute the task on that host, using the specified job submission method, by passwordless ssh. The relevant suite task scripts and executables, and cylc itself, must be installed on the remote host. The task must also specify

remote cylc directory and *remote suite directory*. An *owner* must be defined if the task owner's username on the remote host is not the same as the local suite owner's. Passwordless ssh must be configured between the local suite owner and remote task owner accounts.

- *section:* [tasks] → [[TASK]]
- *type:* string
- *legal values:* a valid hostname on your network
- *default:* None
- *example:* `remote host = thor.niwa.co.nz`

A.8.1.7 remote cylc directory

For remotely hosted tasks, this must be used to specify the path to the cylc installation (i.e. `$CYLC_DIR`) on the remote host.

- *section:* [tasks] → [[TASK]]
- *type:* string
- *legal values:* a valid directory path on the remote host
- *default:* None
- *example:* `remote cylc directory = /path/to/cylc/on/remote/host`

Every remotely hosted task must set this item, either here or at suite level.

A.8.1.8 remote suite directory

For remotely hosted tasks, this specifies the path to the suite definition directory on the remote host, in order to give the task access to files in the stored there (via `$CYLC_SUITE_DIR`) and in the suite bin directory (via `$PATH`).

- *section:* [tasks] → [[TASK]]
- *type:* string
- *legal values:* a valid directory path on the remote host
- *default:* None
- *example:* `remote suite directory = /path/to/suite/on/remote/host`

This item is not compulsory for remotely hosted tasks, because some tasks may not require access to files in the suite definition directory.

A.8.1.9 task EVENT hook scripts

Task event hooks facilitate centralized alerting for critical events. The following hooks are available:

- **task submitted hook script**
- **task submission failed hook script**
- **task started hook script**
- **task succeeded hook script**
- **task failed hook script**
- **task warning hook script**
- **task timeout hook script**

These are task-specific hooks; you can also set suite level defaults. Cylc provides a hook script that sends emails: cylc email-alert. Your own hook scripts can be located in suite bin directories. Hook scripts are passed the following arguments:


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

where MESSAGE describes what has happened; EVENT is either 'submitted', 'started', 'succeeded', 'failed', 'timeout', or 'submission_failed'; and TASKID is the unique task identifier (e.g. `NAME%CYCLE_TIME` for cycling tasks). Note that *hook scripts are called by cylc, not by tasks,* so if you wish to pass in additional information via the environment, use the [cylc local environment] section, not [environment].

- *section:* [tasks] → [[TASK]]
- *type:* string
- *default:* None
- *example:* `task failed hook script = cylc email-alert`

A.8.1.10 task EVENT timeout in minutes

You can set timeout intervals for task submission or execution with the following items:

- **task submission timeout in minutes**
- **task execution timeout in minutes**

If a task has not started (or finished) N minutes after it was submitted (or started), the task timeout hook script will be called by cylc with the following arguments:

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

where MESSAGE describes what has happened; EVENT is 'submission' or 'execution'; and TASKID is the unique task identifier (e.g. `NAME%CYCLE_TIME` for cycling tasks). Like the hook scripts, these are task-specific settings; you can also set suite level defaults.

- *section:* [tasks] → [[TASK]]
- *type:* float (minutes)
- *default:* None
- *example:* `task execution timeout in minutes = 10`

A.8.1.11 reset execution timeout on incoming messages

If True, and you have set an execution timeout, the timer will reset to zero every time a message is received from a running task. Otherwise, the task will timeout if it does not finish in time, even if it last sent a message (and was, by implication, still alive) within the timeout interval.

- *section:* [tasks] → [[TASK]]
- *type:* boolean
- *default:* True
- *example:* `reset execution timeout on incoming messages = False`

A.8.1.12 extra log files

Any files named here will be added to the list of task logs (stdout and stderr) viewable at run time via gcylc. The intention is to make easily accessible any output from tasks that resubmit sub-jobs at run time (i.e. they don't remain under the control of the job script initially submitted by cylc). *WARNING: this feature is not well tested.*

- *section:* [tasks] → [[TASK]]
- *type:* list of strings

- *legal values:* valid file paths, may contain environment variables
- *default:* empty
- *example:* `extra log files = /a/b/c, /d/e/f`

A.8.1.13 manual task completion messaging

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 insert some minimal cylc messaging in appropriate places in the task implementation. There is an equivalent global setting in the unlikely event that all, or most, of your tasks are in this category.

- *section:* (top level)
- *type:* boolean
- *default:* `None`
- *example:* `manual task completion messaging = True`

A.8.1.14 [[[environment]]]

Use this section to define the task-specific task execution environment. Variables defined here may refer to variables in the global environment. Order of definition is preserved. Cylc utility commands can be used in variable assignment expressions because the cylc environment is defined prior to the user-defined environment. See *Task Execution Environment* (Section 7.4.3) for more information.

A.8.1.14.1 <MANY>

Replace MANY with each task environment variable definition.

- *section:* [tasks] → [[TASK]] → [[[environment]]]
- *type:* string
- *legal values:* any environment variable assignment expression valid in the *job submission shell*. 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 = $( cylc cycletime --add=6 )`
  - `PREV_CYCLE = 'cylc cycletime -s 6'`
  - `ZAZ = "${FOO#bar}"` Variable expansion expressions containing the hash character must be quoted because hash is the suite.rc comment delimiter.
- *default:* None

A.8.1.15 [[[directives]]]

Use this section to define task-specific batch queue scheduler directives, or similar, for this task. These are written to the top of the job script that cylc submits when the task is ready to run. Whether or not directives defined here are used depends on the task's job submission method, which should also define the directive comment prefix ('`# @`' for loadleveler) and final directive ('`# @ queue`').

A.8.1.15.1 <MANY>

Replace MANY with each task batch queue scheduler directive.

- *section:* [tasks] → [[TASK]] → [[[directives]]]
- *type:* string
- *legal values:* any legal directive for your batch scheduler
- *default:* None
- *example:* `class = parallel`

A.8.1.16 [[[outputs]]]

Only required if other tasks trigger off specific internal outputs *of this task*, as opposed to triggering off it finishing. The task implementation must report the specified output message by calling `cylc task message OUTPUT_MESSAGE` when the corresponding real output has been completed.

A.8.1.16.1 <MANY>

Replace MANY with each output message definition, for any explicit output messages emitted by this task and depended on by other tasks in the dependency graph.

- *section:* [tasks] → [[TASK]] → [[[outputs]]]
- *type:* string
- *legal values:* a message containing `$(CYCLE_TIME)` with an optional offset as shown below. **Note the round parentheses** - this is not a shell variable, although without an offset it does correspond to the `$CYCLE_TIME` in the task execution environment.
- *default:* None
- *examples:*

```
foo = "sea state products ready for $(CYCLE_TIME)"
r6  = "nwp restart files ready for $(CYCLE_TIME+6)"
r12 = "nwp restart files ready for $(CYCLE_TIME+12)"
```

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

```
[dependencies]
   [[6,18]]
       graph = TaskA:foo => TaskB
```

A.9 [simulation mode]

Configuration items specific to running suites in simulation mode.

A.9.1 clock offset from initial cycle time in hours

Specify a clock offset of 0 to simulate real time operation, greater than zero to simulate catching up from a delay and transitioning to real time operation.

- *section:* [simulation mode]
- *type:* integer
- *legal values:* ≥ 0
- *default:* 24
- *example:* `clock offset from initial cycle time in hours = 6`

A.9.2 clock rate in seconds per simulation hour

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

- *section:* [simulation mode]
- *type:* integer
- *legal values:* ≥ 0
- *default:* 10
- *example:* `clock rate in seconds per simulation hour = 20`

A.9.3 task run time in seconds

Set the approximate number of **real** seconds that a dummy task takes to execute.

- *section:* [simulation mode]
- *type:* integer
- *legal values:* ≥ 0
- *default:* 10
- *example:* `task run time in seconds = 20`

A.10 [visualization]

Graph plotting configuration items for suite.rc and run time graphs. These do not affect the graph-based suite control interface.

A.10.1 initial cycle time

Initial cycle time for graph plotting.

- *section:* [visualization]
- *type:* integer
- *default:* 2999010106
- *example:* `initial cycle time = 2011052318`

A.10.2 final cycle time

Final cycle time for graph plotting. This should typically be just far enough ahead of the initial cycle time to show the full suite.

- *section:* [visualization]
- *type:* integer
- *default:* 2999010206
- *example:* `final cycle time = 2011052318`

A.10.3 show family members

Whether to plot task family members, or the family as a whole.

- *section:* [visualization]
- *type:* boolean
- *default:* False
- *example:* `show family members = True`

A.10.4 use node color for edges

Outgoing graph edges (dependency arrows) can be plotted in the same color as the upstream node (task); this can make it easier to follow a path through a complex graph.

- *section:* [visualization]
- *type:* boolean
- *default:* True
- *example:* `use node color for edges = False`

A.10.5 default node attributes

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

- *section:* [visualization]
- *type:* list of *quoted* `'attribute=value'` pairs
- *legal values:* see graphviz or pygraphviz documentation
- *default:* `'style=unfilled', 'color=black', 'shape=ellipse'`
- *example:* `default node attributes = 'style=filled', 'shape=box'`

A.10.6 default edge attributes

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

- *section:* [visualization]
- *type:* list of graph edge attributes
- *legal values:* see graphviz or pygraphviz documentation
- *default:* `'color=black'`
- *example:* `default edge attributes = 'color=red'`

A.10.7 [[node groups]]

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

A.10.7.1 <MANY>

Replace MANY with each node group. Tasks can appear in multiple groups.

- *section:* [visualization] → [[node groups]]
- *type:* list of task names
- *default:* empty
- *example:* `BigModels = TaskX, TaskY`

A.10.8 [[node attributes]]

Here you can assign graph node attributes to specific tasks or named groups of tasks defined in the [[node groups]] section.

A.10.8.1 <MANY>

Replace MANY for any specific tasks or named groups that you want to assign attributes to.

- *section:* [visualization] → [[node attributes]]
- *type:* list of *quoted* `'attribute=value'` pairs
- *legal values:* see graphviz or pygraphviz documentation
- *default:* None
- *example:*

```
BigModels = 'style=filled', 'color=blue'
TaskX = 'color=red'
```

A.10.9 [[run time graph]]

Cylc can generate run time graphs of resolved dependencies, i.e. what actually triggers off what as the suite runs. Use simulation mode to generate run time graphs very quickly.

A.10.9.1 enable

Run time graphing is disabled by default because it is mainly intended for cylc development and debugging (the run time graph can be compared with the suite.rc graph to ensure that new suite.rc graph elements are parsed correctly).

- *section:* [visualization][[run time graph]]
- *type:* boolean
- *default:* False
- *example:* `enable = True`

A.10.9.2 cutoff in hours

Each new task will be added to the run time graph, as the suite runs, unless its cycle time exceeds the initial cycle time by more than this cutoff value.

- *section:* [visualization][[run time graph]]
- *type:* integer
- *legal values:* >= 0
- *default:* 24
- *example:* `cutoff in hours = 12`

A.10.9.3 directory

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

- *section:* [visualization][[run time graph]]
- *type:* string
- *legal values:* a valid local file path
- *default:* `$CYLC_SUITE_DIR/graphing`
- *example:* `directory = $HOME/mygraph`

A.11 [task insertion groups]

Define named groups of tasks that can be inserted into a suite en mass, as if inserting a single task. May be useful for groups of related cold start tasks, for instance.

A.11.1 <MANY>

Replace MANY with each task insertion group.

- *section:* [task insertion groups]
- *type:* list of task names
- *default:* None
- *example:* `NWPCold = ColdX, ColdY`

A.12 [cylc local environment]

Use this section to add variables to the environment in which cylc itself runs. These variables will be available to processes spawned directly by cylc, namely timeout and alert hook scripts. *Do not use this section to alter the task execution environment - use the plain [environment] sections for that - variables defined in this section will only be available to tasks if local direct job submission methods are used.*

A.12.1 <MANY>

Replace MANY with each cylc local environment variable definition.

- *section:* [cylc local environment]
- *type:* string
- *default:* None
- *legal values:* any valid environment variable assignment expression. White space around the '=' is fine (the `suite.rc` file is not a shell script).
- *examples:* for the bash shell:
  - `FOO = $HOME/bar/baz`
  - `BAZ = $(echo "hello world")`
  - `WAZ = ${FOO%.jpg}.png`

A.13 [experimental]

Section for experimenting with new configuration items

A.13.1 live graph movie

Turning this item on will result in a new dot file being written to the suite graphing directory every time the suite state changes. These can later be converted into movie frames and animated with appropriate image processing tools. A script for automating this is currently in the cylc development repository.

- *section:* [experimental]
- *type:* boolean
- *default:* False
- *example:* `live graph movie = True`

B Command Reference


```
cylc release version: 3.4.0-a

USAGE:
  cylc help,--help,-h,?                            # this help page
  cylc -v,--version                                # release version
  cylc help CATEGORY                               # help by category
  cylc CATEGORY help                               # (ditto)
  cylc help [CATEGORY] COMMAND                     # command help
  cylc [CATEGORY] COMMAND help,--help              # (ditto)

  cylc [CATEGORY] COMMAND [options] SUITE [args]   # command syntax

Any unique abbreviation can be used for a COMMAND or CATEGORY, e.g.:
  $ cylc control trigger SUITE TASK  # manually trigger TASK in SUITE
  $ cylc con trig SUITE TASK         # ditto
  $ cylc c t SUITE TASK              # ditto!
Use of categories is optional,
  $ cylc trig SUITE TASK             # ditto
but they provide structure to the command set, and they disambiguate
highly abbreviated commands. Some commands are aliased (ls|list), and
some common typos are automatically corrected (cylc,cycl; help,hlep).

Note that 'SUITE' refers to a suite's registered [OWNER:]GROUP:NAME,
and 'TASK' refers to the unique NAME%TAG of a task in a suite. For
cycling tasks TAG is a cycle time (e.g. foo%2011080806), and for
asynchronous tasks it is an integer preceded by 'a:' (e.g. foo%a:1).

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:
  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:
  gcylc ..................... The cylc Graphical User Interface
  register .................. Add a suite to your private registration database
  reregister ................ Change private or central suite registrations
  unregister|delete ......... Remove suites from the private or central database
  copy|cp ................... Copy a suite or group of suites.
  print ..................... Print private or central suite registrations
  get-dir ................... Print a suite definition directory path
  export .................... Export private registrations to the central database
  import .................... Import central registrations to your private database
  refresh|check ............. Check for invalid registrations and update titles
  edit ...................... Edit a suite.rc file in $EDITOR, optionally inlined
  inline .................... View suite.rc files in $EDITOR, include-files inlined
  validate .................. Parse and validate a suite config (suite.rc) file
  search|grep ............... An intelligent search tool for cylc suites
  graph ..................... A dependency graph viewer that updates as you edit
  diff|compare .............. Compare two suite definitions and print differences
  list|ls ................... Print a suite's task list
  describe .................. Print a suite's title and description
  dump ...................... Print the state of each task in a running suite
  show ...................... Print task-specific information (prerequisites...)
  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
  ping ...................... Check that a suite is running
  scan ...................... Scan a host for running suites and lockservers
  run|start ................. Start a suite running at a specified cycle time
  stop|shutdown ............. Stop a suite running by various means
  restart ................... Restart a suite from a previous state
  trigger ................... Cause a task to trigger immediately
  insert .................... Insert a task or group into a running suite
  remove|kill ............... Remove a task from a running suite
  purge ..................... Remove a full dependency tree from a running suite
  hold ...................... Put a hold on a suite or a single task
  release|unhold ............ Release a  hold on a suite or a single task
  block ..................... Do not comply with subsequent intervention commands
  unblock ................... Comply with subsequent intervention commands
  reset ..................... Force a task to waiting, ready, or succeeded state
  depend .................... Add prerequisites to tasks on the fly
  maxrunahead ............... Change the runahead limit in a running suite.
  lockserver ................ The cylc lockserver daemon
  lockclient|lc ............. Manual suite and task lock management
  verbosity ................. Change a suite's logging verbosity level
  cycletime ................. Cycle time arithmetic
  checkvars ................. Check required environment variables en masse
  template .................. Powerful cycle time offset filename templating
  housekeeping .............. Parallel archiving and cleanup on cycle time offsets
  email-alert ............... Send alerts by email when tasks fail (for example)
  submit|single ............. Run a single task just as its parent suite would
  started|task-started ...... Acquire a task lock and report started
  message|task-message ...... Report progress and completion of outputs
  succeeded|task-succeeded ... Release task lock and report succeeded
  failed|task-failed ........ Release task lock and report failure
  create-cdb ................ (ADMIN) Create the central suite database
```



```
  test-db ..................... (ADMIN) Automated suite database test
  test-suite .................. (ADMIN) Automated cylc scheduler test
  warranty .................... Print the GPLv3 disclaimer of warranty
  conditions .................. Print the GNU General Public License v3.0
```

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:
  gcylc ............ The cylc Graphical User Interface
  run|start ........ Start a suite running at a specified cycle time
  stop|shutdown .... Stop a suite running by various means
  restart .......... Restart a suite from a previous state
  trigger .......... Cause a task to trigger immediately
  insert ........... Insert a task or group into a running suite
  remove|kill ...... Remove a task from a running suite
  purge ............ Remove a full dependency tree from a running suite
  hold ............. Put a hold on a suite or a single task
  release|unhold ... Release a  hold on a suite or a single task
  block ............ Do not comply with subsequent intervention commands
  unblock .......... Comply with subsequent intervention commands
  reset ............ Force a task to waiting, ready, or succeeded state
  nudge ............ Cause the cylc task processing loop to be invoked
  depend ........... Add prerequisites to tasks on the fly
  maxrunahead ...... Change the runahead limit in a running suite.
  lockserver ....... The cylc lockserver daemon
  lockclient|lc .... Manual suite and task lock management
  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:
  gcylc ............... The cylc Graphical User Interface
  register ............ Add a suite to your private registration database
  reregister .......... Change private or central suite registrations
  unregister|delete ... Remove suites from the private or central database
  copy|cp ............. Copy a suite or group of suites.
  print ............... Print private or central suite registrations
  get-dir ............. Print a suite definition directory path
  export .............. Export private registrations to the central database
  import .............. Import central registrations to your private database
  refresh|check ....... Check for invalid registrations and update titles
```

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:
  gcylc ...... The cylc Graphical User Interface
  list|ls .... Print a suite's task list
  describe ... Print a suite's title and description
  dump ....... Print the state of each task in a running suite
  show ....... Print task-specific information (prerequisites...)
  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
```

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:
  warranty ..... Print the GPLv3 disclaimer of warranty
  conditions ... Print the GNU General Public License v3.0
```

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:
  gcylc ......... The cylc Graphical User Interface
  edit .......... Edit a suite.rc file in $EDITOR, optionally inlined
  inline ........ View suite.rc files in $EDITOR, include-files inlined
  validate ...... Parse and validate a suite config (suite.rc) file
  search|grep .... An intelligent search tool for cylc suites
  graph ......... A dependency graph viewer that updates as you edit
  diff|compare ... Compare two suite definitions and print differences
```

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:
```

```
  submit|single ............... Run a single task just as its parent suite would
  started|task-started ....... Acquire a task lock and report started
  message|task-message ....... Report progress and completion of outputs
  succeeded|task-succeeded ... Release task lock and report succeeded
  failed|task-failed ......... Release task lock and report failure
```

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:
  cycletime ...... Cycle time arithmetic
  checkvars ...... Check required environment variables en masse
  template ....... Powerful cycle time offset filename templating
  housekeeping ... Parallel archiving and cleanup on cycle time offsets
  email-alert .... Send alerts by email when tasks fail (for example)
```

B.2 Commands

B.2.1 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 useful
when running important suites, or multiple suites at once.

(These commands are 'block/unblock' rather than 'lock/unlock' in order
to distinguish them from the functionality of the cylc lockserver
commands, to which they are unrelated).

You must be the owner of the target suite to use this command.

arguments:
   SUITE                Registered GROUP:NAME of the 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.2.2 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.2.3 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.2.4 copy

```
Usage: cylc [db] copy|cp [OPTIONS] FROM TO [DIR]

Copy suites or groups in your private suite database. If DIR is given, suite
definition directories will be copied to that location, otherwise the new
registration will refer to the old suite definition directory. Copying a
group to GROUP:foo results in the suite definition being copied to
DIR/foo, and so on.

If DIR is not given, a new reference to the original source suite will
be created (i.e. the suite definition directory will not be copied).

Arguments:
    FROM, TO - private suite registrations (GROUP:NAME) or groups (GROUP:)
    [DIR]    - destination for the copied suite definition (must not exist).
               or top level directory for copied suite definitions.

Options:
  -h, --help  show this help message and exit
```

B.2.5 create-cdb

```
USAGE: cylc [admin] create-cdb

This admin command should be used by the cylc administrator
to create the central suite registration database immediately
after installing cylc. It can be run multiple times without
any adverse affect.
```

B.2.6 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 $CYCLE_TIME:
   $ export CYCLE_TIME=2010082318  # or task execution environment
   $ cylc cycletime -s 6
  2010082312

Arguments:
   CYCLE        Cycle time (YYYYMMDDHH) defaults to $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.2.7 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 DEP 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 will not be 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                Registered GROUP:NAME of the target suite.
   TASK                 The target task.
   DEP                  The 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.2.8 describe

```
Usage: cylc [info] describe [OPTIONS] SUITE

Parse a suite config file and print the suite title and description.

Arguments:
```

```
   SUITE    - suite registration [OWNER:]GROUP:NAME
            (use OWNER: to access the central database)

Options:
  -h, --help   show this help message and exit
  -d, --debug  print the exception traceback for validation errors.
```

B.2.9 diff

```
Usage: cylc [prep] diff|compare [options] SUITE1 SUITE2

Compare two suite definitions and display any differences.

NOTE: 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 include-file content after
inlining.

Seemingly identical suites (immediately after copying one from the
other, for instance) can differ slightly if they use default
configuration items, such as the default job submission log directory,
that are suite-specific (i.e. the includes $CYLC_SUITE_NAME etc.).

Files in the suite bin directory, and other files held in the suite
definition directory, are not currently differenced (they may be
important in task implementations, but are not part of the suite
definition).

Arguments:
   SUITE1, SUITE2  - suite registrations [OWNER:]GROUP:NAME
                     (use OWNER: to access the central database)

Options:
  -h, --help    show this help message and exit
  -d, --debug   print the exception traceback for validation errors.
  -n, --nested  print suite.rc section headings in nested form.
```

B.2.10 dump

```
Usage: cylc [info] dump [options] SUITE

Print suite and task state information to stdout. All available
information is printed by default. Output is designed to be
grep-friendly. For small suites 'watch cylc [info] dump SUITE' is an
effective non-gui real time monitor.

For more information about a specific task, such as the current
state of each registered prerequisite and output, 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               Registered GROUP:NAME of the 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 instead of name.
```

B.2.11 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 config (suite.rc) files with include-files inlined between
special markers. After the file is saved *and you exit the editor*
the file will be split into its constituent include-files again.
This can be used for quick global suite editing when parameters of
interest span several include-files. Include-files can be nested or
multiply-included (in which case only the first inclusion will be
inlined).

3/ Remove backup files left by previous *inlined* edit sessions.

SAFETY: If any of the original files are found to have changed while the
inlined file is being edited, you will be warned on exiting the editor
and the affected files will be written to safe backup copies instead of
overwriting the originals.  This safety measure notwithstanding, it is
recommended that you manage suite definitions with a revision control
system so that (a) you can easily check that new changes are as you
expect, and (b) you can easily back out in case of disaster.

NOTE: in the inlined edit mode you must save and exit the editor
before your changes are applied to the actual suite definition!

The edit process is spawned in the foreground as follows:
  (usage 1/): $(G)EDITOR suite.rc
  (usage 2/): $(G)EDITOR <inlined-suite.rc-file in $TMPDIR>
$GEDITOR or $EDITOR, and $TMDPIR, 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 search
    cylc prep inline
    cylc prep validate

Arguments:
```

```
  SUITE   - registered GROUP:NAME of the target suite.

Options:
  -h, --help    show this help message and exit
  -i, --inline  Edit with include-files inlined as described above.
  --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.2.12 email-alert

```
USAGE: cylc email-alert EVENT SUITE TASKID MESSAGE

A cylc event hook script that sends an email alert. The arguments
are supplied automatically by cylc. To get an email alert whenever
a task in your suite fails, for example, do this:

# SUITE.RC
task failed hook script = cylc email-alert # globally or per task
[cylc local environment]
   MAIL_ADDRESS = foo@bar.baz.waz

See the Suite.rc Reference (Cylc User Guide) for details.
```

B.2.13 export

```
Usage: cylc [db] export [OPTIONS] SOURCE [TARGET]

Export private suite or group registrations to the central database. The central
database can be a reference to the owner's suite definition, or we can copy
the suite definition to [CentralDB]/owner/group/name/.

Arguments:
    SOURCE  - Private database suite registration GROUP:NAME
    TARGET  - Central database suite registration OWNER:GROUP:NAME

Options:
  -h, --help  show this help message and exit
  -c, --copy  Copy the suite definition directory to the central database.
```

B.2.14 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.2.15 gcylc

```
Note that you can invoke gcylc directly.
Usage: 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,--graph option you'll get the standard filtered treeview
suite control and monitoring interface; with it, the
newer dependency graph based interface.

NOTE: daemonize important suites with the POSIX nohup command:
    $ nohup gcylc [options] [SUITE] &

Arguments:
    [SUITE]                 registered GROUP:NAME of a 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.2.16 get-dir

```
Usage: cylc [db] get-dir 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
```

B.2.17 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, but you omit
cold start tasks with the '-w,--warmstart' option.  Specify the optional
initial and final cycle time arguments to override the suite.rc defaults.

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.
```

```
    * "Best Fit" and "Normal Size" - self-explanatory.

Arguments:
  SUITE   - suite registration [OWNER:]GROUP:NAME
  START   - (optional) Initial cycle time (YYYYMMDDHH) to plot.
  STOP    - (optional) Final cycle time (YYYYMMDDHH) to plot (default=START).


Options:
  -h, --help            show this help message and exit
  -w, --warmstart       Plot the raw start graph instead of cold start.
  -f FILE, --file=FILE  View a specific dot-language graphfile.
  -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.2.18 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                Registered GROUP:NAME of the 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.2.19 housekeeping

```
Usage: 1/ cylc [util] housekeeping [options] SOURCE MATCH OPER OFFSET [TARGET]
Usage: 2/ cylc [util] housekeeping [options] FILE

Parallel (as in spawning of multiple Unix processes at once) batch
archiving and cleanup of files or directories with names that contain a
cycle time.

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 $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 $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
                        (defaults to 1, i.e. sequential processing).
```

B.2.20 import

```
Usage: cylc [db] import [OPTIONS] SOURCE [TARGET] DIR

Import suites or groups from the central registration database.

Arguments:
    SOURCE    - Central database registration OWNER:GROUP:NAME or OWNER:GROUP:
    [TARGET]  - Private database registration GROUP:NAME or GROUP:
                (omit TARGET to keep the central GROUP:NAME or GROUP:)
    DIR     - destination for the imported suite definition (must not exist).
              or top level directory for imported groups.

Options:
  -h, --help  show this help message and exit
```

B.2.21 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 $TMDPIR, 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 search
    cylc prep edit
    cylc prep validate

Arguments:
  SUITE   - suite registration [OWNER:]GROUP:NAME
            (user OWNER: to access the central database)

Options:
  -h, --help    show this help message and exit
  -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.2.22 insert

```
Usage: cylc [control] insert [options] SUITE TASK[%STOP]

Insert a single task or a task group into a running suite. Task groups must be
defined in the suite config (suite.rc) file.

Inserted tasks will spawn successors as normal, unless they are 'one off' tasks.
See also 'cylc task submit', for running single tasks without a scheduler.

You must be the owner of the target suite to use this command.

arguments:
   SUITE                Registered GROUP:NAME of the target suite.
   TASK                 Target task or insertion group.
   STOP                 Optional stop tag (e.g. 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.2.23 list

```
Usage: cylc [info] list|ls [OPTIONS] SUITE

Parse a suite config file and print the configured task list.

Arguments:
   SUITE    - suite registration [OWNER:]GROUP:NAME
              (use OWNER: to access the central database)

Options:
  -h, --help   show this help message and exit
  -d, --debug  print the exception traceback for validation errors.
```

B.2.24 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:
#_________________________
# SUITE.RC
use lockserver = False
#----------------------

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:NAME%CYCLE
                        Acquire a task lock.
  --release-task=SUITE:NAME%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.2.25 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:
#_____________________
# SUITE.RC
use lockserver = False
#---------------------

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
                        /tmp/oliverh/19583/cylc-3.4.0-a/conf/lockserver.conf)
```

B.2.26 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 - Private suite database registration GROUP:NAME
          of the suite whose log you want to view.

Options:
  -h, --help            show this help message and exit
  -p, --print           Print the suite log file location and exit.
  -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.2.27 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                Registered GROUP:NAME of the 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.2.28 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.2.29 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 for small suites and, because it
is just a passive monitor that cannot intervene in a suites operation,
it is allowed to monitor suites owned by others and/or running on remote
hosts. See also 'cylc dump', and the GUI gcylc.

Arguments:
   SUITE                Registered GROUP:NAME of the 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.2.30 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                Registered GROUP:NAME of the 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.2.31 ping

```
Usage: cylc [discover] ping [options] SUITE

Check that SUITE is running.

Arguments:
SUITE             Target suite registration GROUP:NAME.

Arguments:
```

```
   SUITE                Registered GROUP:NAME of the 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.2.32 print

```
Usage: cylc [db] print [options]
     $ cylc [db] pr                     # print all
     $ cylc [db] pr [FILTER options]    # print some

Print private or central suite registrations, which associate a suite name
with a suite definition directory (stored in $HOME/.cylc/registrations).
Cylc commands target a particular suite using its registered name.

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').
  (c) Omission of an item filter is taken to mean "match any of item".

Options:
  -h, --help         show this help message and exit
  -o RE, --ofilt=RE  Owner filter Regular Expression.
  -g RE, --gfilt=RE  Group filter Regular Expression.
  -n RE, --nfilt=RE  Name filter Regular Expression.
  -c, --centraldb    Print suite registrations from the central database.
  -v, --verbose      Turn on verbose output.
```

B.2.33 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., but 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                Registered GROUP:NAME of the target suite.
   TASK                 Task 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.2.34 refresh

```
Usage: cylc [db] refresh|check [options]

Check the private or central database for invalid registrations (this
probably means a suite definition directory has been deleted or moved
manually) and update any suite titles that have changed since their
initial registration (for the central database this also updates the
titles of suites owned by others).

Options:
  -h, --help       show this help message and exit
  -c, --centraldb  Print suite registrations from the central database.
  -v, --verbose    Turn on verbose output.
```

B.2.35 register

```
Usage: cylc [db] register [options] SUITE PATH

Register a suite in your private suite database. Suite registrations
associate a GROUP and a NAME with a suite definition directory. This
must be done before you can use a suite - cylc commands reference a
particular suite by its GROUP:NAME.

Arguments:
     SUITE  -  Private suite database registration GROUP:NAME
     PATH   -  A cylc suite definition directory location.

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Turn on verbose output.
```

B.2.36 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                Registered GROUP:NAME of the 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.2.37 remove

```
Usage: cylc [control] remove|kill [options] SUITE TASK|TAG

Remove a task, or all tasks with a common TAG (cycle time or
asynchronous 'a:INT'), from a running suite.
See also 'cylc [control] purge'.

Target tasks will be forced to spawn successors before being removed, if
they have not done so already (unless: --no-spawn).

You must be the owner of the target suite to use this command.

arguments:
   SUITE                Registered GROUP:NAME of the target suite.
   TASK                 Target task.
   TAG                  Target cycle or asynchronous tag ('a:INTEGER').

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.2.38 reregister

```
Usage: cylc [db] reregister [options] FROM TO

Change the registration of a particular suite, or group of suites.

Arguments:
    FROM, TO  - suite registrations:
                  OWNER:GROUP:NAME (Central DB), or
                        GROUP:NAME (Private DB)
                or registration group (note trailing colon!):
                  OWNER:GROUP: (Central DB), or
                        GROUP: (Private DB)

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Turn on verbose output.
```

B.2.39 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.

See also:
   cylc [control] trigger
   cylc [control] hold
   cylc [control] release

You must be the owner of the target suite to use this command.

arguments:
   SUITE                Registered GROUP:NAME of the 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.2.40 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 &

See also:
    cylc [control] run
    cylc [control] stop|shutdown

Arguments:
   SUITE                Private database registration GROUP:NAME of the
                        suite you want to restart.
   [FILE]               Optional non-default state dump file (assumed to
                        be in the suite state dump directory unless you
                        supply the file 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.
  --pause-at=YYYYMMDDHH
                        Refrain from running tasks AFTER this cycle time.
  --paused              Pause immediately on starting to allow intervention in
                        the suite state before resuming operation.
  -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.2.41 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).

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.
  SUITE.RC: [special tasks] -> cold start = task1, task2, ...

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). The cold start task list must be defined in the suite config
file:
  SUITE.RC: [special tasks] -> cold start = task1, task2, ...

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. The cold start task list
must be defined in the suite config file:
  SUITE.RC: [special tasks] -> cold start = task1, task2, ...

If you provide an initial cycle time on the command line, the suite.rc
initial AND final cycle times, if set, will be ignored.

NOTE:
  SUITE.RC: [special tasks]-> startup = task1, task2, ...
Start up tasks are one off tasks that are used only in a cold start; any
dependence on a start up task is ignored in subsequent cycles. This
allows you to make everything wait on one or more start up tasks that
could, for example, clean out suite workspaces when a suite is started
from scratch (cold start).

NOTE: daemonize important suites with the POSIX nohup command:
  $ nohup cylc [con] run SUITE [START] > suite.out 2> suite.err &

See also:
    cylc [control] restart
    cylc [control] stop|shutdown

Arguments:
   SUITE                Private database registration GROUP:NAME of the
                        suite you want to run.
   [START]              (optional) Initial cycle time (YYYYMMDDHH). This will
                        override a start time set in the suite.rc file.

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.
  --pause-at=YYYYMMDDHH
                        Refrain from running tasks AFTER this cycle time.
  --paused              Pause immediately on starting to allow intervention in
                        the suite state before resuming operation.
  -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.2.42 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.2.43 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.

For case insenstive matching use '(?i)PATTERN'.

See also:
  cylc prep edit
  cylc prep inline

Arguments:
  SUITE   - Suite database registration [OWNER:]GROUP:NAME
            (user OWNER: to access the central database).
  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
```

B.2.44 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 or not their delayed
start time has been reached.

Arguments:
   SUITE                Registered GROUP:NAME of the 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.2.45 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.2.46 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.

See also:
    cylc [control] start|run
    cylc [control] restart

You must be the owner of the target suite to use this command.

arguments:
   SUITE                Registered GROUP:NAME of the 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.2.47 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_TIME, 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.2.48 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.2.49 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 CYCLE_TIME=2010082318 # (as from task execution environment)

   $ 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        Cycle time (YYYYMMDDHH) defaults to $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.2.50 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.2.51 test-suite

```
USAGE: cylc [admin] test-suite [--help]

Run an automated test of core cylc functionality using a new copy of
the 'examples:test' suite. 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.2.52 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.

See also:
   cylc [control] reset
   cylc [control] hold
   cylc [control] release

You must be the owner of the target suite to use this command.

arguments:
   SUITE                Registered GROUP:NAME of the 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.2.53 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 useful
when running important suites, or multiple suites at once.

(These commands are 'block/unblock' rather than 'lock/unlock' in order
to distinguish them from the functionality of the cylc lockserver
commands, to which they are unrelated).

You must be the owner of the target suite to use this command.

arguments:
   SUITE                Registered GROUP:NAME of the 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.2.54 unregister

```
Usage: cylc [db] unregister|delete [OPTIONS] SUITE

Remove suite (or suite group) registrations from your private, or the
central, suite database.  By default this does not actually delete
suite definition directories.

If you accidentally unregister a running suite, just reregister it under
the same GROUP:NAME name to regain access to it.

Arguments:
    SUITE - suite registration:
            OWNER:GROUP:NAME (Central DB), or
                  GROUP:NAME (Private DB), or
          or registration group (note trailing colon!):
            OWNER:GROUP: (Central DB), or
                  GROUP: (Private DB)

Options:
  -h, --help     show this help message and exit
  --obliterate   Delete the suite definition directory too (!DANGEROUS!).
  --force        Don't ask for confirmation before obliterating suite
                 definitions
  --gcylc        (DO NOT USE THIS OPTION).
  -v, --verbose  Turn on verbose output.
```

B.2.55 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 [OWNER:]GROUP:NAME
              (use OWNER: to access the central database).

Options:
  -h, --help   show this help message and exit
  -d, --debug  print the exception traceback for validation errors.
```

B.2.56 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                Registered GROUP:NAME of the 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.2.57 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 startup 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 startup 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 Post-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 and for each task, in a single structured, validated, config 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.

G Pyro

Pyro (Python Remote Objects) is a widely used open source objected oriented Remote Procedure Call technology, see *http://pyro.sourceforge.net*.

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), and David Matthews (Met Office UK), for discussion, bug finding, and many suggestions that have improved cylc's usability and functionality.

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`.