The Cylc Forecast Suite Metascheduler User Guide

4.5.0

Hilary Oliver

July 31, 2012

Abstract

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

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

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

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

Contents

List of Figures

1 Introduction: How Cylc Works

1.1 Scheduling Forecast Suites

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

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

1.2 EcoConnect

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

1.3 Dependence Between Tasks

1.3.1 Intracycle Dependence

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

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

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

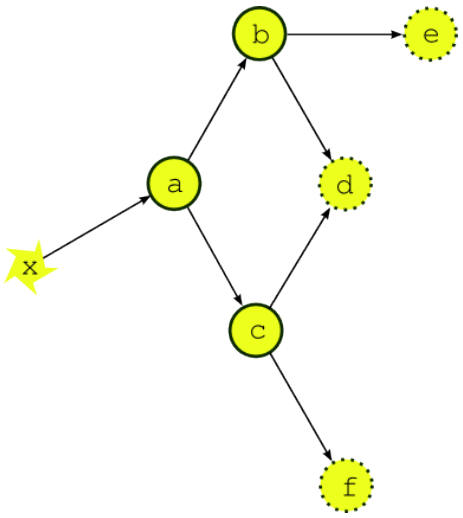

Figure 1: The dependency graph for a single forecast cycle of a simple example suite. Tasks *a, b,* and *c* represent forecast models, *d, e* and *f* are post processing or product generation tasks, and *x* represents external data that the upstream forecast model depends on.

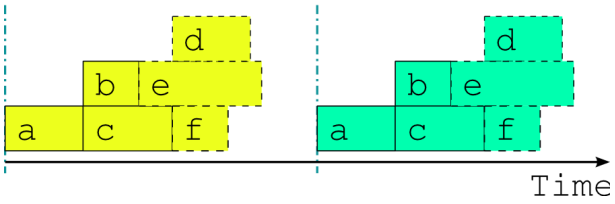

Figure 2: The optimal job schedule for two consecutive cycles of our example suite during real time operation, assuming that all tasks trigger off upstream tasks finishing completely. The horizontal extent of a task bar represents its execution time, and the vertical blue lines show when the external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

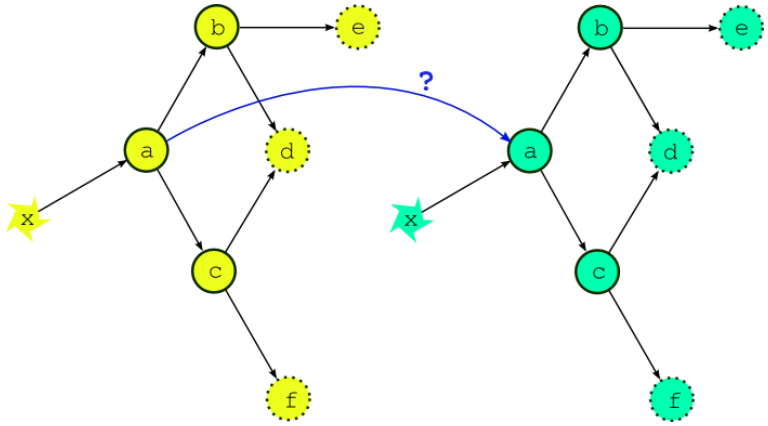

Figure 3: If the external driving data is available in advance, can we start running the next cycle early?

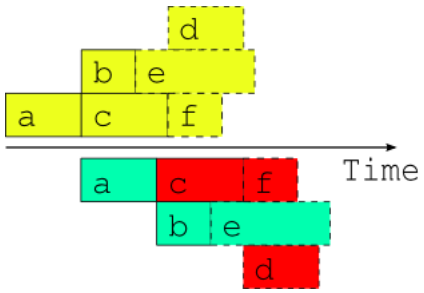

Figure 4: A naive attempt to overlap two consecutive cycles using the single-cycle dependency graph. The red shaded tasks will fail because of dependency violations (or will not be able to run because of upstream dependency violations).

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

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

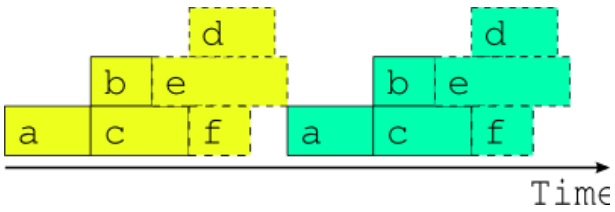

Figure 5: The best that can be done *in general* when intercycle dependence is ignored.

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

1.3.2 Intercycle Dependence

Forecast models typically depend on their own most recent previous forecast for background state or restart files of some kind, and different types of tasks in different forecast cycles can also be linked (in an atmospheric forecast analysis suite, for instance, the weather model may also generate background states for use by the observation processing and data-assimilation systems in the next cycle). In real time operation this intercycle dependence can be ignored because it is automatically satisfied when each cycle finishes before the next one begins. If, on the other hand, it is explicitly accounted for, it complicates the dependency graph by destroying the clean boundary between forecast cycles. Figure 6 illustrates the problem for our simple example suite assuming the minimal intercycle dependence: the forecast models (a, b, and c) each depend on their own previous instances.

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

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

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

Similarly, Figure 9 shows example suite job schedules for an historical case study, or when catching up after a very long delay; i.e. when the external driving data are available many cycles in advance. Task a, which as the most upstream forecast model is likely to be a resource intensive atmosphere or ocean model, has no upstream dependence on cotemporal tasks and can

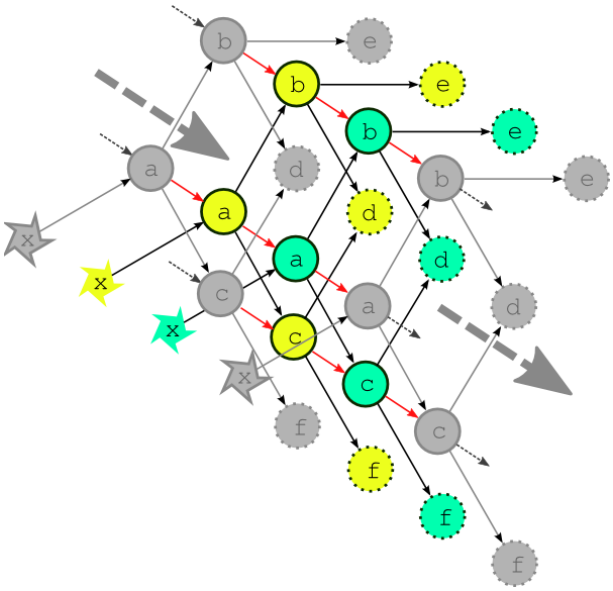

Figure 6: The complete dependency graph for the example suite, assuming the least possible intercycle dependence: the forecast models (a, b, and c) depend on their own previous instances. The dashed arrows show connections to previous and subsequent forecast cycles.

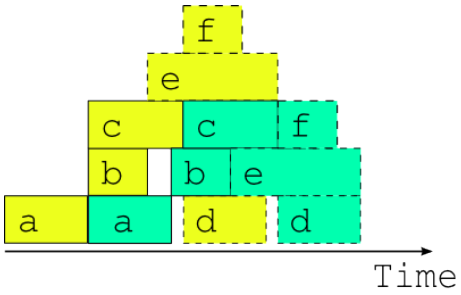

Figure 7: The optimal two cycle job schedule when the next cycle's driving data is available in advance, possible in principle when intercycle dependence is handled explicitly.

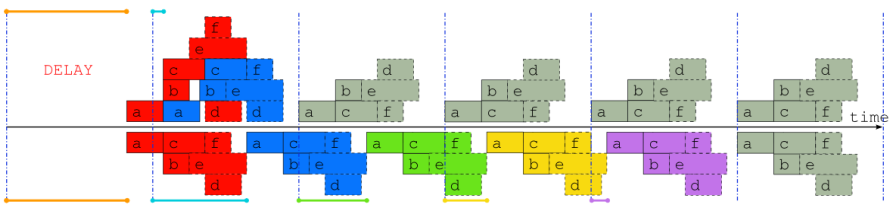

Figure 8: Job schedules for the example suite after a delay of almost one whole forecast cycle, when intercycle dependence is taken into account (above the time axis), and when it is not (below the time axis). The colored lines indicate the time that each cycle is delayed, and normal "caught up" cycles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

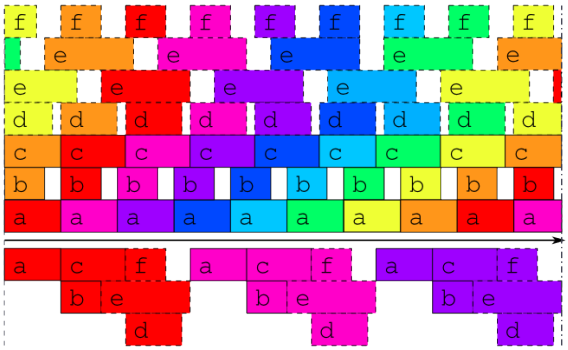

Figure 9: Job schedules for the example suite in case study mode, or after a long delay, when the external driving data are available many cycles in advance. Above the time axis is the optimal schedule obtained when the suite is constrained only by its true dependencies, as in Figure 3, and underneath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

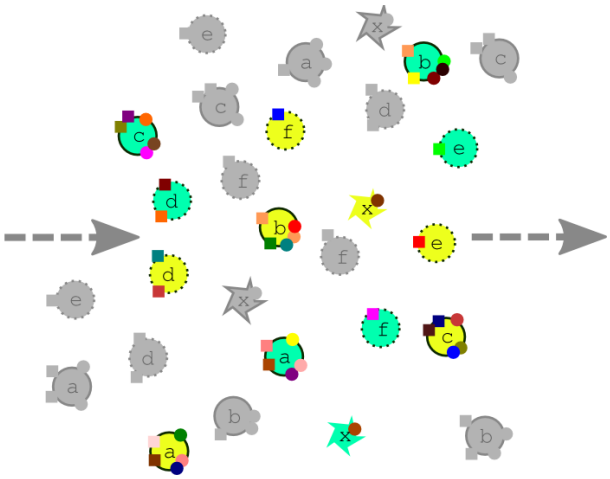

Figure 10: How cylc sees a suite, in contrast to the multicycle dependency graph of Figure 6. Task colors represent different cycle times, and the small squares and circles represent different prerequisites and outputs. A task can run when its prerequisites are satisfied by the outputs of other tasks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

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.

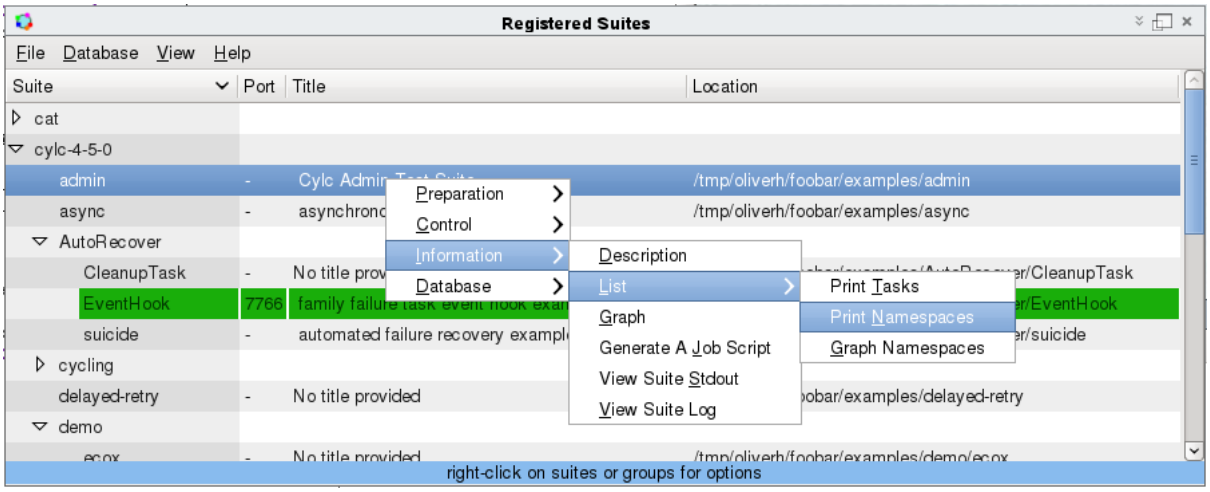

Figure 11: The gcylc GUI, showing one suite running on port 7766.

1.4 The Cylc Scheduling Algorithm

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

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

2 Cylc Screenshots

[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.

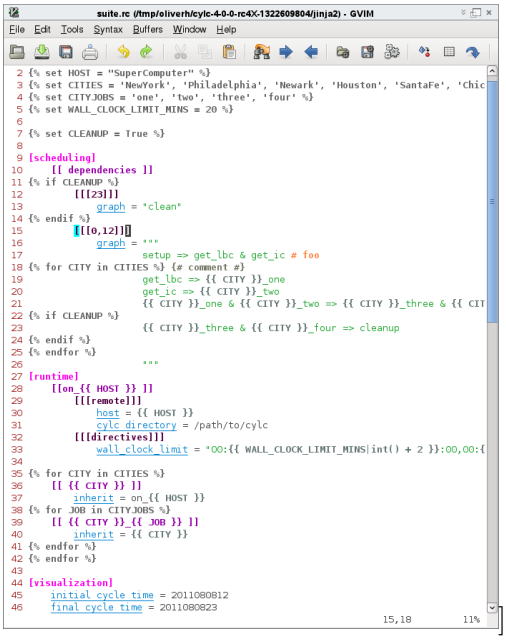

Figure 12: A cylc suite definition in the *vim* editor.

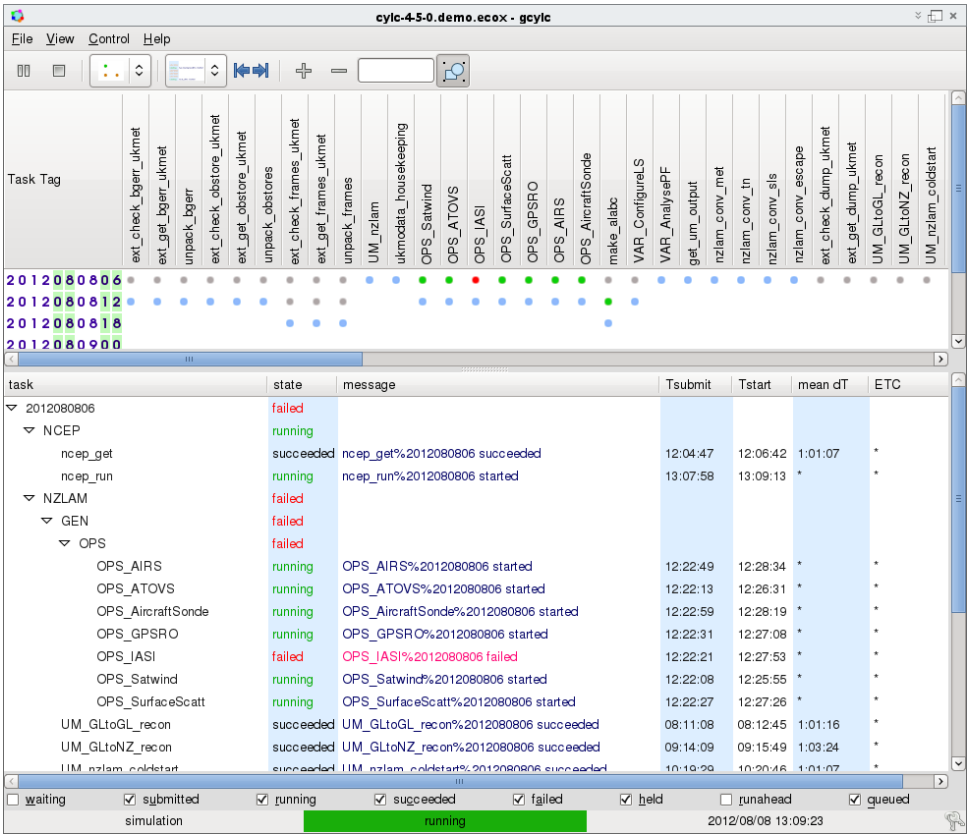

Figure 13: The gcontrol GUI, showing dot and text views.

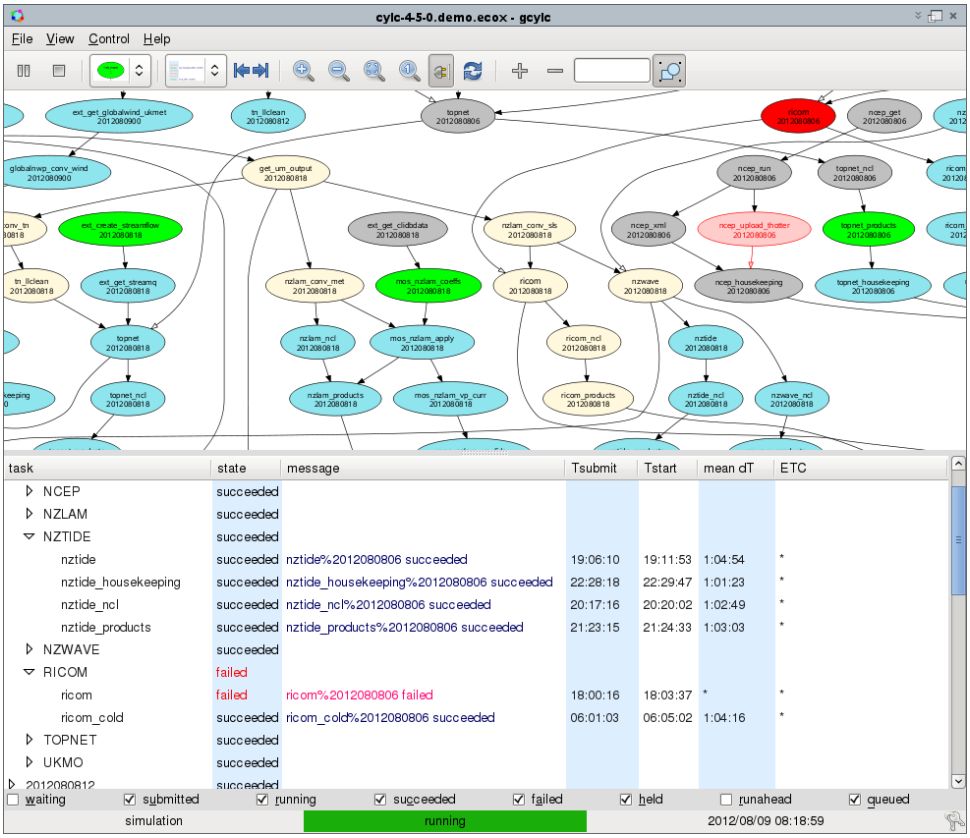

Figure 14: The gcontrol GUI, showing graph and text views.

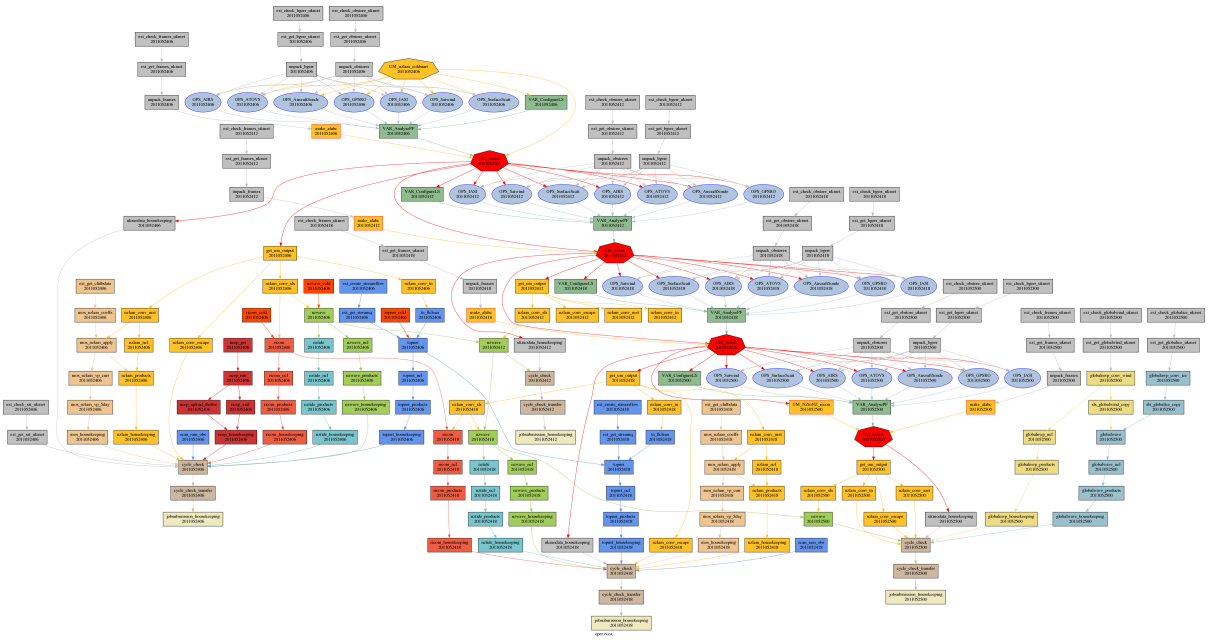

Figure 15: A large suite graphed by cylc.

3 Required Software

- **OS: Linux or Unix (including Mac OS X).** Cylc may assume Unix-style file paths in places, a tiny minority of cylc commands are Bash shell scripts (most are Python), and cylc-generated task job scripts are written for Bash.
- **The Python Language, v 2.4+, preferably 2.5+, latest tested 2.7.2**. Not Python 3.x yet; as of mid 2011 version 2.7 is the standard for new Linux distributions.
  http://python.org
- **Pyro (Python Remote Objects), version 3.10+, latest tested 3.16**; not Pyro 4.x as yet. Pyro is used by cylc for network communication between server processes (cylc suites) and client programs (running tasks, control GUIs and commands).
  http://irmen.home.xs4all.nl/pyro3
- **Cylc**, this version: 4.5.0 .
  http://hjoliver.github.com/cylc

The following packages are technically optional as you can construct and control cylc suites without dependency graphing, the cylc GUIs, and template processing:

- **PyGTK**, a Python wrapper for the GTK+ GUI toolkit, required for the cylc GUI (but you can prepare and control cylc suites entirely from the command line if you like). PyGTK is included in most Linux Distributions.
  http://www.pygtk.org
- **Graphviz** (latest tested 2.28.0) and **Pygraphviz** (latest tested 1.1), a graph layout engine and a Python interface to it. These are required for suite dependency graphing and the graph suite control GUI (but you can also run cylc without them).
  http://www.graphviz.org
  http://networkx.lanl.gov/pygraphviz
- **Jinja2**, a template processor for Python (latest tested: 2.6). Jinja2 adds variables, conditional and mathematical expressions, loop control structures, etc., to cylc suite definition files.
  http://jinja.pocoo.org/docs

3.1 Known Version Compatibility Issues

Cylc should run "out of the box" on recent Linux distributions.

For distributed suites the Pyro versions installed on all suite or task hosts must be mutually compatible. Using identical Pyro versions guarantees compatibility but may not be strictly necessary because cylc uses Pyro rather minimally.

Recent versions of Pyro require Python 2.5 or greater, due to use of the `with` statement introduced in 2.5.

3.1.0.1 Pyro 3.9 and Earlier

Beware of Linux distributions that come packaged with old Pyro versions. Pyro 3.9 and earlier is not compatible with the new-style Python classes used in cylc. It has been reported that Ubuntu 10.04 (Lucid Lynx), released in September 2009, suffers from this problem. And surprisingly so does Ubuntu 11.10 (Oneiric Ocelot), released in October 2011 - and therefore, presumably, all earlier Ubuntu releases. Attempting to run `cylc admin test-suite` with Pyro 3.9 or earlier results in the following Python traceback:

```
Traceback (most recent call last):
```



```
 File "/home/oliverh/cylc/bin/_run", line 232, in <module>
 server = start()
 File "/home/oliverh/cylc/bin/_run", line 92, in __init__
 scheduler.__init__( self )
 File "/home/oliverh/cylc/lib/cylc/scheduler.py", line 141, in
__init__
 self.load_tasks()
 File "/home/oliverh/cylc/bin/_run", line 141, in load_tasks_cold
 itask = self.config.get_task_proxy( name, tag, 'waiting',
stopctime=None, startup=True )
 File "/home/oliverh/cylc/lib/cylc/config.py", line 1252, in
get_task_proxy
 return self.taskdefs[name].get_task_class()( ctime, state,
stopctime, startup )
 File "/home/oliverh/cylc/lib/cylc/taskdef.py", line 453, in
tclass_init
 print '-', sself.__class__.__name__, sself.__class__.__bases_
AttributeError: type object 'A' has no attribute '_taskdef__bases_'
_run --debug testsuite.1322742021 2010010106 failed: 1
```

3.1.0.2 Apple Mac OSX

It has been reported that cylc runs fine on OSX 10.6 SnowLeopard, but on OSX 10.7 Lion there is an issue with constructing proper FQDNs (Fully Qualified Domain Names) that requires a change to the DNS service. Here's how to solve the problem:

- Edit `/System/Library/LaunchDaemons/com.apple.mDNSResponder.plist` by adding `<string>-AlwaysAppendSearchDomains</string>` after line 16:

```
<key>ProgramArguments</key>
  <array>
    <string>/usr/sbin/mDNSResponder</string>
    <string>-launchd</string>
        <string>-AlwaysAppendSearchDomains</string>
  </array>
```

- Now unload and reload the mDNSResponder service:

```
% sudo launchctl unload -w
/System/Library/LaunchDaemons/com.apple.mDNSResponder.plist
% sudo launchctl load -w
/System/Library/LaunchDaemons/com.apple.mDNSResponder.plist
```

3.2 Other Software Used Internally By Cylc

Cylc has absorbed the following in modified form (no need to install these separately):

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

4 Installation

4.1 Install The External Packages

First install Pyro, graphviz, Pygraphviz, and Jinja2 on your system, following the instructions provided with each package. If you do not have root access on your intended cylc host machine and cannot get a sysadmin to do this at system level, see Section 4.7 for local (user-specific) installation instructions.

4.2 Install The Cylc Release

Cylc typically installs into a normal user account; just unpack the release tarball in the desired location - which will be referred to below as `$CYLC_DIR`.

4.3 Configure Your Environment

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

```
#!/bin/bash

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

# These environment variables are required for interactive cylc usage.
# Access to cylc by running tasks is automatically configured by cylc.
#-------------------------------------------------------------------------
# Add the cylc bin directory to $PATH. The 'cylc' and 'gcylc' commands
# configure access to cylc sub-commands and python modules at run time.
export PATH=/path/to/cylc/bin:$PATH
#-------------------------------------------------------------------------
# For 'cylc edit' or gcylc -> Edit, set terminal and GUI editors:
export EDITOR=vim
export GEDITOR='gvim -f'
# (See 'cylc edit --help' for examples of other editors).
#-------------------------------------------------------------------------
# To access the Cylc User Guide via the gcylc GUI menus:
export PDF_READER=evince
export HTML_READER=firefox
# (The HTML guide is opened via file path, not http URL).
#-------------------------------------------------------------------------
# Some cylc commands require a writeable temporary directory. This is
# now determined automatically by Python's tempfile.mkdtemp() which is
# likely to use $TMPDIR if it is defined. You can override this if
# necessary by setting the environment variable CYLC_TMPDIR:
  #==># export CYLC_TMPDIR=$TMPDIR/cylc
  #==># mkdir -p $CYLC_TMPDIR
# (Use of plain TMPDIR reportedly causes problems under KDE desktop)
# In the default (non-CYLC_TMPDIR) case the directory will be cleaned up
# when the relevant cylc command exits.
#-------------------------------------------------------------------------
# For a local user install of one or more of Pyro, Graphviz, Pygraphviz,
# and Jinja2 (if you can't easily get them installed at system level on
# the cylc host) follow Cylc User Guide Installation instructions on how
# to install them locally, and modify your PYTHONPATH accordingly, e.g.:
## PYTHONPATH=$HOME/external/lib64/python2.6/site-packages:$PYTHONPATH
## export PYTHONPATH
```

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

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

4.4 Import The Example Suites

Run the following command immediately after installation to create your suite database and import the example suites to it (this involves copying the suite definitions to sub-directories of the given destination directory, and registering each one for use in your suite database:

```
% cylc admin import-examples TOPDIR
```

Where `TOPDIR` is the top level directory into which the example suite definitions will be copied. To view the content of the resulting suite database, run gcylc or use the `cylc db print` command:


```
% cylc db print --tree -x 'Auto|Quick'
 cylc-x-y-z
   |-AutoCleanup
   | |-FamilyFailHook  family failure hook script example
   | `-FamilyFailTask  family failure cleanup task example
   |-AutoRecover
   | |-async           asynchronous automated failure recovery example
   | `-cycling         cycling automated failure recovery example
   `-QuickStart
     |-a          Quick Start Example A
     |-b          Quick Start Example B
     |-c          Quick Start Example C
     `-z          Quick Start Example Z
```

Note that the dots in the cylc release version number are replaced with hyphens because '.' is the registration name delimiter. Type `cylc db print --help` to see what the command options mean.

4.5 Automated Database Test

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

4.6 Automated Scheduler Test

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

4.7 Complete Non-System-Level Installation

If you do not have root access to your host machine and cannot easily get Pyro, graphviz, Pygraphviz, and Jinja2 installed at system level, here's how to install everything under your home directory.

First, cylc is already designed to be installed into a normal user account - just unpack the release tarball into `$CYLC_DIR`. If you invoke cylc commands at this stage you will get a warning that Pyro is not installed.

Next, create a new sub-directory in the cylc source tree, `$CYLC_DIR/external`, and download the Pyro, Graphviz, and Pygraphviz source distributions to it (the URLs are given at the beginning of Section 3).

4.7.1 Pyro

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

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

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

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


The exact path will depend on your local Python environment. Configure your login script for the cylc environment as described in Section 4.3 above and add in this new installation path; for example:

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

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

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

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

Note that Pyro can also be installed with `easy_install`*, which downloads and installs python packages in one shot.*

At this point you should have access to all cylc functionality except for suite graphing and the graph control GUI view. For example

4.7.2 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 --with-qt=no
% make
% make install
```

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

Note that the graphviz build may fail on systems that do not have QT installed, hence the `./configure --with-qt=no` option above.

4.7.3 Pygraphviz

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

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

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

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

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

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

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


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

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

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

4.7.4 Jinja2

You can download Jinja2 from the project web site and install it with:

```
python setup.py install --prefix=/path/to/install/location
```

or use the `easy_install` command to do it all in one step. Either way the final installed package location must be present in the `PYTHONPATH` variable, and you may have to arrange for this first. You may also need to create the installed package directory if it doesn't exist already (if so the install will abort and print the name of the missing directory in the error message). Here's how to easy_install Jinja2 into your new private python site packages directory:

```
% LOCALPREFIX=$CYLC_DIR/external/installed
% LOCALPACKAGES=$CYLC_DIR/external/installed/lib64/python2.6/site-packages
% export PYTHONPATH=$LOCALPACKAGES:$PYTHONPATH
% easy_install --prefix=$LOCALPREFIX Jinja2
```

Adapt the site-packages path according to your actual path, as above.

4.8 What Next?

You should now have access to all cylc functionality. Import the example suites if you have not done so already (Section 4.4) then test your cylc installation by running the automated suite database test (Section 4.5) and the automated scheduler test (Section 4.6), then go on to the *Quick Start Guide* (Section 6).

4.9 Upgrading To New Cylc Versions

Upgrading is just a matter of unpacking the new cylc release and optionally re-importing the example suites for the new version.

4.10 Cylc Version Re-invocation (Pseudo Backward Compatibility)

It may not be convenient for users to upgrade all of their suites at once when a new cylc release is installed. As of cylc-4.2.0 any installed version of cylc can act as a transparent front end to any other installed version that is installed in parallel (i.e. in the same location) with the invoked version. Technically this also provides pseudo forward compatibility. New-version suites can therefore be worked on at the same time as older suites that have yet to be upgraded, without the user reconfiguring his environment or having to explicitly invoke different cylc versions for

different suites. Since `cylc-4.5.0` this applies to commands that connect to running suites as well as those that parse suite definitions.

The required cylc version must be specified on the *first line* of a non-Jinja2 suite definition:

```
#!cylc-4.2.0
# suite definition follows ...
```

or on the *second line* for Jinja2 suites:

```
#!Jinja2
#!cylc-4.2.0
# suite definition follows ...
```

This results in the right cylc version being used for any suite-referencing command, e.g.:

```
% cylc validate foo
------------------------------------------------------------
Cylc version re-invocation:
  Invoked version: cylc-4.2.0 (/home/oliverh/cylc-4.2.0)
  Suite requires:  cylc-4.2.1
(assuming parallel cylc installations)
=> Re-issuing command using /home/oliverh/cylc-4.2.1
------------------------------------------------------------
Suite foo is valid for cylc-4.2.1
```

Note that non suite-referencing commands necessarily execute exactly as invoked. This includes all command line help so it is advisable, although not strictly necessary, to keep your login environment configured for the latest cylc version.

If the required cylc version is not installed the command will abort and you will have to upgrade the suite definition for compatibility with an installed version: use `cylc validate` and the *Suite.rc Reference* (Appendix A) to identify problems, and consult the cylc change log to see what changed between versions.

4.10.1 Explicit Cylc Version Reinvocation

Other cylc versions can be specified explicitly on the command line, `cylc -V x.y.z` (capital V). This feature works for versions 4.5.0 and later.

4.10.2 Available Cylc Versions

To list available cylc versions installed in parallel with the invoked version: `cylc -V avail` (capital V).

4.10.3 Limitations

The top level command interface, `$CYLC_DIR/bin/cylc`, determines the legality of requested sub-command names so any attempt to run a sub-command that no longer exists in the invoked version of cylc will fail. Additionally, the command line must be parsed first by the sub-command of the invoked cylc version in order to determine which command line argument references the suite (the suite name is always the first non-option argument but only sub-commands know if particular options have their own arguments or are just logical flags, and can therefore determine which is the first true argument); consequently any attempt to use a command option that is illegal in the invoked version of cylc will also fail.

4.10.4 A Usage Suggestion

You can use the Jinja2 template processor (see Section 8.6) to store the required cylc version in a single variable that can also be used elsewhere in the suite definition - to specify the cylc version to use on a remote task host, for example:


```
#!Jinja2
{% set CYLC_VERSION = "cylc-4.2.2" %}
#!{{CYLC_VERSION}}

#...
[runtime]
   [[on_hpc]]
       [[[remote]]]
           cylc directory = /path/to/cylc/top/dir/{{CYLC_VERSION}}
```

5 On The Meaning Of *Cycle Time* In Cylc

From using other schedulers you may be accustomed to the idea that a forecasting suite has a "current cycle time", which is typically the analysis time or nominal start time of the main forecast model(s) in the suite, and that the whole suite advances to the next forecast cycle when all tasks in the current cycle have finished (or even when a particular wall clock time is reached, in real time operation). As is explained in the Introduction, this is not how cylc works.

Cylc suites advance by means of individual tasks with private cycle times independently spawning successors at the next valid cycle time for the task, not by incrementing a suite-wide forecast cycle. Each task will be submitted when its own prerequisites are satisfied, regardless of other tasks with other cycle times running, or not, at the time. It may still be convenient at times, however, 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. But keep in mind that the members of these groups may not be present simultaneously in the running suite - i.e. different tasks may pass through the "current cycle" (etc.) at different times as the suite evolves, particularly in delayed (catch up) operation.

6 Quick Start Guide

This section works through some basic cylc functionality using the "QuickStart" example suites, which you can import to your suite database by running the `cylc admin import-examples` command:

```
% cylc admin import-examples $HOME/examples
% cylc db print --tree QuickStart
 cylc-x-y-z
   `-QuickStart
     |-a        Quick Start Example A | ~/examples/cylc-x-y-z/QuickStart/a
     |-b        Quick Start Example B | ~/examples/cylc-x-y-z/QuickStart/b
     |-c        Quick Start Example C | ~/examples/cylc-x-y-z/QuickStart/c
     `-z        Quick Start Example Z | ~/examples/cylc-x-y-z/QuickStart/z
```

6.1 Configure Your Environment

To get access to cylc you just need to set a few environment variables in your login script, as described in Section 4.3.

6.2 View The QuickStart.a Suite Definition

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

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


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

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

Suites that use include-files can optionally be edited in a temporarily inlined state - the inlined file gets split back into its constituent include-files when you save it and exit the editor. While editing, the inlined file becomes the official suite definition so that changes take effect whenever you save the file.

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

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

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

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

(Cylc comes with syntax highlighting and section folding for the *vim* editor - see Section 8.2.3).

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

All the tasks in this suite can run in parallel with their own previous instances if the opportunity arises (i.e. if their prerequisites are satisfied before the previous instance is finished). Most tasks should be capable of this (see Section 14.4) but if necessary you can force particular tasks to run sequentially like this:

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

Finally, when the suite is *cold-started* (started from scratch) it is made to wait on a special *synchronous start-up task* called Prep. Start-up tasks are one-off (non-spawning) tasks that are only used at suite start-up, and any dependence on them only applies at suite start-up. They cannot be used in conditional trigger expressions with normal cycling tasks, because the trigger becomes undefined in subsequent cycles. Start-up tasks are *synchronous* because they have a

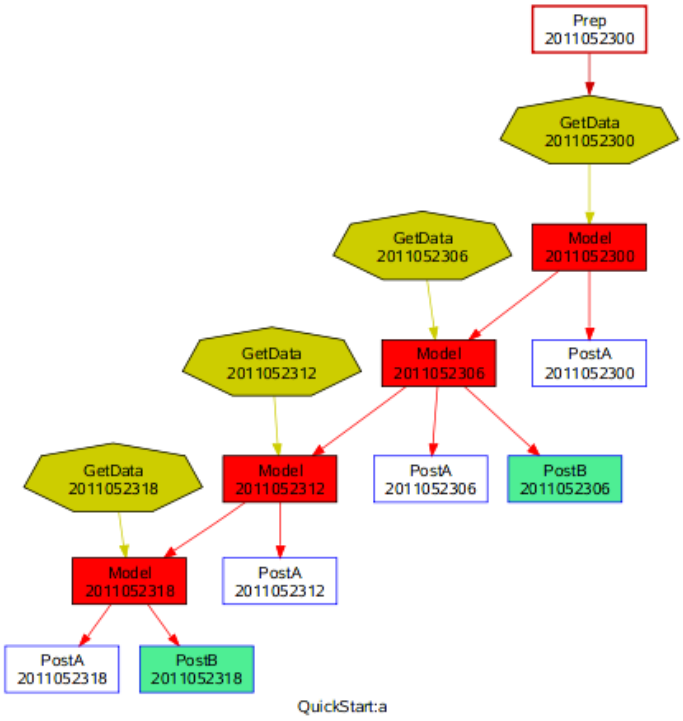

Figure 16: The *QuickStart.a* dependency graph, plotted by cylc.

defined cycle time even though they are not cycling tasks. Cylc also has *asynchronous one-off tasks*, which have no cycle time:

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

The optional visualization section configures graph plotting.

6.3 Plotting The QuickStart.a Dependency Graph

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

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

This will pop up a zoomable, pannable, graph viewer showing the graph of Figure 16. If you edit the suite.rc file the viewer will update in real time whenever you save the file.

6.4 Run The QuickStart.a Suite

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

```
% cylc get-config QuickStart.a runtime GetData 'command scripting'
['echo Dummy command scripting; sleep $CYLC_TASK_DUMMY_RUN_LENGTH']
```


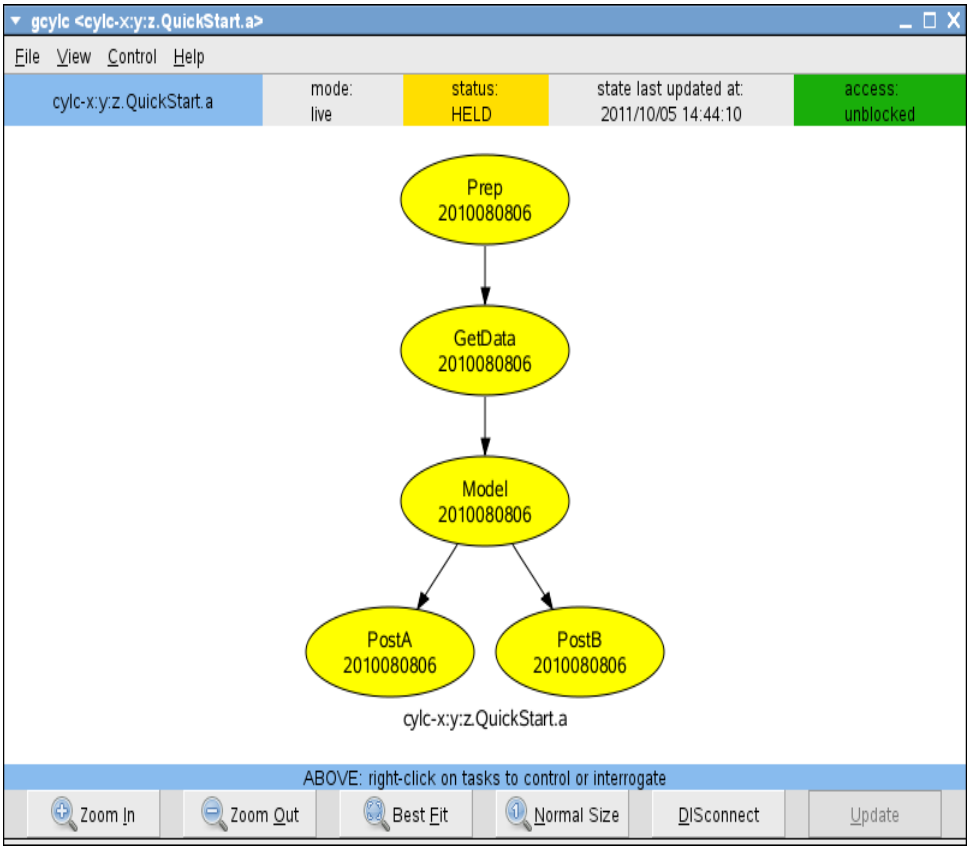

Figure 17: Suite *QuickStart:one* at start-up with an initial cycle time ending in 06 or 18 hours. Yellow nodes represent waiting tasks in the held state.

where `$CYLC_TASK_DUMMY_RUN_LENGTH` defaults to 10 seconds but can be overridden in task or family environments in the suite definition; this is exported to the task execution environment by cylc. The command arguments above reflect suite definition section nesting.

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

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

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

If the initial cycle time ends in 06 or 18 the suite controller should look like Figure 17, or otherwise (00 or 12) like Figure 18.

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

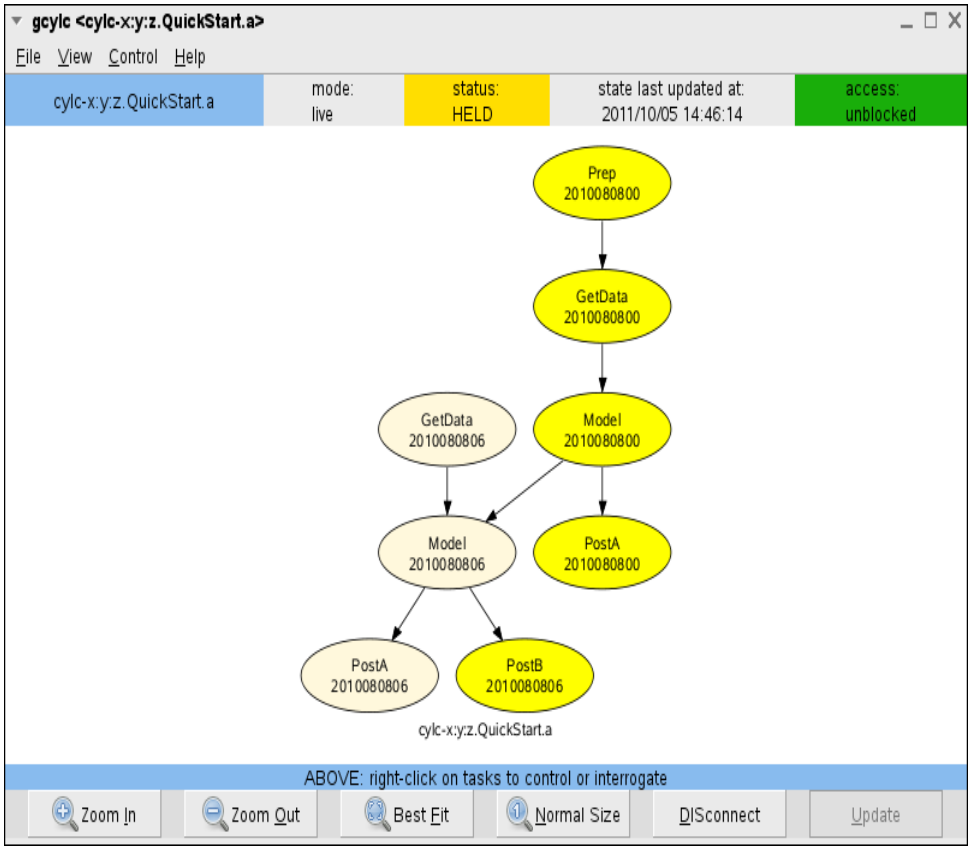

Figure 18: Suite *QuickStart.a* at start-up with an initial cycle time ending in 00 or 12 hours. Yellow nodes represent waiting tasks in the held state and off-white nodes are tasks from the base graph, defined in the suite.rc file, that aren't currently live in the suite.

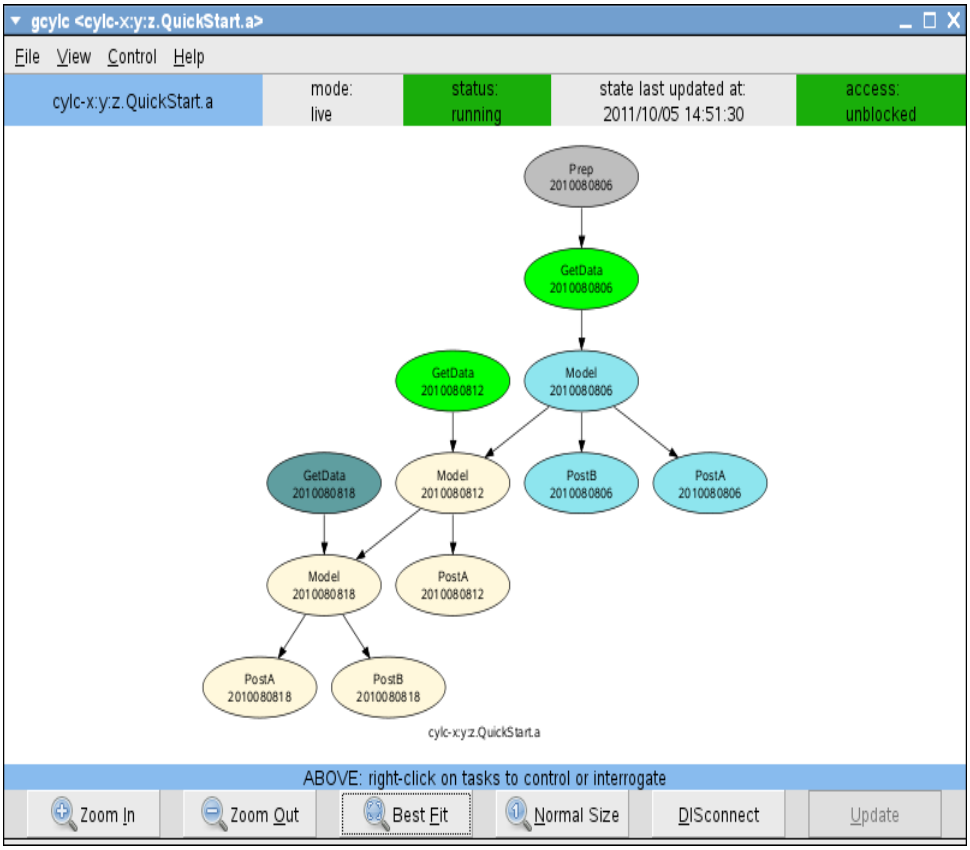

Figure 19: Suite *QuickStart.a* running, showing several consecutive instances of the clock-triggered GetData task running at once, out to the suite runahead limit of 12 hours.

Now, click on Control → Release in the suite control GUI to *release the hold on the suite*, and observe what happens: the GetData tasks will rapidly go off in parallel out to a few cycles ahead (how far ahead depends on the suite runahead limit as explained below and in *The Suite Runahead Limit*, Section 11.3.1) and then the suite will stall, as shown in Figures 19 and 20.

The Prep task runs immediately because it has no prerequisites and is not clock-triggered. The clock-triggered GetData tasks then all go off at once because they have no prerequisites (i.e. they do not have to wait on any upstream tasks), their trigger time has long passed (the initial cycle time was in the past), and they are not sequential tasks (so they are able to run in parallel - try declaring GetData sequential to see the difference). Beyond the suite *runahead limit* of 12 hours (set in the suite.rc file), however, GetData is put into a special 'runahead' held state indicated by the darker blue graph node. The task will be released from this state once the slower tasks in the suite have caught up sufficiently. The runahead limit is designed to stop free tasks like this from running off too far into the future in delayed operation. It is of little conseqence in real time operation[6] because clock triggered tasks are then constrained by the wall clock, and other tasks have to wait on them, generally speaking. See Section 11.3.1 for more on this.

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

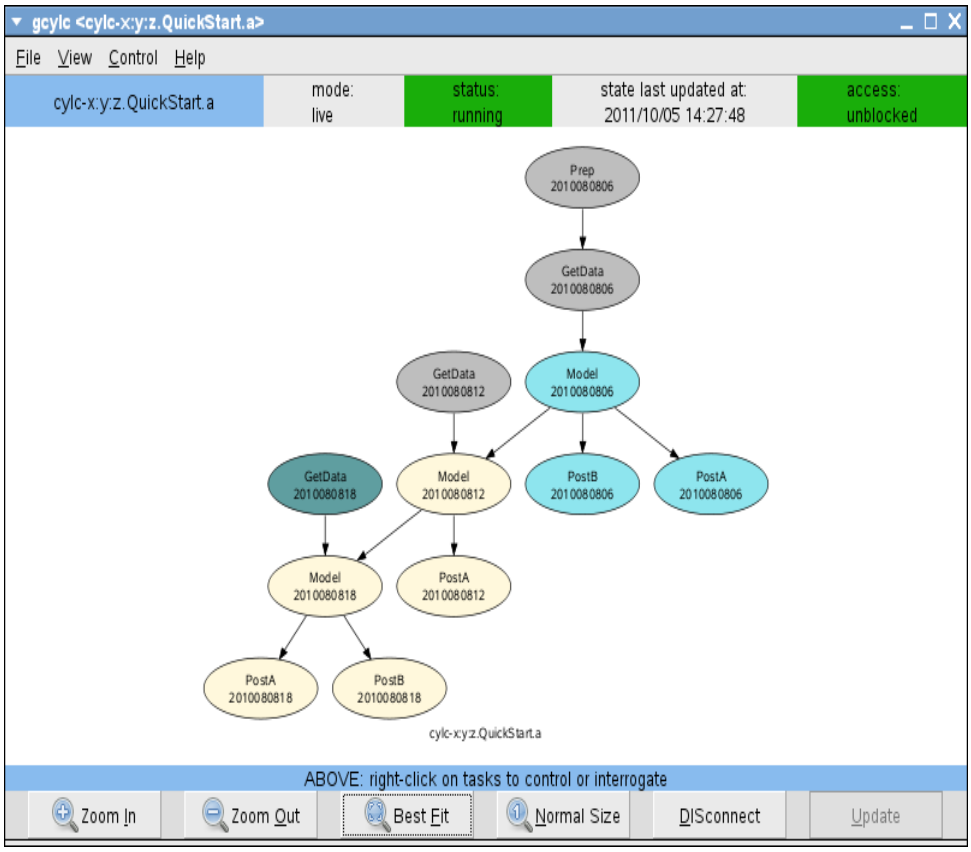

Figure 20: Suite *QuickStart.a* stalled after the clock-triggered GetData tasks have finished, because of Model's previous-cycle dependence and the suite runahead limit (see main text).

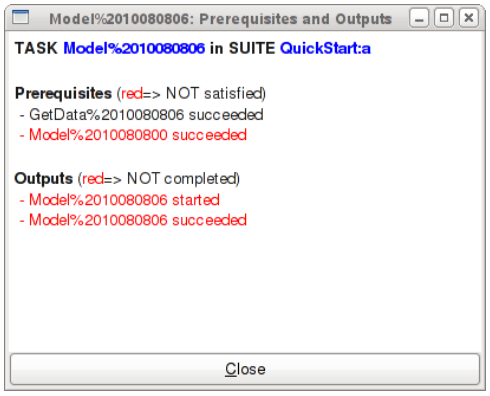

Figure 21: Viewing current task state after right-clicking on a task in gcontrol. The same information is available from the `cylc show` command.

6.4.1 Viewing The State Of Tasks

If you're wondering why a particular task has not triggered yet in a running suite you can view the current state of its prerequisites by right-clicking on the task and choosing 'View State', or using `cylc show`. Do this for the first Model task, which appears to be stuck in the waiting state; it will pop up a small window as in Figure 21.

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

6.4.2 Triggering Tasks Manually

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

6.4.3 Suite Shut-Down And Restart

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

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

6.5 QuickStart.b - Handling Cold-Starts Properly

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

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

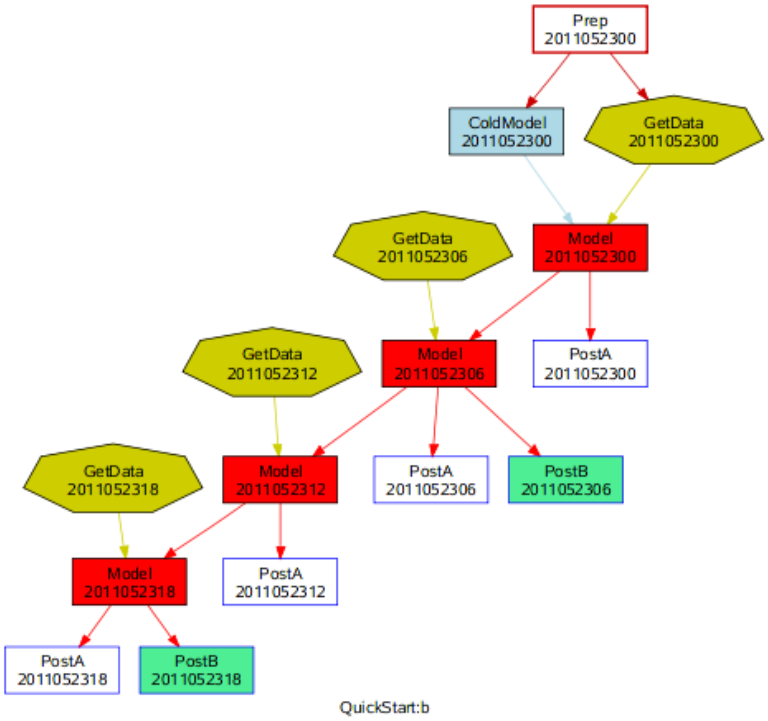

Figure 22: The *QuickStart.b* dependency graph showing a model cold start task.

```
        cold-start = ColdModel
```

and the dependency graph (see also Figure 22) looks like this:

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

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

Cold-start tasks are one-off tasks used in the first cycle to satisfy another task's intercycle-cycle dependence at suite start-up (when there is no previous cycle to do it). For instance, a series of cold-start tasks may be used to cold-start a warm-cycled model. Unlike *start-up* tasks though, cold-start dependence is preserved in subsequent cycles, so they must generally appear in OR'd conditional triggers in order to avoid stalling the suite after the first cycle (as in this example). This means cold-start tasks can be inserted into a running suite, if necessary, to cold-start their associated tasks in case of problems that prevent continued normal warm cycling.

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

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

6.6 QuickStart.c - Real Task Implementations

The suite *QuickStart.c* is the same as *QuickStart.b* except that it has real task implementations (scripts located in the suite bin directory) that generate and consume files in such a way that they have to run according to the graph of Figure 22. The suite gets them to run together out of a common I/O workspace, configured via the suite.rc file.

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

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

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

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

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

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

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

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



```
        command scripting = Model.sh

    [[Post]]
        description = "post processing for model"
        [[[environment]]]
            INPUT_DIR  = $WORKSPACE
            OUTPUT_DIR = $WORKSPACE
    [[PostA,PostB]]
        inherit = Post
        command scripting = <TASK>.sh

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

Here's the namespace hierarchy defined by this suite:

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

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

```
#!/bin/bash

set -e

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

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

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

sleep $TASK_EXE_SECONDS

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

6.7 Monitoring Running Suites

6.7.1 Suite stdout And stderr

When cylc runs a suite it writes warnings and other informative messages, such as when and how each task is submitted, to the stdout stream. If you start a suite at the command line you can direct this output wherever you like. If you start a suite via gcylc, however, the output

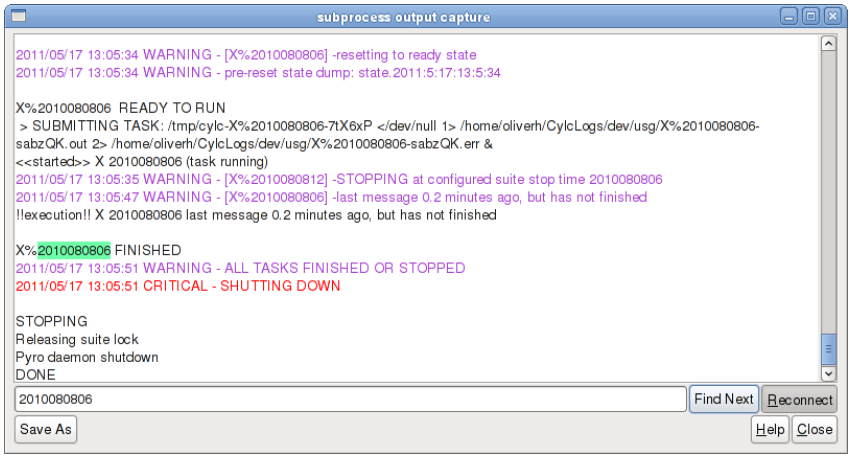

Figure 23: Cylc suite stdout/stderr example.

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.

6.7.2 Suite Logs

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

Figure 24 shows a suite log viewed from within gcylc. The `cylc log` command also enables viewing and filtering of suite logs without having to remember the actual log file location. But if you want to know the location:

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

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

6.7.3 Task stdout And stderr Logs

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

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

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

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

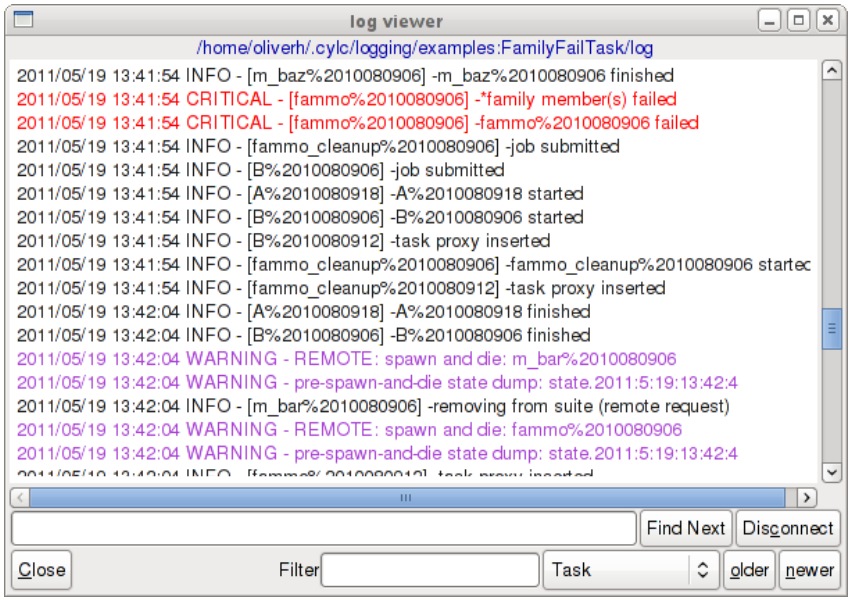

Figure 24: A cylc suite log viewed via gcylc.

6.8 Searching A Suite

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

```
% cylc grep OUTPUT_DIR QuickStart.c

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

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

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

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

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

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

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

6.9 Comparing Suites

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

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

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

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

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

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

6.10 Validating A Suite

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

```
% cylc validate -v foo.bar
Parsing Suite Definition
LOADING suite.rc
VALIDATING against the suite.rc specification.
PARSING clock-triggered tasks
PARSING runtime generator expressions
PARSING runtime hierarchies
PARSING SUITE GRAPH
Instantiating Task Proxies:
root
 |-GEN
 | |-OPS
 | | |-aircraft    ... OK
 | | |-atovs       ... OK
 | | `-atovs_post  ... OK
 | `-VAR
 |   |-AnPF        ... OK
 |   `-ConLS       ... OK
 |-baz
 | |-bar1          ... OK
 | `-bar2          ... OK
 |-foo             ... OK
 `-prepobs         ... OK
Suite foo.bar is valid for cylc-4.2.0
```


For more information on suite validation see Section 8.2.5.

6.11 Other Example Suites

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

Cylc has example suites intended to illustrate most facets of suite construction. These are held centrally under `$CYLC_DIR/examples` and can be imported to your suite database by running 'cylc admin import-examples'. They all run "out the box" and can be copied and modified by users to test almost anything. Some of them just configure a suite dependency graph, in which case cylc will run dummy tasks according to the graph; some also configure task runtime properties (e.g. command scripting and environment variables) within the suite definition; and some have real task implementations that generate and consume real files and which will fail if they are not executed in the right order. All of the example suites are portable in the sense that all suite and task I/O directory paths incorporate the suite registration name (this is the default situation for any cylc suite in fact) so you can run multiple copies of the same suite at once without any interference between them.

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

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

(This suite is explained in the *Quick Start Guide*, Section 6).

6.11.1 Choosing The Initial Cycle Time

When running any suite in live mode (or suites that depend on clock-triggered tasks at least) do not give an initial cycle time in the future unless you want nothing to happen until that time. However, you can also run any suite in *simulation mode* in which case a future start time is fine (see Appendix E).

7 Suite Registration

Cylc commands target particular suites via names registered in a *suite database*, so that you don't need to remember and continually refer to the actual location of the suite definition on disk. A suite registration name is a hierarchical name akin to a directory path but delimited by the '.' character; this allows suites to be organised in nested tree-like structures:

```
% cylc db print -t nwp
nwp
 |-oper
 | |-region1  Local Model Region1       /oper/nwp/suite_defs/LocalModel/nested/Region1
```



```
 |  `-region2  Local Model Region2       /oper/nwp/suite_defs/LocalModel/nested/Region2
 `-test
   `-region1  Local Model TEST Region1  /home/oliverh/nwp_suites/Regional/TESTING/Region1
```

Note that registration groups are entirely virtual, they do not need to be explicitly created before use, and they automatically disappear if all tasks are removed from them. From the listing above, for example, to move the suite `nwp.oper.region2` into the `nwp.test` group:

```
% cylc db rereg nwp.oper.region2 nwp.test.region2
REREGISTER nwp.oper.region2 to nwp.test.region2
% cylc db print -tx nwp
nwp
 |-oper
 |  `-region1  Local Model Region1
 `-test
   |-region1  Local Model TEST Region1
   `-region2  Local Model Region2
```

And to move `nwp.test.region2` into a new group `nwp.para`:

```
% cylc db rereg nwp.test.region2 nwp.para.region2
REREGISTER nwp.test.region2 to nwp.para.region2
% cylc db print -tx nwp
nwp
 |-oper
 |  `-region1  Local Model Region1
 |-test
 |  `-region1  Local Model TEST Region1
 `-para
   `-region2  Local Model Region2
```

Currently you cannot explicitly indicate a group name on the command line by appending a dot character. Rather, in database operations such as copy, reregister, or unregister, the identity of the source item (group or suite) is inferred from the content of the database; and if the source item is a group, so must the target be a group (or it will be, in the case of an item that will be created by the operation). This means that you cannot copy a single suite into a group that does not exist yet unless you specify the entire target registration name.

`cylc db register --help` shows a number of other examples.

7.0.2 Suite Passphrases

Any client process that connects to a running suite (this includes task messaging and user-invoked interrogation and control commands) must authenticate with a secure passphrase that has been loaded by the suite. A random passphrase is generated automatically in the suite definition directory at registration time, if one does not already exist there. For the default Pyro-based connection method the passphrase file must be distributed to any other accounts that host running tasks or from which you need monitoring or control access to the running suite. Alternatively, an ssh-based communication method can be used to automatically re-invoke cylc commands, including task messaging, on the suite host, in which case the suite passphrase is only needed on the suite host. See Section 11.2 for more on how cylc's client/server communication works and how to use it.

7.1 Suite Databases

Each user has a suite database that associates registered suite names with their respective suite definition directory locations. Hierarchical suite names stored in the database can be displayed in a tree structure. By right-clicking on a suite in your database, from within gcylc, or using cylc commands, you can:

1. start a suite control GUI to run the suite (or connect to a running suite),

2. submit a single task to run, just as it would be submitted by its suite
3. view the suite stdout and stderr streams,
4. view the suite log (which records all events and messages from tasks),
5. retrieve the suite description,
6. list tasks in the suite,
7. view the suite namespace hierarchy,
8. edit the suite definition in your editor,
9. plot the suite dependency graph,
10. search the suite definition and bin scripts,
11. validate the suite definition,
12. copy the suite or group,
13. alias the suite name to another name,
14. compare (difference) the suite with another suite,
15. unreregister the suite or group,
16. 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.

The default user suite database file is `$HOME/.cylc/DB`.

7.2 Database Operations

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

```
CATEGORY: db|database - Suite registration, copying, deletion, etc.

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

COMMANDS:
  alias ............... Register an alternative name for a suite
  copy|cp ............. Copy a suite or a group of suites
  get-directory ....... Retrieve suite definition directory paths
  gui|gcylc ........... Main Graphical User Interface (a.k.a. gcylc)
  print ............... Print registered suites
  refresh ............. Report invalid registrations and update suite titles
  register ............ Register a suite for use
  reregister|rename ... Change the name of a suite
  unregister .......... Unregister and optionally delete suites
```

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

```
% cylc copy foo.bar.baz boo $HOME/suites
```

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

```
% cylc copy foo.bar boo $HOME/suites
```

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

```
% cylc copy foo boo $HOME/suites
```


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

The same functionality is also available by right-clicking on suites or suite groups in the gcylc GUI, as shown in Figure 11.

8 Suite Definition

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

8.1 Suite Definition Directories

A cylc *suite definition directory* contains:

- **A suite.rc file**: this is the suite definition.
  - And any include-files used in it (see below; may be kept in sub-directories).
- **A `bin/` sub-directory**.
  - *Optional.*
  - For scripts and executables that implement, or are used by, suite tasks.
  - Automatically added to `$PATH` in task execution environments.
  - Alternatively, tasks can call external commands, scripts, or programs; or they can be scripted entirely within the suite.rc file.
- **A `python/` sub-directory**.
  - *Optional.*
  - For user-defined job-submission methods, if needed (see Section 10).
  - Alternatively, new job submission methods can be installed into the cylc source tree, or in any directory in your `$PYTHONPATH`.
- **Any other sub-directories and files** - documentation, control files, etc.
  - *Optional.*
  - Holding everything in one place makes proper suite revision control possible.
  - Portable access to files here, for running tasks, is provided through `$CYLC_SUITE_DEF_PATH` (see Section 8.4.4).
  - Ignored by cylc, but the entire suite definition directory tree is copied when you copy a suite using cylc commands.

A typical example:

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



```
        nwp-tasks.rc
        globals.rc
        ...
    doc/                # documentation
    control/            # control files
    ancil/              # ancillary files
    ...
```

8.2 Suite.rc File Overview

Suite.rc files conform to the ConfigObj extended INI format (http://www.voidspace.org.uk/python/configobj.html) with several modifications to allow continuation lines and include-files, and to make it legal to redefine environment variables and scheduler directives (duplicate config item definitions are normally flagged as an error).

Additionally, embedded template processor expressions may be used in the file, to programatically generate the final suite definition seen by cylc. Currently the Jinja2 template engine is supported (http://jinja.pocoo.org/docs). In the future cylc may provide a plug-in interface to allow use of other template engines too. See *Jinja2 Suite Templates* (Section 8.6) for some examples.

8.2.1 Syntax

The following list shows legal raw suite.rc syntax. Suites using the Jinja2 template processor (Section 8.6) can of course use Jinja2 syntax as well (it must generate raw syntax on processing).

- **Entries** are of the form `item = value`, and item names may contain spaces.
- **Comments** # follow a hash character.
- **List Values** are comma, separated.
- **Strings** must be quoted "if, they, contain, commas"
- **Multiline Strings** must be """triple-quoted""".
- **Boolean Values** are written as True or False (capitalized).
- **White Space** is ignored; indentation can be used for clarity.
- **Continuation Lines** follow a trailing backslash.\
- **[Section Headings]** are enclosed in square brackets.
- **[[Subsection Nesting]]** is indicated by the number of square brackets.[7]
- **Duplicate Items** are illegal, except in `environment` and `directives` sections.[8]
- **Include-files** can be used: `%include inc/foo.rc`.

The following pseudo-listing illustrates suite.rc syntax:

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

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

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


```
[section]
    item = value
%include inc/vars/foo.inc  # include file
    [[subsection]]
        item = value
        [[[subsubsection]]]
            item = value
[another section]
    [[another subsection]]
        # ...
    # ...
# ...
```

8.2.2 Include-Files

Cylc has native support for suite.rc include-files, which may help to organize large suites. Inclusion boundaries are completely arbitrary - you can think of include-files as chunks of the suite.rc file simply cut-and-pasted into another file. Include-files may be included multiple times in the same file, and even nested. Include-file paths can be specified portably relative to the suite definition directory, e.g.:

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

8.2.2.1 Editing Temporarily Inlined Suites

Cylc's native file inclusion mechanism supports optional inlined editing:

```
% cylc edit --inline SUITE
```

The suite will be split back into its constituent include-files when you exit the edit session. While editing, the inlined file becomes the official suite definition so that changes take effect whenever you save the file. See `cylc prep edit --help` for more information.

8.2.2.2 Include-Files via Jinja2

Jinja2 (Section 8.6) provides template inclusion functionality although this is more akin to Python module import than simple text inclusion, and the implications of this for suite design have not yet been explored.

8.2.3 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 12. 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.

8.2.4 Gross File Structure

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

- **[cylc]** - *non task-related suite configuration*
  - suite logging directories, simulation mode, UTC mode, etc.
- **[scheduling]** - *determines when tasks are ready to run*

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

8.2.5 Validation

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

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

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

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

Validation does not check the validity of chosen job submission methods; this is to allow users to extend cylc with their own job submission methods, which are by definition unknown to the suiterc.spec file.

8.3 Scheduling - Dependency Graphs

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

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

Figure 25 shows the complete suite.rc listing alongside the suite graph. This is actually a complete, valid, runnable suite (it will use default runtime properties such as dummy command scripting, because no runtime properties are explicitly defined, and you'll need to trigger task A manually to get the suite started because A[T] depends on A[T-6] and at start-up there is no previous cycle to satisfy that dependence - how to handle this properly is described in *Handling Intercycle Dependencies At Start-Up* (Section 8.3.5) and in the *Quick Start Guide* (Section 6).


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

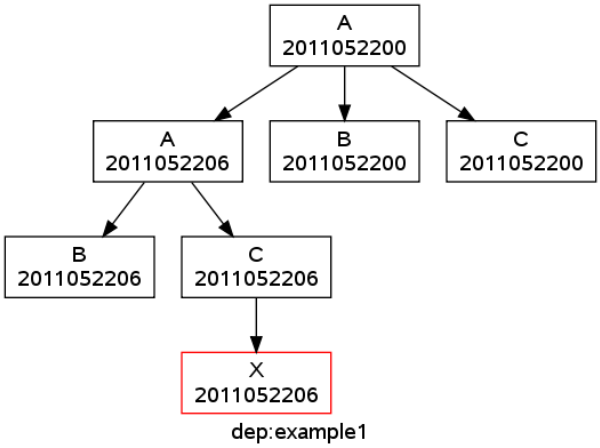

Figure 25: Example Suite

8.3.1 Graph String Syntax

Multiline graph strings may contain:

- **blank lines**
- **arbitrary white space**
- **task names**
- **internal comments:** following the # character
- **cycle time offsets:** `A[T-6]`
- **start triggers:** `foo:start => bar`
- **success triggers:** `foo:succeed => bar` OR `foo => bar`
- **fail triggers:** `foo:fail => bar`
- **suicide triggers:** `foo => !bar`
- **internal output triggers:** `foo:upload1 => bar`
- **conditional triggers:** `(A | B) & C => D`

8.3.2 Interpreting Graph Strings

Suite dependency graphs can be broken down into pairs in which the left side (which may be a single task or family, or several that are conditionally related) defines a trigger for the task or family on the right. For instance the "word graph" *C triggers off B which triggers off A* can be deconstructed into pairs *C triggers off B* and *B triggers off A*. In this section we use only the default trigger type, which is to trigger off the upstream task succeeding; see Section 8.3.4 for other available triggers.

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

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

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

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

```
# SUITE.RC
```

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

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

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

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

is equivalent to this:

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

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

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

but this isn't:

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

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

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

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

is equivalent to this:

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

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

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



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

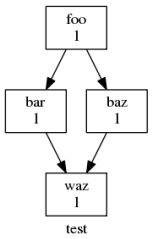

Figure 26: One-off Asynchronous Tasks.

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

8.3.2.1 Handling Long Graph Lines

Long chains of dependencies can be split into pairs:

```
# SUITE.RC
    graph = "A => B => C"
# is equivalent to this:
    graph = """A => B
                B => C"""
# BUT THIS IS AN ERROR:
    graph = """A => B => # WRONG!
                C"""      # WRONG!
```

If you have very long task names, or long conditional trigger expressions (below) then you can use the suite.rc line continuation marker:

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

Note that a line continuation marker must be the final character on the line; it cannot be followed by trailing spaces or a comment.

8.3.3 Graph Types (VALIDITY)

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

8.3.3.1 One-off Asynchronous Tasks

Figure 26 shows a small suite of one-off asynchronous tasks; these have no associated cycle time and don't spawn successors (once they're all finished the suite just exits). The integer 1 attached to each graph node is just an arbitrary label, akin to the task cycle time in cycling tasks; it increments when a repeating asynchronous task (below) spawns.

8.3.3.2 Cycling Tasks

Valid cycle times for cycling tasks are defined by the graph VALIDITY section headings - lists of hours in the day - for the graph strings in which the tasks appear. Figure 27 shows a small suite of cycling tasks.


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

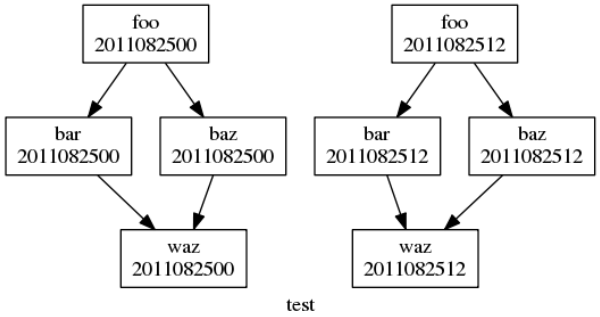

Figure 27: Cycling Tasks.

8.3.3.2.1 How Multiple Graph Strings Combine

For a cycling graph with multiple validity sections for different hours of the day, the different sections *add* to generate the complete graph. Different graph sections can overlap (i.e. the same hours may appear in multiple section headings) and the same tasks may appear in multiple sections, but individual dependencies should be unique across the entire graph. For example, the following graph defines a duplicate prerequisite for task C:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        [[[0,6,12,18]]]
            graph = "A => B => C"
        [[[6,18]]]
            graph = "B => C => X"
            # duplicate prerequisite: B => C already defined at 6, 18
```

This does not affect scheduling, but for the sake of clarity and brevity the graph should be written like this:

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        [[[0,6,12,18]]]
            graph = "A => B => C"
        [[[6,18]]]
            # X triggers off C only at 6 and 18 hours
            graph = "C => X"
```

8.3.3.3 Combined Asynchronous And Synchronous Graphs

Cycling tasks can be made to wait on one-off asynchronous tasks, as shown in Figure 28. Alternatively, they can be made to wait on one-off synchronous start-up tasks, which have an associated cycle time even though they are non-cycling - see Figure 29.

8.3.3.3.1 Synchronous Start-up vs One-off Asynchronous Tasks

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

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

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


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

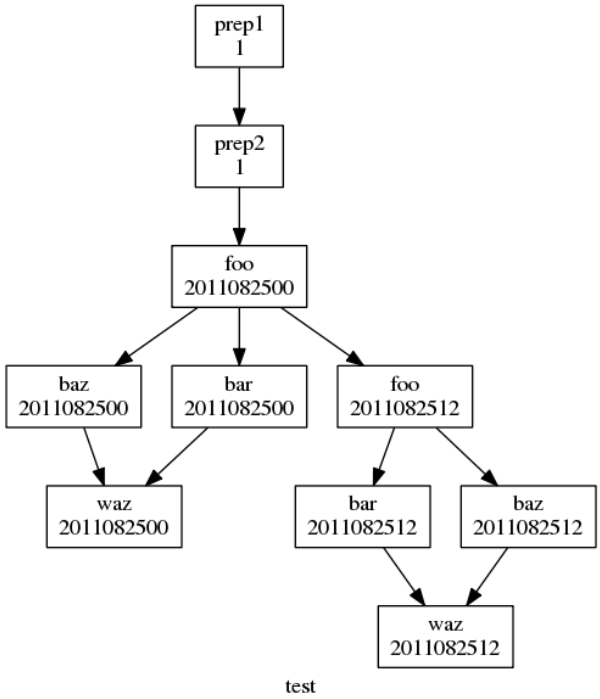

Figure 28: One-off asynchronous and cycling tasks in the same suite.

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

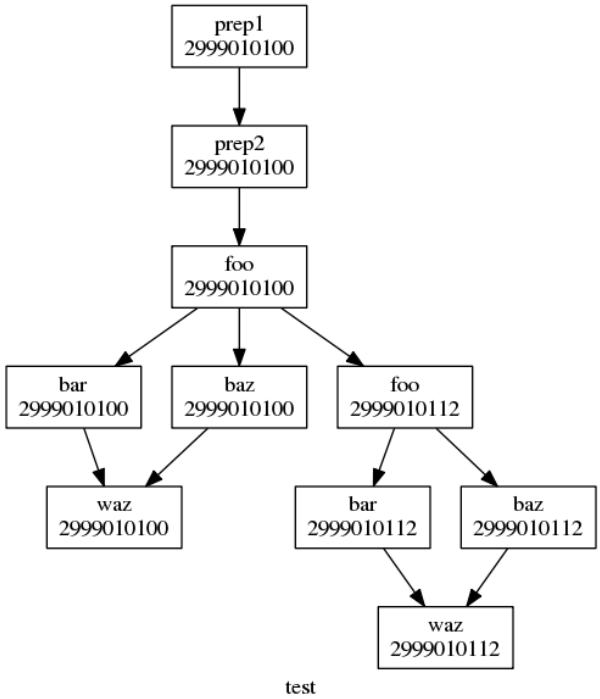

Figure 29: One-off synchronous and cycling tasks in the same suite.

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

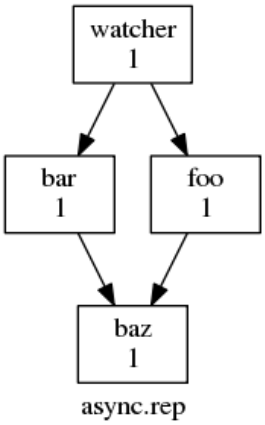

Figure 30: Repeating Asynchronous Tasks.

8.3.3.4 Repeating Asynchronous Tasks

Repeating asynchronous tasks can be used, for example, to process satellite data that arrives at irregular time intervals. Each new dataset must have a unique "asynchronous ID". If it doesn't naturally have such an ID a string representation of the data arrival time could be used. The graph VALIDITY section heading must contain "ASYNCID:" followed by a regular expression that matches the actual IDs. Additionally, one task in the suite must be a designated "daemon" that waits indefinitely on incoming data and reports each new dataset (and its ID) back to the suite by means of a special output message. When the daemon task proxy receives a matching message it dynamically registers a new output (containing the ID) that downstream tasks can then trigger off. The downstream tasks likewise have prerequisites containing the ID pattern (because they trigger off the aforementioned outputs) and when these get satisfied during dependency negotation the actual ID is substituted into their own registered outputs. Finally, each asynchronous repeating task proxy passes the ID to its task execution environment as `$ASYNCID` to allow identification of the correct dataset by task scripts. In this way a tree of tasks becomes dedicated to processing each new dataset, and multiple datasets can be processed in parallel if they become available in quick succession. As Figure 30 shows, a repeating asynchronous suite currently plots just like a one-off asynchronous suite. But at run time the daemon task stays put, while the others continually spawn successors to wait for new datasets to come in. The *asynchronous.repeating* example suite demonstrates how to do this in a real suite. *Note that other trigger types (success, failure, start, suicide, and conditional) cannot currently be used in a repeating asynchronous graph section.*

8.3.4 Trigger Types

Trigger type, indicated by *:type* after the upstream task (or family) name, determines what kind of event results in the downstream task (or family) triggering.

8.3.4.1 Success Triggers

The default, with no trigger type specified, is to trigger off the upstream task succeeding:

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

For consistency and completeness, however, the success trigger can be explicit:

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

8.3.4.2 Failure Triggers

To trigger off the upstream task reporting failure:

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

Section 8.3.4.8 (*Suicide Triggers*) shows one way of handling task B here if A does not fail.

8.3.4.3 Start Triggers

To trigger off the upstream task starting to execute:

```
# SUITE.RC
# B triggers if A STARTS EXECUTING:
    graph = "A:start => B"
```

This can be used to trigger tasks that monitor other tasks once they (the target tasks) start executing. Consider a long-running forecast model, for instance, that generates a sequence of output files as it runs. A postprocessing task could be launched with a start trigger on the model (`model:start => post`) to process the model output as it becomes available. Note, however, that there are several alternative ways of handling this scenario: both tasks could be triggered at the same time (`foo => model & post`), but depending on external queue delays this could result in the monitoring task starting to execute first; or a different postprocessing task could be triggered off an internal output for each data file (`model:out1 => post1` etc.; see Section 8.3.4.5), but this may not be practical if the number of output files is large or if it is difficult to add cylc messaging calls to the model.

8.3.4.4 Finish Triggers

To trigger off the upstream task succeeding or failing, i.e. finishing one way or the other:

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

8.3.4.5 Internal Triggers

These are only required to trigger off events that occur before a task finishes.

```
# SUITE.RC
[scheduling]
    [[dependencies]]
        [[[6,18]]]
            # B triggers off internal output "upload1" of task A:
            graph = "A:upload1 => B"
[runtime]
    [[A]]
        [[[outputs]]]
            upload1 = "NWP products uploaded for [T]"
```

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

8.3.4.6 Intercycle Triggers

Typically most tasks in a suite will trigger off other cotemporal (i.e. the 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 other kinds of intercycle dependence are possible too.[9] 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"
```

Intercycle and trigger type (and internal output) notation can be combined:

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

8.3.4.6.1 Bootstrapping Intercycle Triggers

Tasks with intercycle triggers require an associated *cold-start* task to bootstrap them into operation when the suite is cold-started, because they depend on a previous cycle that does not exist at start time. Otherwise the first such task will require manual triggering (and that will only suffice if the real task does not have real previous-cycle dependence in the first cycle). Section 8.3.5, *Handling Intercycle Dependence At Start-Up*, explains how to use cold-start tasks in cylc.

8.3.4.7 Conditional Triggers

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

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

OR operators (`|`) which provide true conditional triggers, can only appear on the left,[10]

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

[9] 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.

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


```
# SUITE.RC
        graph = """
# D triggers if A or (B and C) succeed
A | B & C => D
# just to align the two graph sections
D => W
# Z triggers if (W or X) and Y succeed
(W|X) & Y => Z
                """
```

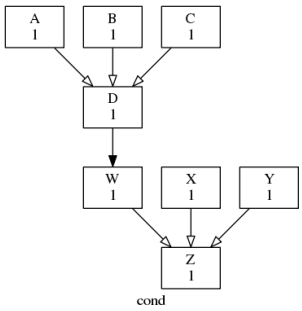

Figure 31: Conditional triggers are plotted with open arrow heads.

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

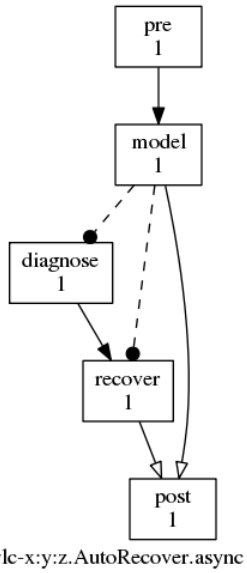

Figure 32: Automated failure recovery via suicide triggers.

Forecasting suites typically have simple requirements conditional triggering requirements, but any valid conditional expression can be used, as shown in Figure 31 (conditional triggers are plotted with open arrow heads).

8.3.4.8 Suicide Triggers

Suicide triggers take tasks out of the suite. This can be used for automated failure recovery. The suite.rc listing and accompanying graph in Figure 32 show how to define a chain of failure recovery tasks that trigger if they're needed but otherwise remove themselves from the suite (you can run the *AutoRecover.async* example suite to see how this works). The dashed graph edges ending in solid dots indicate suicide triggers, and the open arrowheads indicate conditional triggers as usual.

8.3.4.9 Family Triggers

Family names defined by the namespace inheritance hierarchy (Section 8.4) can be used in the suite graph as shorthand for dependencies among the member tasks. Current semantics are:

- *family started* means *one or more members started*
- *family succeeded* means *all members succeeded*

- *family failed* means *all members finished but one or more of them failed*

For example, this:

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

is equivalent to this:

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

And this:

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

is equivalent to this:

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

If current family trigger semantics (i.e. what a family trigger means in terms of member behaviour) does not fit your requirements you will have to use task members explicitly in the graph. In the future cylc may support alternate family trigger semantics; for instance *family succeeded* could mean *at least one member succeeded* and *family failed* could mean *all members failed*.

Task family triggers should be used sparingly as a convenient simplification for groups of tasks that would naturally trigger at the same time anyway (e.g. forecast ensembles and multiple tasks for processing different observations at the same time) and whose outputs would all be put to use at a similar time too. Otherwise family triggers will unnecessarily constrain cylc's ability to achieve maximum functional parallelism, because all family members have to finish before downstream processing can continue.[11]

8.3.5 Handling Intercycle Dependence At Start-Up

In suites with intercycle dependence some kind of bootsrapping process is required to get the suite going initially. In the example shown in *Intercycle Triggers* (Section 8.3.4.6), for instance, in the very first cycle there is no previous instance of task A to satisfy B's prerequisites.

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

8.3.5.1 Cold-Start Tasks

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

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

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

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

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

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

8.3.5.2 Warm-Starting A Suite

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

8.3.6 Model Restart Dependencies

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

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

In other words, task A can trigger off a cotemporal cold-start task, *or* off its own previous instance, *or* off the instance before that, and so on. Restart dependencies are unusual because

although A *could* trigger off A[T-12] we don't actually want it to do so unless A[T-6] fails and can't be fixed. *This is why Task A, above, is declared to be 'sequential'.*[12] 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 to have explicit restart outputs, and is not declared to be sequential, and you define appropriate labeled restart outputs which *must contain the word 'restart'*, then the task will spawn as soon its last restart output is completed so that successives instances of the task will be able to overlap (i.e. run in parallel) if the opportunity arises. Whether or not this is worth the effort depends on your needs.

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

8.4 Runtime Properties - Task Execution

The `[runtime]` section of a suite.rc file configures what to execute, and where and how, when tasks are ready to run. The sections immediately below `[runtime]` are, for want of a better term, *namespaces*[13] that define runtime properties for individual tasks and families of tasks.

Every namespace contains the same set of configuration items (see the *Suite.rc Reference*, Appendix A, for the complete list of items). Namespaces can inherit from other namespaces,

[12] 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.

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

overriding inherited items as required; *this allows configuration of related tasks without repetition.* A namespace represents a family if other namespaces inherit from it.

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

Nested families from the namespace inheritance hierarchy, even if they are not used as family triggers in the graph, can be expanded or collapsed in the suite dependency graphs viewer, the graph view of the suite control GUI, and since cylc-4.5.0, in the text tree view of the control GUI.

The following listing of the *namespace.one* example suite illustrates basic runtime property inheritance. How it works should be reasonable clear by inspection; if not read on.

```
# SUITE.RC
title = "User Guide [runtime] example."
[cylc]
    simulation mode only = True # (no task implementations)
[scheduling]
    initial cycle time = 2011010106
    final cycle time = 2011010200
    [[dependencies]]
        graph = "foo => OBS => bar"
[runtime]
    [[root]] # base namespace for all tasks (defines suite-wide defaults)
        [[[job submission]]]
            method = at_now
        [[[environment]]]
            COLOR = red
    [[OBS]]  # family (inherited by land, ship); implicitly inherits root
        command scripting = run-<TASK>.sh
        [[[environment]]]
            RUNNING_DIR = $HOME/running/$CYLC_TASK_NAME
    [[land]] # a task (a leaf on the inheritance tree) in the OBS family
        inherit = OBS
        description = land obs processing
    [[ship]] # a task (a leaf on the inheritance tree) in the OBS family
        inherit = OBS
        description = ship obs processing
        [[[job submission]]]
            method = loadleveler
        [[[environment]]]
            RUNNING_DIR = $HOME/running/ship  # override OBS environment
            OUTPUT_DIR = $HOME/output/ship    # add to OBS environment
    [[foo]]
        # (just inherits from root)

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

8.4.1 Namespace Names

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

8.4.2 Root - Runtime Defaults

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

Most root items are unset by default, but some values are set, sufficient to allow simple test suites to be defined by dependency graph alone - as can be seen from many of the examples that illustrate this User Guide (command scripting, for example, defaults to printing a simple message, sleeping for ten seconds, and then exiting). These default values are documented with each configuration item in Appendix A, and Section A.7 shows them all in context. You can override them or provide your own defaults for other items by explicitly configuring the root namespace.

8.4.3 Defining Multiple Namespaces At Once

Groups of similar tasks or families that differ only in a few configuration details can be defined in single namespace sections by putting a list of names in the section heading. Potential applications include groups of similar obs processing tasks, and forecasting ensembles. Note that if the list of tasks are also members of the same family then any common runtime configuration could equally go under the inherited family namespace.

As the suite.rc file is parsed any occurrence of the special variable `<NAMESPACE>` in a runtime configuration item will be replaced immediately with the name of the namespace in which the special variable appears. Then, after inheritance processing, in each task any occurrence of `<TASK>` will be replaced with the name of the task. If both special variables are used in a family namespace definition, then in inheriting member tasks `<NAMESPACE>` will evaluate to the family name while `<TASK>` will evaluate to the task name.

The following example illustrates this:

```
# SUITE.RC
[runtime]
    [[foo]]
    command scripting = "echo hello from <TASK> via <NAMESPACE>"
    [[bar, baz]]
        inherit = foo
```

```
% cylc get-config test runtime foo 'command scripting'
['echo Hello from foo via foo']
% cylc get-config test runtime bar 'command scripting'
['echo Hello from bar via foo']
% cylc get-config test runtime baz 'command scripting'
['echo Hello from baz via foo']
```

Of course particular tasks can also be singled out at runtime, in common command scripting or invoked task scripts, by testing the `$CYLC_TASK_NAME` variable.

As another example, consider a suite containing an ensemble of closely related forecast tasks in which each task invokes the same external script but with a unique argument that reflects the task name (which presumably results in the right initial conditions being used, or similar):

```
# SUITE.RC
[runtime]
    [[ensemble]]
        command scripting = "run-model.sh ic-<TASK>"
    [[m1, m2, m3]]
        # ensemble member tasks
        inherit = ensemble
```

This defines a unique command line for each ensemble member task:

```
% cylc get-config test runtime m2 'command scripting'
['run-model.sh ic-m2']
```


For large ensembles Jinja2 template processing can be used to automatically generate the member names and associated dependencies (see Section 8.6).

Note that in the namespace scripting and environment sections, which are evaluated in task job scripts at run time, `<TASK>` is functionally equivalent to `$CYLC_TASK_NAME`, but in runtime config items that are never interpreted by the shell (e.g. `description`) `<TASK>` must be used to specialize from the list of names in the namespace section heading to the individual task name.

8.4.4 Task Execution Environment

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

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

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

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

8.4.4.1 User Environment Variables

A task's user-defined environment results from its inherited `[[[environment]]]` sections:

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

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

8.4.4.2 Overriding Environment Variables

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

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


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

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

8.4.4.3 Suite And Task Identity Variables

The task identity variables provided to tasks by cylc are:

```
$CYLC_TASK_ID                    # X%2011051118 (e.g.)
$CYLC_TASK_NAME                  # X
$CYLC_TASK_CYCLE_TIME            # 2011051118
$CYLC_TASK_LOG_ROOT              # ~/cylc-run/foo.bar.baz/log/job/X%2011051118-1317298378.191123
$CYLC_TASK_NAMESPACE_HIERARCHY   # "root postproc X" (e.g.)
```

And the suite identity variables are:

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

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

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

8.4.4.4 Suite Share And Task Work Directories

The following variables are also available to running tasks, and can be configured in the suite.rc file (see Sections A.4.1.9.5 and A.4.1.9.6):

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

Task command scripting is executed from within a work directory created on the fly, if necessary, by the task's job script. In non-detaching tasks the work directory is automatically removed again *if it is empty* before the job script exits.

The share directory is also created on the fly, if necessary, by the job script. It is intended as a shared data area for multiple tasks on the same host, but as for any task runtime config item it can be specialized to particular tasks or groups of tasks.

The code for creating these directories, and removing empty work directories, can be seen by examining a task job script.

8.4.4.5 Other Cylc-Defined Environment Variables

Initial and final cycle times, if supplied via the suite.rc file or the command line, are passed to task execution environments as:


```
$CYLC_SUITE_INITIAL_CYCLE_TIME
$CYLC_SUITE_FINAL_CYCLE_TIME
```

Running tasks can use these to determine whether or not they are running in the first or final cycles.

8.4.4.6 Environment Variable Evaluation

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

8.4.5 Remote Task Hosting

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

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

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

- passwordless ssh must be configured between the suite owner on the suite host and the task owner on the remote host.
- cylc and Pyro must be installed on the remote host so that the remote task can communicate with the suite (but graphviz, Pygraphviz, and Jinja2 are not needed there).
- the suite definition directory must be installed on the remote host, if the task needs access to scripts in the suite bin directory or to any other files stored there.

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

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

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

The remote cylc directory must be used to give remote tasks access to cylc commands if cylc is not in the default `$PATH`) on the remote host. Similarly the remote suite directory gives task access to suite files via `$CYLC_SUITE_DEF_PATH` on the remote host and to the suite bin directory via `$PATH`. If a remote suite definition directory is not given the local path will be used with the local user's home directory, if present, substituted for the literal `$HOME` for evaluation on the remote host.

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

8.4.5.1 Remote Log Directories

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

8.5 Visualization

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

8.5.1 Collapsible Task Families In Suite Graphing And GUIs

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

Nested families from the namespace inheritance hierarchy, even if they are not used as family triggers in the graph, can be expanded or collapsed in suite graphs and in the suite control GUI's text and graph views.

In the graph view ungraphed tasks, which includes the members of collapsed families, are automatically plotted as rectangular nodes to the right of the main graph if they are doing anything interesting (submitted, running, or failed).

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

Figure 33 illustrates successive expansion of nested task families in the *namespaces* example suite, which has the following namespace hierarchy:

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

8.6 Jinja2 Suite Templates

Support for the Jinja2 template processor adds general variables, mathematical expressions, loop control structures, and conditional expressions to suite.rc files - which are automatically preprocessed to generate the final suite definition seen by cylc.

The need for Jinja2 processing must be declared with a hash-bang comment as the first line of the suite.rc file:

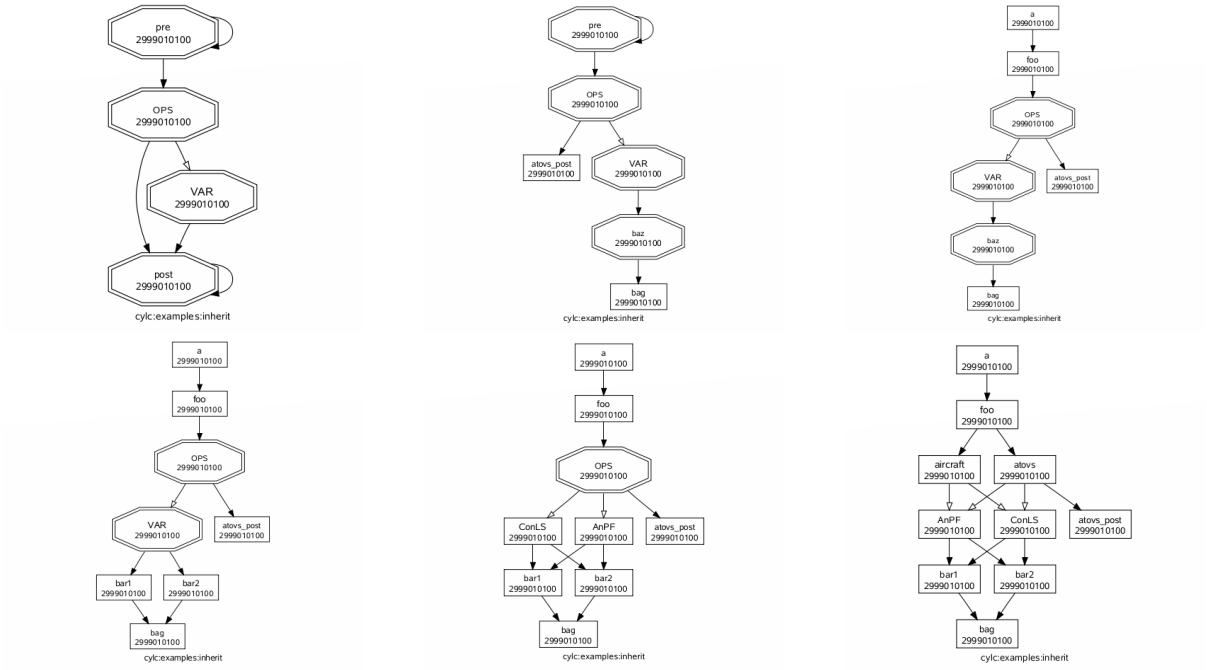

Figure 33: Graphs of the *namespaces* example suite showing various states of expansion of the nested namespace family hierarchy, from all families collapsed (top left) through to all expanded (bottom right). This can also be done by right-clicking on tasks in the suite control GUI graph view.

```
#!Jinja2
# ...
```

Potential uses for this include automatic generation of repeated groups of similar tasks and dependencies, and inclusion or exclusion of entire suite sections according to the value of a single flag. Consider a large complicated operational suite and several related parallel test suites with slightly different task content and structure (the parallel suites, for instance, might take certain large input files from the operation or the archive rather than downloading them again) - these can now be maintained as a single master suite definition that reconfigures itself according to the value of a flag variable indicating the intended use.

Template processing is the first thing done on parsing a suite definition so Jinja2 expressions can appear anywhere in the file (inside strings and namespace headings, for example).

Jinja2 is well documented at http://jinja.pocoo.org/docs, so here we just provide an example suite that uses it. The meaning of the embedded Jinja2 code should be reasonably self-evident to anyone familiar with standard programming techniques.

The `jinja2.ensemble` example, graphed in Figure 34, shows an ensemble of similar tasks generated using Jinja2:

```
#!jinja2
{% set N_MEMBERS = 5 %}
[scheduling]
    [[dependencies]]
        graph = """{# generate ensemble dependencies #}
            {% for I in range( 0, N_MEMBERS ) %}
               foo => mem_{{ I }} => post_{{ I }} => bar
            {% endfor %}"""
```

Here is the generated suite definition, after Jinja2 processing:

```
#!jinja2
[scheduling]
    [[dependencies]]
```

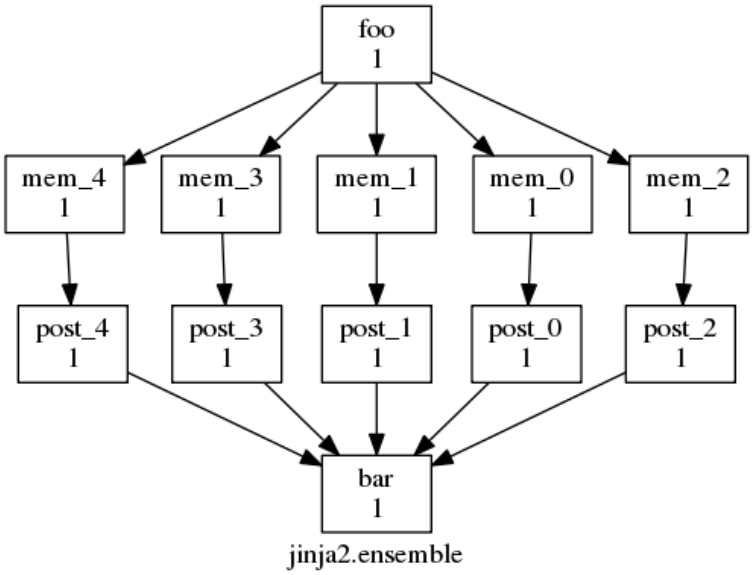

Figure 34: The Jinja2 ensemble example suite graph.

```
        graph = """
          foo => mem_0 => post_0 => bar
          foo => mem_1 => post_1 => bar
          foo => mem_2 => post_2 => bar
          foo => mem_3 => post_3 => bar
          foo => mem_4 => post_4 => bar
                """
```

And finally, the `jinja2.cities` example uses variables, includes or excludes special cleanup tasks according to the value of a logical flag, and it automatically generates all dependencies and family relationships for a group of tasks that is repeated for each city in the suite. To add a new city and associated tasks and dependencies simply add the city name to list at the top of the file. The suite is graphed, with the New York City task family expanded, in Figure 35.

```
#!Jinja2

title = "Jinja2 city suite example."
description = """
Illustrates use of variables and math expressions, and programmatic
generation of groups of related dependencies and runtime properties."""

{% set HOST = "SuperComputer" %}
{% set CITIES = 'NewYork', 'Philadelphia', 'Newark', 'Houston', 'SantaFe', 'Chicago' %}
{% set CITYJOBS = 'one', 'two', 'three', 'four' %}
{% set LIMIT_MINS = 20 %}

{% set CLEANUP = True %}

[scheduling]
    [[ dependencies ]]
{% if CLEANUP %}
        [[[23]]]
            graph = "clean"
{% endif %}
        [[[0,12]]]
            graph = """
                    setup => get_lbc & get_ic # foo
{% for CITY in CITIES %} {# comment #}
                    get_lbc => {{ CITY }}_one
                    get_ic => {{ CITY }}_two
```

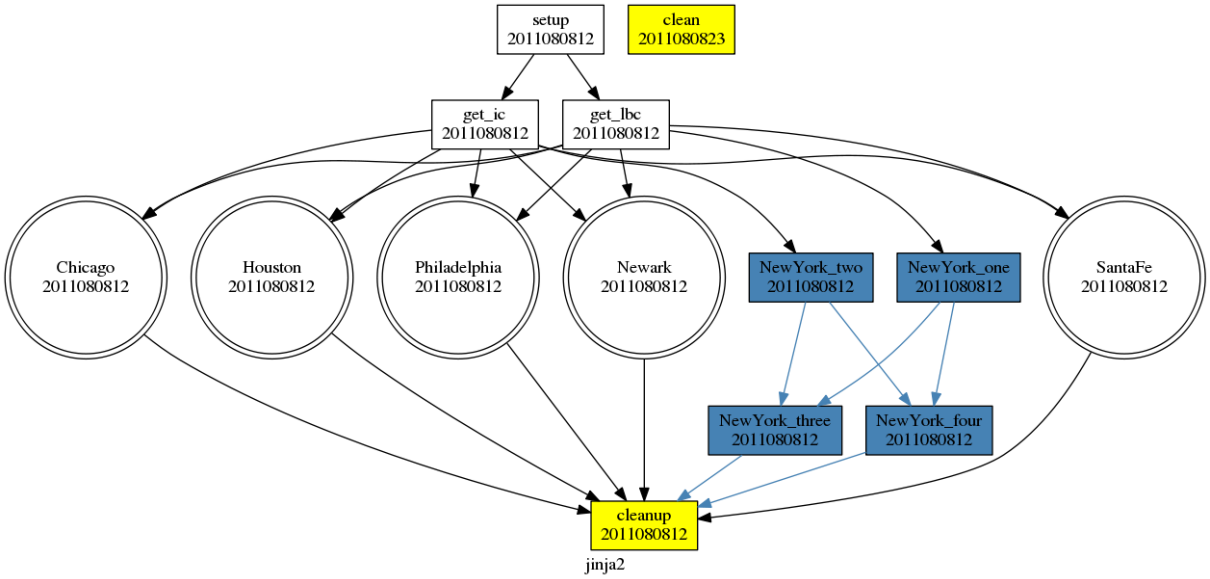

Figure 35: The Jinja2 cities example suite graph, with the New York City task family expanded.

```
                    {{ CITY }}_one & {{ CITY }}_two => {{ CITY }}_three & {{ CITY }}_four
{% if CLEANUP %}
                    {{ CITY }}_three & {{ CITY }}_four => cleanup
{% endif %}
{% endfor %}
                    """
[runtime]
    [[on_{{ HOST }} ]]
        [[[remote]]]
            host = {{ HOST }}
            cylc directory = /path/to/cylc
        [[[directives]]]
            wall_clock_limit = "00:{{ LIMIT_MINS|int() + 2 }}:00,00:{{ LIMIT_MINS }}:00"

{% for CITY in CITIES %}
    [[ {{ CITY }} ]]
        inherit = on_{{ HOST }}
{% for JOB in CITYJOBS %}
    [[ {{ CITY }}_{{ JOB }} ]]
        inherit = {{ CITY }}
{% endfor %}
{% endfor %}

[visualization]
    initial cycle time = 2011080812
    final cycle time = 2011080823
    [[node groups]]
        cleaning = clean, cleanup
    [[node attributes]]
        cleaning = 'style=filled', 'fillcolor=yellow'
        NewYork = 'style=filled', 'fillcolor=lightblue'
```

8.6.1 Accessing Environment Variables With Jinja2

This functionality is not provided by Jinja2 by default, but cylc automatically imports the user environment to the template in a dictionary structure called *environ*. A usage example:

```
#!Jinja2
#...
```

```
[runtime]
    [[root]]
        [[[environment]]]
            SUITE_OWNER_HOME_DIR_ON_SUITE_HOST = {{environ['HOME']}}
```

This example is emphasizes that *the environment is read on the suite host at the time the suite definition is parsed* - it is not, for instance, read at task run time on the task host.

8.6.2 Custom Jinja2 Filters

Jinja2 variable values can be modified by "filters", using pipe notation. For example, the built-in `trim` filter strips leading and trailing white space from a string:

```
{% set MyString = "   dog   " %}
{{ MyString | trim() }}  # "dog"
```

(See official Jinja2 documentation for available built-in filters.)

Cylc also supports custom Jinja2 filters. A custom filter is a single Python function in a source file with the same name as the function (plus ".py" extension) and stored in one of the following locations:

- `$CYLC_DIR/lib/Jinja2Filters/`
- `[suite definition directory]/Jinja2Filters/`
- `$HOME/.cylc/Jinja2Filters/`

In the filter function argument list, the first argument is the variable value to be "filtered", and subsequent arguments can be whatever is needed. Currently there is one custom filter called "pad" in the central cylc Jinja2 filter directory, for padding string values to some constant length with a fill character - useful for generating task names and related values in ensemble suites:

```
{% for i in range(0,100) %}  # 0, 1, ..., 99
    {% set j = i | pad(2,'0') %}
    A_{{j}}          # A_00, A_01, ..., A_99
{% endfor %}
```

8.7 Special Placeholder Variables

A small number of special variables are used as placeholders in cylc suite definitions:

- `[T]` and `[T+n]`
  Where `n` is an integer cycle time offset, the units of which (e.g. hours, days, months, or years) is determined by the cycling module in use. This is replaced, in internal task message strings under the runtime outputs config section, by the actual offset cycle time of the task. The syntax is the same as for cycle time offsets in the suite graph; and they also allow cylc to avoid erroneously matching literal capital T's in the output message.
- `<TASK>` and `<NAMESPACE>`
  These are replaced with actual task or namespace names in runtime settings. The difference between the two is explained in Section 8.4.3, *Defining Multiple Namespaces At Once*.
- `<ASYNCID>`
  This is replaced with the actual "asynchronous ID" (e.g. satellite pass ID) for repeating asynchronous tasks.

To use proper variables (c.f. programming languages) in suite definitions, see the Jinja2 template processor (Section 8.6).

8.8 Omitting Tasks At Runtime

It is sometimes convenient to omit certain tasks from the suite at runtime without actually deleting their definitions from the suite.

Defining [runtime] properties for tasks that do not appear in the suite graph results in verbose-mode validation warnings that the tasks are disabled. They cannot be used because the suite graph is what defines their dependencies and valid cycle times. Nevertheless, it is legal to leave these orphaned runtime sections in the suite definition because it allows you to temporarily remove tasks from the suite by simply commenting them out of the graph.

To omit a task from the suite at runtime but still leave it fully defined and available for use (by insertion or `cylc submit`) use one or both of [scheduling][[special task]] lists, *include at start-up* or *exclude at start-up* (documented in Sections A.3.5.8 and A.3.5.7). Then the graph still defines the validity of the tasks and their dependencies, but they are not actually inserted into the suite at start-up. Other tasks that depend on the omitted ones, if any, will have to wait on their insertion at a later time or otherwise be triggered manually.

Finally, with Jinja2 (Section 8.6) you can radically alter suite structure by including or excluding tasks from the [scheduling] and [runtime] sections according to the value of a single logical flag defined at the top of the suite.

9 Task Implementation

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

9.1 Most Tasks Require No Modification For Cylc

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

- Tasks with internal (pre-completion) outputs that other tasks need to trigger off must be modified to report to the suite when those outputs have been completed.
- Tasks that spawn secondary internal processes which they then detach from and exit early must be modified so that the secondary processes report final success or failure to the suite.
- Tasks that do not return non-zero exit status on error must be made to do so, to allow cylc's automatic error trapping to work. This is normal best-practice scripting anyway.

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

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

```
# SUITE.RC
[runtime]
    [[foo]]
        description = a task that runs foo.sh
        command scripting = foo.sh OPTIONS ARGUMENTS
    [[bar]]
        description = a task that runs bar.sh
        command scripting = """echo HELLO
                               bar.sh
                               echo BYE"""
    [[baz]]
        description = a task that runs baz.sh and retries on failure
        retry delays = 1,1
        command scripting = """echo attempt No. $CYLC_TASK_TRY_NUMBER
                               baz.sh"""
```


9.2 Suite.rc Inlined Tasks

Simple tasks can be entirely implemented within the suite.rc file because the task *command scripting* string can contain as many lines of code as you like.

9.3 Voluntary Messaging Modifications

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

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

You can also use this syntax:

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

But not this:

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

You can also send warning messages, or general information:

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

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

9.4 Tasks Requiring Modification For Cylc

There are two main categories of tasks that require some modification to work with cylc: those with internal (pre-completion) outputs that other tasks need to trigger off, and those that spawn detaching internal processes prior to exiting before the spawned processing is finished. Additionally, any task that fails to indicate a fatal error by return non-zero exit status must be corrected.

9.4.1 Returning Non-zero Exit Status On Error

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


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

See Section 9.3, *Voluntary Messaging Modifications*, for more examples of error detection and cylc messaging.

9.4.2 Reporting Internal Outputs Completed

If a task has *internal outputs* that others can trigger off before it finishes, then it must report completion of those outputs with messages sent back to the suite at appropriate times. Output messages must be unique within the suite or else downstream tasks will trigger off whichever task happens to send the message first; they must exactly match the corresponding outputs registered for the task because cylc distinguishes between registered outputs that others can trigger off and general messages that are just logged; and for cycling tasks they must contain the cycle time in order to distinguish between the outputs of successive instances of the same task.

The "outputs" example is a self-contained suite that illustrates use of internal outputs:

```
title = "triggering off internal task outputs"

description = """
This is a self contained example (task implementation, including output
messaging, is entirely contained within the suite definition)."""

[scheduling]
    initial cycle time = 2010080806
    final cycle time = 2010080812
    [[dependencies]]
        [[[0,12]]]
          graph = """
            foo:out1 => bar
            foo:out2 => baz
                  """
[runtime]
    [[foo]]
        command scripting = """
echo HELLO
sleep 10
# use task runtime environment variables here
cylc message "$CYLC_TASK_NAME uploaded file set 1 for $CYLC_TASK_CYCLE_TIME"
sleep 10
cylc message "$CYLC_TASK_NAME uploaded file set 2 for $CYLC_TASK_CYCLE_TIME"
sleep 10
echo BYE"""
        [[[outputs]]]
            # use cylc placeholder variables here
            out1 = "<TASK> uploaded file set 1 for <CYLC_TASK_CYCLE_TIME>"
            out2 = "<TASK> uploaded file set 2 for <CYLC_TASK_CYCLE_TIME>"
```

Note that cycle time and task name in the output message registration are expressed by special placeholder variables in angle brackets (see Section 8.7) not by the corresponding environment variables in the task runtime environment. This is because registered output messages are held by task proxies, inside cylc, for comparison with incoming task messages; they are never interpreted by the shell and consequently may not contain environment variables. The actual messaging calls made by running tasks, on the other hand, can make use of variables in the task runtime environment.

9.4.3 Reconnecting Detaching Processes

You may be able to convert a detaching task to a non-detaching task very easily. If the detaching process is just a background shell process, for instance, run it in the foreground instead; for

loadleveler the `-s` option prevents `llsubmit` from returning until the job has completed; for Sun Grid Engine, `qsub -sync yes` has the same effect. Section 10.4 shows how to override the command template used by cylc in order to customize job submission command options like this.

9.4.4 Tasks That Detach And Exit Early

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

First check that you can't easily "reconnect" the detaching internal processes, as described above in Section 9.4.3. If not, then start by disabling cylc's automatic completion messaging:

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

Now, reporting success or failure is just a matter of calling the cylc messaging commands:

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

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

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

To make this easier, for tasks that declare manual completion messaging, cylc makes non user-defined environment scripting available in a single variable called `$CYLC_SUITE_ENVIRONMENT` that can be inserted into the aforementioned native task scripts prior to calling the cylc messaging commands.[14]

[14] 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.

9.4.5 A Custom Task Wrapper Example

The *detaching* example suite 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

MY_TMPDIR=${CYLC_TMPDIR:-${TMPDIR:-/tmp}}

RES=$MY_TMPDIR/atnow$$.txt
( at now <<EOF
$SCRIPT 1> $OUT 2> $ERR
EOF
) > $RES 2>&1
if grep 'No atd running' $RES; then
    echo 'ERROR: atd is not running!'
    exit 1
fi
# model.sh should now be running at the behest of the 'at' scheduler.
echo "run-model.sh: done"
```

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

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

```
title = "Cylc User Guide 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 Cylc User Guide."""

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

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

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


```
#/bin/bash
set -e

# A custom wrapper for the 'model' task from the detaching example suite.
# See the Cylc User Guide for more information.

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

MY_TMPDIR=${CYLC_TMPDIR:-${TMPDIR:-/tmp}}
# Get a temporary copy of the native job scripts:
TDIR=$MY_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 ${PREFIX}-${CYLC_TASK_CYCLE_TIME}-$$.out ${PREFIX}-${CYLC_TASK_CYCLE_TIME}-$$.err

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

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

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

10 Task Job Submission

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

10.1 Task Job Scripts

When a task is ready to run cylc generates a temporary *task job script* to configure the task's execution environment and call its command scripting. The job script is the embodiment of all suite.rc runtime settings for the task. It is submitted to run by the *job submission method* configured for the task. Different tasks can have different job submission methods. Like other runtime properties, you can set a suite default job submission method and override it for specific tasks or families:

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

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

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

And here is the generated job script:

```
#!/bin/bash

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

echo "TASK JOB SCRIPT STARTING"

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

# TASK IDENTITY:
export CYLC_TASK_ID=Model%2011080506
export CYLC_TASK_NAME=Model
export CYLC_TASK_CYCLE_TIME=2011080506
export CYLC_TASK_LOG_ROOT=~/cylc-run/QuickStart.c/log/job/Model%2011080506-1317298378.191123
export CYLC_TASK_NAMESPACE_HIERARCHY="root Models Model"

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

# SET ERROR TRAPPING:
set -u # Fail when using an undefined variable
# Define the trap handler
HANDLE_TRAP() {
```



```
  echo Received signal "$@"
  cylc task failed "Task job script received signal $@"
  trap "" EXIT
  exit 0
}
# Trap signals that could cause this script to exit:
trap "HANDLE_TRAP EXIT" EXIT
trap "HANDLE_TRAP ERR"  ERR
trap "HANDLE_TRAP TERM" TERM
trap "HANDLE_TRAP XCPU" XCPU

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

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

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

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

# TASK COMMAND SCRIPTING:
Model.sh

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

# SEND TASK SUCCEEDED MESSAGE:
cylc task succeeded

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

#EOF
```

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

10.2 Built-in Job Submission Methods

Cylc has a number of built-in job submission methods. If these do not suite your needs Section 10.5 shows how to extend cylc with new user-defined job submission methods.

10.2.1 background

This job submission method runs tasks directly in a background shell.

10.2.2 at

This job submission method submits tasks to the rudimentary Unix `at` scheduler. The `atd` daemon must be running.

10.2.3 loadleveler

This job submission method submits tasks to loadleveler using the `llsubmit` command. Loadleveler directives can be provided in the suite.rc file:

```
# SUITE.RC
[runtime]
    [[__NAME__]]
        [[[directives]]]
            foo = bar
            baz = waz
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
# @ foo = bar
# @ baz = waz
# @ queue
```

10.2.4 pbs

This job submission method submits tasks to PBS (or Torque) using the `qsub` command. PBS directives can be provided in the suite.rc file:

```
# SUITE.RC
[runtime]
    [[__NAME__]]
        [[[directives]]]
            -q = foo
            -l = 'nodes=1,walltime=00:01:00'
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
#PBS -q foo
#PBS -l nodes=1,walltime=00:01:00
```

10.2.5 sge

This job submission method submits tasks to Sun Grid Engine using the `qsub` command. SGE directives can be provided in the suite.rc file:

```
# SUITE.RC
[runtime]
    [[__NAME__]]
        [[[directives]]]
            -cwd = ' '
            -q = foo
            -l = 'h_data=1024M,h_rt=24:00:00'
```

These are written to the top of the task job script like this:

```
#!/bin/bash
# DIRECTIVES
#$ -cwd
#$ -q foo
#$ -l h_data=1024M,h_rt=24:00:00
```

10.2.6 Default Directives Provided

For loadleveler, pbs, and sge job submission, default directives are provided to set the job name and stdout and stderr file paths.

10.2.7 PBS and SGE Cylc Quirks

As shown in the example above, multiple entries for the same PBS or SGE directive option must be comma-separated on the same line, in the suite.rc file. Otherwise, repeating the option on another line will override the previous entry, not add to it. Also, the right-hand side must be quoted to hide the comma from the suite.rc parser (commas indicate list values, whereas directives are treated as singular).

As also shown in the example above, to get a naked option flag such as `-cwd` in SGE you must give a quoted blank space as the directive value in the suite.rc file.

10.3 Task stdout And stderr Log Locations

When a task is ready to run cylc generates a filename root containing the task name, cycle time, and a unique string of digits (seconds since epoch; so that rerunning the task won't overwrite old output):

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

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

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

Cylc obviously has no control over the stdout and stderr output from tasks that do their own internal output management (e.g. tasks that submit internal jobs and direct the associated output to other files). For less internally complex tasks, however, the files referred to here will be complete task job logs.

Cylc's suite control GUIs can display the task job logs (updating in real time for local tasks).

10.4 Overriding the Job Submission Command

To change the form of the actual command used to submit a job you do not need to define a new job submission method; just override the `command template` in the relevant job submission sections of your suite.rc file:

```
# SUITE.RC
[runtime]
    [[root]]
        [[[job submission]]]
            method = loadleveler
            # Use '-s' to stop llsubmit returning until all job steps have completed:
            command template = llsubmit -s %s
```

As explained in the suite.rc reference (Appendix A), the template's first %s will be substituted by the job file path and, where applicable a second and third %s will be substituted by the paths to the job stdout and stderr files.

10.5 Defining New Job Submission Methods

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


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

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

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

from job_submit import job_submit

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

Here is the `at` method:

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

from job_submit import job_submit

class at( job_submit ):
    """
Submit the task job script to the simple 'at' scheduler. The 'atd' daemon
service must be running.
    """
    COMMAND_TEMPLATE = "echo \"%s 1>%s 2>%s\" | at now"

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

And here is the `pbs` method:

```
#!/usr/bin/env python
from job_submit import job_submit

class pbs( job_submit ):
    """
PBS qsub job submission.
    """

    COMMAND_TEMPLATE = "qsub %s"

    def set_directives( self ):
        self.directive_prefix = "#PBS"
        self.final_directive  = None
        self.directive_connector = " "

        defaults = {}
        defaults[ '-N' ] = self.task_id
        defaults[ '-o' ] = self.stdout_file
        defaults[ '-e' ] = self.stderr_file
```



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

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

10.5.1 An Example

The following user-defined job submission class, called *qsub*, overrides the built-in *pbs* class to change the directive prefix from `#PBS` to `#QSUB`:

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

# to import from outside of the cylc source tree:
from cylc.job_submission.pbs import pbs
# OR, from $CYLC_DIR/lib/cylc/job_submission
# from pbs import pbs

class qsub( pbs ):
    """
This is a user-defined job submission method that overrides the '#PBS'
directive prefix of the built-in pbs method.
    """
    def set_directives( self ):
        pbs.set_directives( self )
        # override the '#PBS' directive prefix
        self.directive_prefix = "#QSUB"
```

To check that this works correctly save the new source file to `qsub.py` in one of the allowed locations (see just below), use it in a suite definition:

```
# SUITE.rc
# $HOME/test/suite.rc
[scheduling]
    [[dependencies]]
        graph = "a"
[runtime]
    [[root]]
        [[[job submission]]]
            method = qsub
        [[[directives]]]
            -I = bar=baz
            -l = 'nodes=1,walltime=00:01:00'
            -cwd = ' '
```

and generate a job script to see the resulting directives:

```
$ cylc db reg test $HOME/test
$ cylc jobscript test a | grep QSUB
#QSUB -e /home/oliverh/cylc-run/pbs/log/job/a%1-1321046049.703576.err
#QSUB -l nodes=1,walltime=00:01:00
#QSUB -o /home/oliverh/cylc-run/pbs/log/job/a%1-1321046049.703576.out
#QSUB -N a%1
#QSUB -I bar=baz
#QSUB -cwd
```

10.5.2 Where To Put New Job Submission Modules

You new job submission class code should be saved to a file with the same name as the class (plus ".py" extension). It can reside in any of the following locations, depending on how generally useful the new method is and whether or not you have write-access to the cylc source tree:

- a `python` sub-directory of your suite definition directory.
- any directory in, or added to, your `PYTHONPATH` environment variable.
- in the `lib/cylc/job_submission` directory of the cylc source tree.

Note that the form of the import statement at the top of the new user-defined Python module differs depending on whether or not the file is installed in the cylc source tree (see the comment at the top of the example file above).

11 Running Suites

This section may be incomplete - please see also the Quick Start Guide (Section 6), command documentation (Section B), and play with the example suites.

11.1 Task States

As a suite runs its tasks (or rather the task proxies that represent the real tasks) move through a series of defined states:

- **waiting** - prerequisites have not yet been satisfied (clock-triggered tasks also wait on their trigger time).
- **queued** - ready to be submitted (i.e. prerequisites satisfied) but temporarily held back by an internal cylc queue (see Section 11.3.2).
- **submitted** - submitted to be executed by the specified job submission method, but not yet running (it may be waiting in an external batch scheduler queue, for example).
- **running** - the task "started" message received (sent when a task starts executing) but not yet finished.
- **retry delayed** - if a task configures the "retry delays" item (a list of delay intervals in minutes), then if it fails it will be put in this state for the specified interval and then resubmitted after incrementing the task try number. A task only enters the final "failed" if the last retry fails.
- two finished states:
  - **succeeded** - a *succeeded* task has finished successfully. It will remain in the suite so long as its outputs could be needed to satisfy the prerequisites of waiting tasks, and then it will be removed from the suite.
  - **failed** - a *failed* task has aborted after reporting failure. Failed tasks are not automatically removed from the suite, they must be handled by the suite operator (e.g. by retriggering after fixing the problem).

In addition, there are two held (paused) task states:

- **held** - a *held* task will not be submitted, even if it is ready to run, until it is released. Tasks that are already running cannot be held. The suite operator can deliberately hold tasks, and cylc itself puts tasks on hold under certain circumstances. For example, if the suite has been told to "shut down after currently running tasks have finished" waiting tasks will be put on hold while currently running tasks are finishing.
- **runahead** - a task in the *runahead* state has exceeded the suite runahead limit, i.e. it has got "too far ahead" of the slowest tasks and will be held back until they have caught up sufficiently. This state is automatically managed by cylc. See *The Suite Runahead Limit*, Section 11.3.1.

Finally, you may see references to the following pseudo states:

- **ready** - to set a task to *ready* means to set all of its prerequisites satisfied - i.e. ready to run (except clock-triggered tasks, which also wait on a trigger time).
- **base** (Graph Control GUI only) - off-white base graph nodes do not represent current live task proxies in the suite, they just connect the current live nodes according to the full suite graph.

11.2 Secure Passphrases, Remote Control, And Task Messaging

There are two ways in which suite interrogation and control commands, suite control GUIs, and running tasks (cylc clients) can communicate with running suites (cylc servers):

1. Pyro RPC (remote procedure call) over the network, or
2. Re-invocation of cylc commands on the suite host by passwordless ssh, so that the ultimate Pyro RPC connection only occurs on the suite host.

For the Pyro connection method to work the right network ports must be open (cylc starts grabbing ports at port 7766 by default.[15]) and there must be network routing between the involved hosts.

11.2.1 Pyro Connections and Secure Passphrases

All Pyro connections to a suite require authentication with a secure passphrase.[16] A random passphrase is generated in the suite definition directory when a suite is registered. It must be distributed to any other user account, local or remote, that hosts tasks from the suite, or from which you need to do Pyro-based remote suite monitoring or control. Passphrases can be installed into any of the following locations, in order of preference:

1. `$CYLC_SUITE_DEF_PATH/passphrase`
2. `$HOME/.cylc/SUITE_HOST/SUITE_OWNER/SUITE_NAME/passphrase`
3. `$HOME/.cylc/SUITE_HOST/SUITE_NAME/passphrase`
4. `$HOME/.cylc/SUITE_NAME/passphrase`
5. (or, optionally, given on the commandline with `-p FILE`)

The locations under `$HOME/.cylc` are suitable for remote suite control accounts - in which case the suite definition directory may not be known or accessible. If the suite definition directory is installed on a remote task host (it can be in the same location with respect to `$HOME` as the suite definition directory on the suite host, or it can be specified explicitly in the suite definition) the passphrase can go there; otherwise use one of the secondary locations listed above.

11.2.2 Ssh-Pyro Connections

For ssh-based messaging and suite control, ssh keys must be installed in the relevant accounts to allow passwordless ssh connections back to the suite host account. The suite passphrase is then only required on the suite host.

[15] The Pyro port range is configured in `$CYLC_DIR/conf/CylcGlobals.py`.
[16] Suite passphrases are never transferred across the network (a secure MD5 checksum is).

11.2.3 Choosing The Communication Method

The Pyro connection method is the default because it is simpler and more efficient, but if your remote task host does not have the right network ports open, and the IT staff are reluctant to change that, then the ssh alternative is available *so long as your local system and network configurations do not also prevent ssh connections back to the suite host - in that case see Section 11.2.4 below.*

To use the ssh-Pyro connection method:

- For remote tasks, set `ssh messaging = True` for the relevant tasks or families under the suite definition. `[runtime]` section.
- For user-invoked suite interrogation and control commands, use the `--use-ssh` command line option. Commands that normally prompt before taking action will also require use of `-f,--force`.

11.2.4 If You Cannot Use Pyro or Ssh Connections

It has come to our attention that some HPC facilities do not have any networking routing back out of the compute nodes, for security reasons and/or to avoid network chatter that could affect performance. Any kind of communication back to a cylc suite running on an external host would then presumably be impossible, but:

- You may be able to run cylc itself on one of the HPC login nodes. Depending on what software is installed on the HPC, however, this may preclude use of the cylc GUI and suite visualization tools.
- You may be able to persuade your system administrators to provide networking routing to a dedicated cylc server, if not to the wider world.
- It has been suggested that some kind of network *port forwarding* could provide a possible solution ...

11.3 Internal Queues And The Runahead Limit

Cylc suites are self-limiting to the extent that tasks are only submitted to run if they are actually ready to run. But even so some suites may generate too much activity at once, particularly in delayed/catch-up/historical operation when they are not constrained by the clock-triggered tasks that normally have to wait on real time data. There are two issues to be aware of here: overburdening an external batch queueing system, or the task host hardware, by submitting too many tasks at once - this can be prevented by cylc's *internal queues*; and overburdening cylc itself by letting unconstrained or loosely constrained tasks run off far into the future (which would require cylc to keep vast numbers of "succeeded" tasks around until other tasks, which may depend on them, have caught up) - this can be prevented by the suite runahead limit.

11.3.1 The Suite Runahead Limit

In a cylc suite each task cycles independently constrained only by dependence on other tasks or by clock triggers. A cycling task with no prerequisites and no clock-trigger will tend to run off into the future (it is unlikely that you will need such a task but it is possible to define one). Similarly, in delayed or historical operation a clock-triggered task with no dependence on other tasks will run off ahead because the clock trigger provides no constraint until the task catches up to the wall clock (or rather its cycle time catches up to the wall clock). There is no advantage in letting a few fast data retrieval tasks, say, get way ahead of the slower tasks that actually

use the data, and in large suites this could eventually swamp cylc because a large number of "succeeded" tasks would have to be maintained until the slower tasks, which might depend on them, have caught up.

To control this sort of behaviour cylc has a configurable *suite runahead limit* that prevents the fastest tasks getting too far ahead of the slowest tasks:

```
#SUITE.RC
[scheduling]
    runahead limit = 48 # (hours; default 24)
```

A cycling task spawns its successor when it enters the submitted state or, for sequential tasks, when it finishes. If the successor's cycle time is ahead of the oldest non-failed task by more than the runahead limit it is put into the special "runahead" held state until other tasks catch up sufficiently. In real time operation the runahead limit is of little consequence, because the suite will be constrained by its clock-triggered tasks, but the limit must be long enough to cover the minimum cycle time range present in the suite. A task that only runs once per day, for example, must be able to spawn 24 hours ahead.

Failed tasks, which are not automatically removed from a suite, are ignored when computing the runahead limit (but tasks that can't run because they depend on a failed task are not ignored, of course).

Note that it may occasionally be useful to define tasks that are entirely unconstrained even in real time operation. For instance, consider a tide prediction model that contributes, along with weather and sea state models, to a "seagram" product, but which does not wait on any real time data (tide is astronomically determined) and does not take inputs from other tasks in the suite. Such a task could be prevented from running off into the future by giving it artificial dependence on some other task, but the runahead limit will have the same effect.

11.3.2 Internal Queues

Large suites could potentially swamp the task host hardware or external batch queueing system, depending on the chosen job submission method, by submitting too many tasks at once. Cylc's internal queues prevent this by limiting the number of tasks, within defined groups, that are active (submitted or running) at once.

A queue is defined by a name; a *limit*, which is the maximum number of active tasks allowed for the queue; and a list of member tasks, which are assigned by name to the queue.

Queue configuration is done under the [scheduling] section of the suite.rc file, not as part of the runtime namespace hierarchy, because like dependencies queues constrain *when* a task runs rather than *what* runs after it is submitted. When runtime family relationships and queues do coincide you can assign task family members en masse to queues by using the family name, as shown in the example suite listing below.

By default every task is assigned to a *default* queue, which by default has a zero limit (interpreted by cylc as no limit). To use a single queue for the whole suite just set the default queue limit:

```
#SUITE.RC
[scheduling]
    [[ queues]]
        # limit the entire suite to 5 active tasks at once
        [[[default]]]
            limit = 5
```

To use other queues just name each one, set the limit, and assign member tasks:

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



```
        [[[q_foo]]]
            limit = 5
            members = foo, bar, baz
```

Any tasks not assigned to a particular queue will remain in the default queue. The *queues* example suite illustrates how queues work by running two task trees side by side (as seen in the graph GUI) each limited to 2 and 3 tasks respectively:

```
title = demonstrates internal queueing
description = """
Two trees of tasks: the first uses the default queue set to a limit of
two active tasks at once; the second uses another queue limited to three
active tasks at once. Run via the graph control GUI for a clear view.
              """
[scheduling]
    [[queues]]
        [[[default]]]
            limit = 2
        [[[foo]]]
            limit = 3
            members = n, o, p, fam2, u, v, w, x, y, z
    [[dependencies]]
        graph = """
            a => b & c => fam1 => h & i & j & k & l & m
            n => o & p => fam2 => u & v & w & x & y & z
                """
[runtime]
    [[fam1,fam2]]
    [[d,e,f,g]]
        inherit = fam1
    [[q,r,s,t]]
        inherit = fam2
```

Note assignment of runtime task family members to queues using the family name.

12 Other Topics In Brief

The following topics have yet to be documented in detail.

- The difference between cold-, warm-, raw-, and re-starting, a suite: see `cylc run help`.
- Intervening in suites, e.g. stopping, removing, inserting tasks: see `cylc control help`.
- Interrogating suites and tasks: see `cylc info help`, `cylc show help`, and `cylc discovery help`.
- Understanding suite evolution, particularly in delayed/catchup operation: the *Quick Start Guide* helps here, along with playing with example suites.
- Automatic state dump backups, named pre-intervention state dumps: these are used to resetart a suite from a previous state of operation. They are mentioned in the *Quick Start Guide*. Watch the suite log after intervening in a suite, to get the filenames.
- Centralized alerting and timeouts: see documentation of *task event hooks* in the *Suite.rc Reference*, Appendix A.
- Recursive purge: this is a powerful intervention but you need to understand how it works before using it. See `cylc purge help` for details.
- Handling spin-up processes via temporary tasks and adding prerequisites on-the-fly: see `cylc depend --help`, and note that when you insert a task into a running suite (a) the initial cycle time can be in the past; and (b) you can give a final cycle time after which the task will be eliminated from the suite.
- Sub-suites: to run another suite inside a task, just invoke the subsuite, with appropriate start and end cycles (probably a single cycle), in the host task's command scripting:

```
[runtime]
```



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

13 Suite Dicovery, Sharing, And Revision Control

Until release 4.2.2 cylc had a "central suite database" that users could export to and import from, for sharing suites. It was essentially just a special instance of a user suite database, held under the cylc admin account and with associated export and import commands to copy suite definition directories to and from a central store, with the suite owner's username as the first hierarchical name component. However, it was not on the network, and the suite store had to be writeable by all users and hence very insecure. Rather than develop a network server for better security and wider access, as was the original intention, it was decided to remove this functionality entirely for the following reasons:

- Large sites are likely to have suite meta-data and revision control requirements and preferences beyond what can be provided by a light-weight native cylc database.
- Small groups of light cylc users, on the other hand, can easily share suites by means of manually copying suite definitions, or with the `cylc copy` command which now supports copying registered suites between databases; they are then free to use preferred revision control tools and so on as they see fit.

We may in the future recommend particular tools that can be used for suite discovery, revision control, and so on, with cylc suites.

14 Suite Design Principles

14.1 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).

14.2 Make Tasks Rerunnable

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

14.3 Make Models Rerunnable

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

14.4 Limit Previous-Instance Dependence

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

14.5 Put Task Cycle Time In All Output File Paths

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

14.5.1 Use Cylc Cycle Time Filename Templating

The command line utility program `cylc [util] cycletime` computes offsets (in hours, days, months, and years) from a given or current (in the environment) cycle time, and optionally inserts the resulting computed cycle time, or components of it, into a given template string containing "YYYY" as a placeholder for the year value, "MM" for month, and so on. This can be used in the suite.rc environment or command scripting sections, or in task implementation scripting, to generate filenames containing the current cycle time (or some offset from it) for use by tasks.

See `cylc [util] cycletime --help` for examples.

14.6 How To Manage Input/Output File Dependencies

Dependencies between tasks usually, though not always, take the form of files generated by one task that are used by other tasks. It is possible to manage these files across a suite without hard wiring I/O locations and therefore compromising 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. Many connector tasks may be able to call the same file transfer script or command, with task-dependendent 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.

14.7 Use Generic Task Scripts

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

14.8 Make Suites Portable

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

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

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

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

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

14.9 Make Tasks As Self-Contained As Possible

Where possible, no task should rely on the action of another task, except for the prerequisites embodied in the suite dependency graph that it has no choice but to depend on. If this rule is followed, your suite will be as flexible as possible in terms of being able to run single tasks, or subsets of the suite, whilst debugging or developing new features.[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 other task; then you will be able to run single tasks without having to manually create output directories first.

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


```
# 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 #...
```

14.10 Make Suites As Self-Contained As Possible

The only compulsory content of a cylc suite definition directory is the suite.rc file (and you'll almost certainly have a suite `bin` sub-directory too). However, you can store whatever you like in a suite definition directory;[18] other files there will be ignored by cylc but suite tasks can access them via the `$CYLC_SUITE_DEF_PATH` variable that cylc automatically exports into the task execution environment. Disk space is cheap - if all programs, ancillary files, control files (etc.) required by the suite are stored in the suite definition directory instead of having the suite reference external build directories (etc.), you can turn the directory into a revision control repository and be virtually assured of the ability to exactly reproduce earlier versions, regardless of suite complexity.

14.11 Orderly Product Generation?

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

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

14.12 Clock-triggered Tasks Wait On External Data

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

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

14.13 Do Not Treat Real Time Operation As Special

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

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

In delayed operation (whether due to an actual delay in an operational suite or because you're running an historical trial) clock-triggered tasks will not constrain the suite at all, and cylc's cycle interleaving 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 appendix documents legal content of raw cylc suite.rc files. Many items have sensible default values, and most suites may only need to explicitly configure a few of them.

In addition to the configuration items described below, Jinja2 expressions can also be embedded to programmatically generate the final suite definition seen by cylc. Use of Jinja2 is documented in Section 8.6.

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

A.1 Top Level Items

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

A.1.1 title

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

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

A.1.2 description

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

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

A.2 [cylc]

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

A.2.1 [cylc] → UTC mode

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

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

A.2.2 [cylc] → simulation mode only

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

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

A.2.3 [cylc] → [[logging]]

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

A.2.3.1 [cylc] → [[logging]] → directory

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

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

A.2.3.2 [cylc] → [[logging]] → roll over at start-up

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

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

A.2.4 [cylc] → [[state dumps]]

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

A.2.4.1 [cylc] → [[state dumps]] → directory

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

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

A.2.4.2 [cylc] → [[state dumps]] → number of backups

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

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

A.2.5 [cylc] → [[event hooks]]

Cylc can call a nominated "event handler" script when certain suite events occur. This is intended to facilitate centralized alerting and automated handling of critical events. Event handlers can do anything you like, such as send emails or SMS, call pagers, or intervene in the

operation of their parent suite with cylc commands. The command `cylc [hook] email-suite` is a ready made suite event handler.

A single script can be nominated to handle a list of chosen events, and the event name is passed to the handler to allow it to distinguish between them.

Custom suite event handlers can be located in the suite bin directory. They are passed the following arguments by cylc:

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

Currently there is only one suite EVENT defined:

- 'shutdown' - the suite stopped, normally or not.

MESSAGE, if provided, describes what has happened.

Note that event handlers are called by cylc itself so if you wish to pass them additional information via the environment you must use [cylc] → [[environment]], not task runtime environments.

See also Section A.4.1.11 for task event hooks.

A.2.5.1 [cylc] → [[event hooks]] → events

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

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

A.2.5.2 [cylc] → [[event hooks]] → script

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

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

A.2.6 [cylc] → [[lockserver]]

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

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

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

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

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

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

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

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

A.2.7 [cylc] → [[environment]]

Variables defined here are exported into the environment in which cylc itself runs. They are then available to *local processes spawned directly by cylc*. Any variables read by task event handlers must be defined here, for instance, because event handlers are executed directly by cylc, not by running tasks. And similarly the command lines issued by cylc to invoke event handlers or to submit task job scripts could, in principle, make use of environment variables defined here.

A.2.7.1 Warnings

- Cylc local variables are not available to executing tasks unless you happen to choose a local direct job submission method. *Always use task runtime environments to pass variables into the task execution environment (A.4.1.12.1).*
- Unlike task execution environment variables, which are written to job scripts and interpreted by the shell at run time, cylc local environment variables are written directly to the environment by Python as literal strings *so shell variable expansion expressions cannot be used here.*

A.2.7.2 [cylc] → [[environment]] → __VARIABLE__

Replace __VARIABLE__ with any number of environment variable assignment expressions. Values may refer to other local environment variables (order of definition is preserved) and are not evaluated or manipulated by cylc, so any variable assignment expression that is legal in the shell in which cylc is running can be used (but see the warning above on variable expansions, which will not be evaluated). White space around the '=' is allowed (as far as cylc's suite.rc parser is concerned these are normal configuration items).

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

```
— FOO = $HOME/foo
```

A.2.8 [cylc] → [[simulation mode]]

Items specific to running suites in simulation mode.

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

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

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

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

The clock offset determines the initial time on the simulation clock, at suite startup, relative to the initial cycle time. An offset of 0 simulates real time operation; greater offsets simulate

catch up from a delay and subsequent transition to real time operation.

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

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

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

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

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

Configure job submission for simulation mode.

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

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

- *type:* string (a job submission method name - see Section A.4.1.9.1)
- *default:* `background`

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

Configure event hooks for simulation mode.

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

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

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

A.3 [scheduling]

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

A.3.1 [scheduling] → initial cycle time

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

- *type:* integer (YYYY[MM[DD[HH[mm[ss]]]]])
- *default:* (none)

A.3.2 [scheduling] → final cycle time

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

- *type:* integer (YYYY[MM[DD[HH[mm[ss]]]]])
- *default:* (none)

A.3.3 [scheduling] → runahead limit

The suite runahead limit prevents the fastest tasks in a suite from getting too far ahead of the slowest ones, as documented in Section 11.3.1. Tasks exceeding the limit will be put into a special runahead held state until slower tasks have caught up sufficiently. The limit must be long enough to cover the minimum cycle time range of tasks present in the suite: a task that only runs once per day, for instance, needs to spawn 24 hours ahead. Failed tasks, which are not automatically removed from a suite, are ignored when computing the runahead limit.

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

A.3.4 [scheduling] → [[queues]]

Configuration of internal queues, by which the number of simultaneously active tasks (submitted or running) can be limited, per queue. By default a single queue called *default* is defined, with all tasks assigned to it and no limit. To use a single queue for the whole suite just set the limit on the *default* queue as required. See also Section 11.3.2.

A.3.4.1 [scheduling] → [[queues]] → [[[__QUEUE__]]]

Section heading for configuration of a single queue. Replace __QUEUE__ with a queue name, and repeat the section as required.

- *type:* string
- *default:* "default"

A.3.4.2 [scheduling] → [[queues]] → [[[__QUEUE__]]] → limit

The maximum number of active tasks allowed at any one time, for this queue.

- *type:* integer
- *default:* 0 (i.e. no limit)

A.3.4.3 [scheduling] → [[queues]] → [[[__QUEUE__]]] → members

A list of member tasks, or task family names, to assign to this queue (assigned tasks will automatically be removed from the default queue).

- *type:* list of strings
- *default:* none for user-defined queues; all tasks for the "default" queue

A.3.5 [scheduling] → [[special tasks]]

This section identifies any tasks with special behaviour. By default (i.e. non "special" behaviour) tasks submit (or queue) as soon as their prerequisites are satisfied, and they spawn a successor at the next valid cycle time for the task as soon as they enter the submitted state[19]

A.3.5.1 [scheduling] → [[special tasks]] → clock-triggered

Clock-triggered tasks wait on a wall clock time specified as an offset *in hours* relative to their own cycle time, in addition to any dependence they have on other tasks. Generally speaking, only tasks that wait on external real time data need to be clock-triggered. Note that in computing the trigger time the full wall clock time and cycle time are compared, not just hours and minutes of the day, so when running a suite in catchup/delayed operation, or over historical periods, clock-triggered tasks will not constrain the suite at all until they catch up to the wall clock.

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

A.3.5.2 [scheduling] → [[special tasks]] → start-up

Start-up tasks are one-off tasks (they do not spawn a successor) that only run in the first cycle (and only in a cold-start) and any dependence on them is ignored in subsequent cycles. They can be used to prepare a suite workspace, for example, before other tasks run. Start-up tasks cannot appear in conditional trigger expressions with normal cycling tasks, because the meaning of the conditional expression becomes undefined in subsequent cycles.

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

A.3.5.3 [scheduling] → [[special tasks]] → cold-start

A cold-start task is one-off task used to satisfy the dependence of an associated task with the same cycle time, on outputs from a previous cycle - when those outputs are not available. The primary use for this is to cold-start a warm-cycled forecast model that normally depends on restart files (e.g. model background fields) generated by its previous forecast, when there is no previous forecast. This is required when cold-starting the suite, but cold-start tasks can also be inserted into a running suite to restart a model that has had to skip some cycles after running into problems. Cold-start tasks can invoke real cold-start processes, or they can just be dummy tasks that represent some external process that has to be completed before the suite is started. Unlike *start-up* tasks, dependence on cold-start tasks is preseverved in subsequent cycles so they must typically be used in OR'd conditional expressions to avoid holding up the suite.

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

A.3.5.4 [scheduling] → [[special tasks]] → sequential

By default, a task spawns a successor as soon as it is submitted to run so that successive instances of the same task can run in parallel if the opportunity arises (i.e. if their prerequisites

[19] Spawning any earlier than this brings no advantage in terms of functional parallelism and would cause uncontrolled proliferation of waiting tasks.

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 "explicit restart outputs" below).[20]

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

A.3.5.5 [scheduling] → [[special tasks]] → one-off

Synchronous one-off tasks have an associated cycle time but do not spawn a successor. Synchronous *start-up* and *cold-start* tasks are automatically one-off tasks and do not need to be listed here. Dependence on one-off tasks is not restricted to the first cycle.

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

A.3.5.6 [scheduling] → [[special tasks]] → explicit restart outputs

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

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

A.3.5.7 [scheduling] → [[special tasks]] → exclude at start-up

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

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

A.3.5.8 [scheduling] → [[special tasks]] → include at start-up

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

[20] 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.

dependency graph, in which case some manual triggering, or insertion of excluded tasks, may be required.

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

A.3.6 [scheduling] → [[dependencies]]

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

A.3.6.1 [scheduling] → [[dependencies]] → graph

The dependency graph for any one-off asynchronous (non-cycling) tasks in the suite goes here. This can be used to construct a suite of one-off tasks (e.g. build jobs and related processing) that just completes and then exits, or an initial suite section that completes prior to the cycling tasks starting (if you make the first cycling tasks depend on the last one-off ones). But note that synchronous *start-up* tasks can also be used for the latter purpose. See Section A.3.6.2.1 below for graph string syntax, and Section 8.3.

- *type:* string
- *example:* (see Section A.3.6.2.1 below)

A.3.6.2 [scheduling] → [[dependencies]] → [[[__VALIDITY__]]]

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

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

For a cycling graph with multiple validity sections for different hours of the day, the different sections *add* to generate the complete graph. Different graph sections can overlap (i.e. the same hours may appear in multiple section headings) and the same tasks may appear in multiple sections, but the individual dependencies must be unique across the entire graph. Duplicate dependencies will cause a run time error at start-up, as documented in *How Multiple Graph Strings Combine* (Section 8.3.3.2.1).

A.3.6.2.1 [scheduling] → [[dependencies]] → [[[__VALIDITY__]]] → graph

The dependency graph for the specified validity section (described just above) goes here. Syntax examples follow; see also Sections 8.3 (*Dependency Graphs*) and 8.3.4 (*Trigger Types*).

- *type:* string
- *examples:*


```
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:start => Y              # Y triggers if X starts executing
   X:fail => Y               # Y triggers if X fails
   foo[T-6]:fail => bar      # bar triggers if foo[T-6] fails
   X => !Y                   # Y suicides if X succeeds
   X | X:fail => Z           # Z triggers if X succeeds or fails
   X:finish => Z             # Z triggers if X succeeds or fails
   (A | B & C ) | D => foo   # general conditional triggers
   # comment
   """
```

- *default:* (none)

A.3.6.2.2 [scheduling] → [[dependencies]] → [[[__VALIDITY__]]] → daemon

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

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

A.4 [runtime]

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

A.4.1 [runtime] → [[__NAME__]]

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

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

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

- *legal values:*
  - `[[foo]]`
  - `[[foo, bar, baz]]`

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

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

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

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

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

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

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

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

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

Scripting defined here is put at the top of the task job script, before the task execution environment is configured, and before configuring task access to cylc. This can be used when there is a need for remotely hosted tasks to source login scripts - when Pyro has been installed locally rather than at system level, for example - because ssh does not do this for you. Use of initial scripting is not restricted to remote tasks, but pre-command or command scripting, which has access to the task execution environment, should normally be used in preference to this item.

- *type:* string
- *default:* (none)
- *example:* `initial scripting = ". $HOME/.profile"`

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

The scripting to execute when the associated task is ready to run - this can be a single command or a multiline string of scripting.

- *type:* multiline string
- *root default:* `echo "Default command scripting: sleep 10..."; sleep 10`

A.4.1.5 [runtime] → [[__NAME__]] → retry delays

A list of time intervals in minutes after which to resubmit the task if it fails. Before each try the variable `$CYLC_TASK_TRY_NUMBER` is incremented and passed to the task execution environment; this can be used to alter the behaviour at each try.

- *type:* multiline string
- *root default:* `echo "Default command scripting: sleep 10..."; sleep 10`

A.4.1.6 [runtime] → [[__NAME__]] → pre-command scripting

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

- *type:* string, or a list of strings

- *default:* (none)
- *example:*

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

A.4.1.7 [runtime] → [[__NAME__]] → post-command scripting

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

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

A.4.1.8 [runtime] → [[__NAME__]] → manual completion

If a task's initiating process detaches and exits before task processing is finished then cylc cannot arrange for the task to automatically signal when it has succeeded or failed. In such cases you must use this configuration item to tell cylc not to arrange for automatic completion messaging, and insert some minimal completion messaging yourself in appropriate places in the task implementation (see Section 9.4.4).

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

A.4.1.9 [runtime] → [[__NAME__]] → [[[job submission]]]

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

A.4.1.9.1 [runtime] → [[__NAME__]] → [[[job submission]]] → method

See *Task Job Submission* (Section 10) for how job submission works, and how to define new methods. Cylc has a number of built in job submission methods:

- *type:* string
- *legal values:*
  - *background* - direct background execution
  - *at* - the rudimentary Unix `at` scheduler
  - *loadleveler* - `llsubmit`, with directives defined in the suite.rc file
  - *pbs* - PBS `qsub`, with directives defined in the suite.rc file
  - *sge* - Sun Grid Engine `qsub`, with directives defined in the suite.rc file
- *default:* `background`

A.4.1.9.2 [runtime] → [[__NAME__]] → [[[job submission]]] → command template

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

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

A.4.1.9.3 [runtime] → [[__NAME__]] → [[[job submission]]] → shell

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

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

A.4.1.9.4 [runtime] → [[__NAME__]] → [[[job submission]]] → log directory

This is where task job scripts, and the stdout and stderr logs for local tasks, are written. The directory path may contain environment variables, including suite identity variables to make the path suite-specific (as the default value does). The job script filename is constructed, just before job submission, from the task ID and *seconds since epoch*, and then *.out* and *.err* are appended to construct the stdout and stderr log names, respectively. These filenames are thus unique even if a task gets retriggered and yet will be correctly time ordered if the log directory is listed. The filenames are also recorded by the task proxies for access via cylc commands and the suite control GUIs. Suite identity variables can be used in the path, as in the default value, but *not task identity variables* such as `$CYLC_TASK_NAME` and `$CYLC_TASK_CYCLE_TIME`, because the log directory is created before the task runs.

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

A.4.1.9.5 [runtime] → [[__NAME__]] → [[[job submission]]] → work directory

Task command scripting is executed from within a work directory created on the fly, if necessary, by the task's job script. In non-detaching tasks the work directory is automatically removed again *if it is empty* before the job script exits. The work directory can be accessed by tasks via the environment variable `$CYLC_TASK_WORK_PATH`.

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

A.4.1.9.6 [runtime] → [[__NAME__]] → [[[job submission]]] → share directory

Like task work directories (above) this directory is created on the fly, if necessary, by the job script. It is intended as a shared data area for multiple tasks on the same host, but as for any task runtime config item it can be specialized to particular tasks or groups of tasks. It can be accessed by tasks at run time via the environment variable `$CYLC_SUITE_SHARE_PATH`.

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

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

Remote hosting configuration for tasks that run on hosts other than the suite host. If a remote host is specified cylc will attempt to execute the task on that host by passwordless ssh. Cylc must be installed on remote task hosts, but of the external software dependencies only Pyro

is required there (and actually not even that - see "ssh messaging" below). Passwordless ssh must be configured between the local suite owner on the suite host, and the task owner on the remote task host.

A.4.1.10.1 [runtime] → [[__NAME__]] → [[[remote]]] → host

The remote host for this task or family.

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

A.4.1.10.2 [runtime] → [[__NAME__]] → [[[remote]]] → owner

The task owner username. This is (only) used in the passwordless ssh command line invoked by cylc to submit the remote task (consequently it may be defined using local environment variables (i.e. the shell in which cylc runs, and [cylc] → [[environment]]).

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

A.4.1.10.3 [runtime] → [[__NAME__]] → [[[remote]]] → cylc directory

The path to the remote cylc installation, required if cylc is not in the default search path on the remote host.

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

A.4.1.10.4 [runtime] → [[__NAME__]] → [[[remote]]] → suite definition directory

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

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

A.4.1.10.5 [runtime] → [[__NAME__]] → [[[remote]]] → remote shell template

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

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

A.4.1.10.6 [runtime] → [[__NAME__]] → [[[remote]]] → log directory

This log directory is used for the stdout and stderr logs of remote tasks. The directory will be created on the fly if necessary. If not specified, the local job submission log path will be used (see A.4.1.9.4) with the suite owner's home directory path, if present, replaced by `'$HOME'` for interpretation on the remote host. The stdout and stderr log file names are the same as for local tasks, and are recorded by the task proxies for access via gcylc. Suite identity

variables can be used in the path, but *not task identity variables* such as `$CYLC_TASK_NAME` and `$CYLC_TASK_CYCLE_TIME`, because the log directory is created before the task runs.

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

A.4.1.10.7 [runtime] → [[__NAME__]] → [[[remote]]] → work directory

Use this item if you need to override the local task work directory (see A.4.1.9.5). If omitted, the local directory will be used with the suite owner's home directory path, if present, replaced by `'$HOME'` for interpretation on the remote host.

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

A.4.1.10.8 [runtime] → [[__NAME__]] → [[[remote]]] → share directory

Use this item if you need to override the local share directory (see A.4.1.9.6). If omitted, the local directory will be used with the suite owner's home directory path, if present, replaced by `'$HOME'` for interpretation on the remote host.

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

A.4.1.10.9 [runtime] → [[__NAME__]] → [[[remote]]] → ssh messaging

If your network configuration or firewall blocks the TCP/IP sockets required for remote tasks to communicate with their parent suite you can tell cylc to use passwordless ssh (from remote host to suite host) instead, to invoke local messaging commands on the suite host.

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

This item affects the behaviour of the cylc messaging commands by means of the task execution environment, so no special cylc configuration is required on the remote host itself. Eventually it may be moved to a site and host configuration file (yet to be implemented) because it is host- rather than task-specific. For the moment though you can still set it just once in a namespace inherited by all tasks on the affected host.

Note that you can use a remote section with this item in it even for tasks that are local as far as cylc is concerned, but which end up running on the affected remote host due to the action of the local batch queueing system or resource manager.

A.4.1.11 [runtime] → [[__NAME__]] → [[[event hooks]]]

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

Currently a single script can be nominated to handle a list of chosen events, and the event name is passed to the handler to allow it to distinguish between them (cylc once allowed distinct handlers for each event but this required verbose configuration for little gain).

Custom task event handlers can be located in the suite bin directory. They are passed the following arguments by cylc:

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

where EVENT is one of the following strings:

- 'submitted' - the task was submitted
- 'started' - the task started running
- 'succeeded' - the task succeeded
- 'submission_failed' - the task failed in job submission
- 'warning' - the task reported a warning message
- 'failed' - the task failed
- 'submission_timeout' - task job submission timed out
- 'execution_timeout' - task execution timed out

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

To configure timeout handling, list 'submission_timeout' and/or 'execution_timeout' in the events to handle, and then specify corresponding timeout values in minutes using the timeout config items below.

Note that event handlers are called by cylc itself, not by the running tasks so if you wish to pass them additional information via the environment you must use [cylc] → [[environment]], not task runtime environments.

See also Section A.2.5 for suite event hooks.

A.4.1.11.1 [runtime] → [[__NAME__]] → [[[event hooks]]] → events

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

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

A.4.1.11.2 [runtime] → [[__NAME__]] → [[[event hooks]]] → script

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

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

A.4.1.11.3 [runtime] → [[__NAME__]] → [[[event hooks]]] → submission timeout

If a task has not started the specified number of minutes after it was submitted, the event handler will be called by cylc with *submission_timeout* as the EVENT argument:

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

A.4.1.11.4 [runtime] → [[__NAME__]] → [[[event hooks]]] → execution timeout

If a task has not finished the specified number of minutes after it started running, the event handler will be called by cylc with *execution_timeout* as the EVENT argument:

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

A.4.1.11.5 [runtime] → [[__NAME__]] → [[[event hooks]]] → reset timer

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

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

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

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

A.4.1.12.1 [runtime] → [[__NAME__]] → [[[environment]]] → __VARIABLE__

Replace __VARIABLE__ with any number of environment variable assignment expressions. Order of definition is preserved so values can refer to previously defined variables. Values are passed through to the task job script without evaluation or manipulation by cylc, so any variable assignment expression that is legal in the job submission shell can be used. White space around the '=' is allowed (as far as cylc's suite.rc parser is concerned these are just normal configuration items).

- *type:* string
- *default:* (none)
- *legal values:* depends to some extent on the task job submission shell (Section A.4.1.9.3).
- *examples*, for the bash shell:

```
— FOO = $HOME/bar/baz
— BAR = ${FOO}$GLOBALVAR
— BAZ = $( echo "hello world")
— WAZ = ${FOO%.jpg}.png
— NEXT_CYCLE = $( cylc cycletime -a 6 )
— PREV_CYCLE = `cylc cycletime -s 6`
— ZAZ = "${FOO#bar}"#<-- QUOTED to escape the suite.rc comment character
```

A.4.1.13 [runtime] → [[__NAME__]] → [[[directives]]]

Batch queue scheduler directives. Whether or not these are used depends on the job submission method. For the built-in loadleveler, pbs, and sge methods directives are written to the top of the task job script in the correct format for the method. Specifying directives individually like this allows use of default directives that can be individually overridden at lower levels of the runtime namespace hierarchy.

A.4.1.13.1 [runtime] → [[__NAME__]] → [[[directives]]] → __DIRECTIVE__

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

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

Example directives for the built-in job submission methods are shown in Section 10.2.

A.4.1.14 [runtime] → [[__NAME__]] → [[[outputs]]]

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

A.4.1.14.1 [runtime] → [[__NAME__]] → [[[outputs]]] → __OUTPUT__

Replace __OUTPUT__ with any number of labelled output messages.

- *type:* string (a message containing `<CYLC_TASK_CYCLE_TIME>` with an optional offset as shown below. Note that you cannot use the corresponding shell variable `$CYLC_TASK_CYCLE_TIME` here. The string substitution (replacing this special variable with the actual task cycle time) is done inside cylc, not in the task execution environment.
- *default:* (none)
- *examples:*

```
foo = "sea state products ready for <CYLC_TASK_CYCLE_TIME>"
bar = "nwp restart files ready for <CYLC_TASK_CYCLE_TIME+6>"
```

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

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

A.5 [visualization]

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

A.5.1 [visualization] → initial cycle time

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

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

A.5.2 [visualization] → final cycle time

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

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

A.5.3 [visualization] → collapsed families

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

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

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

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

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

A.5.5 [visualization] → use node color for labels

Graph node labels can be printed in the same color as the node outline.

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

A.5.6 [visualization] → default node attributes

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

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

A.5.7 [visualization] → default edge attributes

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

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

A.5.8 [visualization] → enable live graph movie

If True, the graph-based suite control GUI will write out a dot-language graph file on every change; these can be post-processed into a movie showing how the suite evolves. The frames will be written to the run time graph directory (see below).

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

A.5.9 [visualization] → [[node groups]]

Define named groups of graph nodes (tasks and families) which can styled en masse, by name, in [visualization] → [[node attributes]]. Node groups are automatically defined for all task families, including root, so you can style family and member nodes at once by family name.

A.5.9.1 [visualization] → [[node groups]] → __GROUP__

Replace __GROUP__ with each named group of tasks or families.

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

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

A.5.10 [visualization] → [[node attributes]]

Here you can assign graph node attributes to specific nodes, or to all members of named groups defined in [visualization] → [[node groups]] (task families are automatically node groups). Group styling can be overridden for individual nodes or subgroups.

A.5.10.1 [visualization] → [[node attributes]] → __NAME__

Replace __NAME__ with each node or node group for style attribute assignment.

- *type:* list of quoted `'attribute=value'` pairs
- *legal values:* see graphviz or pygraphviz documentation
- *default:* (none)
- *example:* (with reference to the node groups defined above)

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

A.5.11 [visualization] → [[run time graph]]

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

A.5.11.1 [visualization] → [[run time graph]] → enable

Run time graphing is now disabled by default.

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

A.5.11.2 [visualization] → [[run time graph]] → cutoff

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

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

A.5.11.3 [visualization] → [[run time graph]] → directory

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

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

A.6 Special Placeholder Variables

A small number of special variables are used as placeholders in cylc suite definitions:

- `<CYLC_TASK_CYCLE_TIME>` and `<CYLC_TASK_CYCLE_TIME+/-offset>`
  These are replaced with the actual cycle time of the task (with optional computed offset) in the message strings registered for explicit internal (pre-completion) task outputs. They correspond to `$CYLC_TASK_CYCLE_TIME` in the task execution environment.
- `<TASK>` and `<NAMESPACE>`
  These placeholder variables are replaced with actual task or namespace names in runtime settings. The difference between the two is explained in Section 8.4.3, *Defining Multiple Namespaces At Once*.
- `<ASYNCID>`
  This placeholder variable is replaced with the actual "asynchronous ID" (e.g. satellite pass ID) for repeating asynchronous tasks.

To use real (programming language) variables in suite definitions, use the Jinja2 template processor (Section 8.6).

A.7 Default Suite Configuration

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

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

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

[cylc]
    UTC mode = False  # DEFAULT
    simulation mode only = False  # DEFAULT
    [[logging]]
        directory = '$HOME/cylc-run/$CYLC_SUITE_REG_NAME/log/suite' # DEFAULT
        roll over at start-up = True  # DEFAULT
    [[state dumps]]
        directory = '$HOME/cylc-run/$CYLC_SUITE_REG_NAME/state' # DEFAULT
        number of backups = 10  # DEFAULT
    [[event hooks]]
        script = None
        events = (none)
    [[lockserver]]
        enable = False # DEFAULT
        simultaneous instances = False # DEFAULT
    [[environment]]
        # (none)
    [[simulation mode]]
        clock offset = 24  # DEFAULT
        clock rate = 10  # DEFAULT
        command scripting = "echo SIMULATION MODE; sleep $CYLC_TASK_DUMMY_RUN_LENGTH"  # DEFAULT
```



```
        retry delays = (none)
        [[[job submission]]]
            method = background  # DEFAULT
        [[[event hooks]]]
            enable = False # DEFAULT
[scheduling]
    initial cycle time = None
    final cycle time = None
    runahead limit = (largest minimum runahead limit of each cycler)
                     # DEFAULT via config.py
    [[queues]]
        [[[default]]]
            limit = 0 (no limit) # DEFAULT
            members = (all tasks) # DEFAULT
    [[special tasks]]
        clock-triggered = (none)
        start-up = (none)
        cold-start = (none)
        sequential = (none)
        one-off = (none)
        explicit restart outputs = (none)
        exclude at start-up = (none)
        include at start-up = (none)
    [[dependencies]]
        graph = None
        [[[__many__]]]
            graph = None
            daemon = None
[runtime]
    [[root]]
        inherit = None # (other namespaces inherit first from root)
        description = "No description provided"  # DEFAULT
        initial scripting = None
        pre-command scripting = None
        command scripting = "echo Dummy command scripting; sleep $CYLC_TASK_DUMMY_RUN_LENGTH" # DEFAULT
        retry delays = (none)
        post-command scripting = None
        manual completion = False  # DEFAULT
        [[[job submission]]]
            method = background  # DEFAULT
            command template = None
            shell = /bin/bash  # DEFAULT
            log directory = '$HOME/cylc-run/$CYLC_SUITE_REG_NAME/log/job'  # DEFAULT
            share directory = '$CYLC_SUITE_DEF_PATH/share' # DEFAULT
            work directory = '$CYLC_SUITE_DEF_PATH/work/$CYLC_TASK_ID' # DEFAULT
        [[[remote]]]
            host = None
            owner = None
            cylc directory = None
            suite definition directory = (local directory with '$HOME' replaced) # DEFAULT
            log directory  = (local directory with '$HOME' replaced) # DEFAULT
            share directory = (local directory with '$HOME' replaced) # DEFAULT
            work directory = (local directory with '$HOME' replaced) # DEFAULT
            remote shell template = 'ssh -oBatchMode=yes %s' # DEFAULT
            ssh messaging = False # DEFAULT
        [[[event hooks]]]
            script = None
            events = (none)
            submission timeout = None
            execution timeout = None
            reset timer = False  # DEFAULT
        [[[environment]]]
            CYLC_TASK_DUMMY_RUN_LENGTH = 10 # DEFAULT
            # (SUITE AND TASK IDENTITY VARIABLES ARE PROVIDED)
        [[[directives]]]
            # (none)
        [[[outputs]]]
            # (none)

[visualization]
```



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


B Command Reference

```
Cylc ("silk") is a suite engine and metascheduler that specializes in
cycling weather and climate forecasting suites and related processing
(but it can also be used for one-off workflows of non-cycling tasks).

Version: cylc-4.5.0

This is the Command Line Interface; the Graphical User Interface is
"gcylc" (a.k.a. "cylc gui"). Control GUIs for particular suites can be
invoked directly with "gcontrol SUITE" (a.k.a. "cylc gcontrol SUITE").

USAGE:
  % cylc -v,--version                   # print cylc version
  % cylc help,--help,-h,?               # print this help page
  % cylc -V,--Version=avail[able]       # print available cylc versions
  % cylc -V,--Version=x.y.z             # reinvoke with another version

  % cylc help CATEGORY                  # print help by category
  % cylc CATEGORY help                  # (ditto)

  % cylc help [CATEGORY] COMMAND        # print command help
  % cylc [CATEGORY] COMMAND help,--help # (ditto)

  % cylc [CATEGORY] COMMAND [options] SUITE [arguments]
  % cylc [CATEGORY] COMMAND [options] SUITE TASK [arguments]

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

SUITE NAMES AND DATABASES
  Suites are addressed by hierarchical names (e.g. suite1, nwp.oper,
nwp.test.LAM2, etc.) registered in a suite database that associates
suite the names with suite definition directory locations. Your default
database is located at $HOME/.cylc/DB. The '--db=' command option can be
used to work with suites registered in other databases. You should
be able to read and graph (etc.) another user's suite definitions, from
their default database, with the '--db=u:USER' option (access is
governed by normal filesystem permissions).

REMOTE COMMAND RE-INVOCATION BY SSH
  If passwordless ssh is configured appropriately use of the '--host='
and '--owner=' command options will result in most cylc commands being
re-invoked on another host or user account. With the '--use-ssh' flag
this also goes for suite control commands which normally use direct
network RPC instead of re-invocation - see below.

REMOTE SUITE CONTROL BY RPC (PYRO)
  Cylc commands, GUIs, and tasks that interact with running suites use a
network RPC protocol (Pyro) with passphrase authentication. The '--host'
and '--owner' command options will target a suite running on another
host or user account. The suite passphrase, generated in the suite
definition directory at registration time, should be installed on a
remote control host in one of the following locations:
  1) $HOME/.cylc/<HOST>/<OWNER>/<SUITE>/passphrase,
  2) $HOME/.cylc/<HOST>/<SUITE>/passphrase, or
  3) $HOME/.cylc/<SUITE>/passphrase).
  4) Or specify the location explicitly with '--p','--passphrase='.
A running task first looks in the suite definition directory for the
passphrase, then the above locations. Likewise a remote task checks
the remote suite definition directory first, if one is configured.

REMOVE SUITE CONTROL BY SSH RE-INVOCATION
  If the network ports required by Pyro are not open you can use the
```



```
'--use-ssh' flag to re-invoke suite control commands on the suite host.
For remote tasks the same can be achieved, in the suite definition, by:
  [runtime]->[[NAMESPACE]]->[[[remote]]]->'ssh messaging = True'
then the passphrase is not needed on the remote host because the RPC
connection is only made on the suite host (ssh keys are needed though).

TASK IDENTIFICATION
  Tasks are identified by NAME%TAG where for cycling tasks TAG is a
cycle time (YYYY[MM[DD[HH[mm[ss]]]]]) and for asynchronous tasks TAG is
an integer (just '1' for one-off asynchronous tasks). On the command
line an asynchronous tag must be preceded by "a:" (a:1).

HOW TO DRILL DOWN TO COMMAND USAGE HELP:
  % 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 command set.
  db|database ... Suite registration, copying, deletion, etc.
  preparation ... Suite editing, validation, visualization, etc.
  information ... Interrogate suite definitions and running suites.
  discovery ..... Detect running suites.
  control ....... Suite start up, monitoring, and control.
  utility ....... Cycle arithmetic and templating, housekeeping, etc.
  task .......... The task messaging interface.
  hook .......... Suite and task event hook scripts.
  admin ......... Cylc installation, testing, and example suites.
  license|GPL ... Software licensing information (GPL v3.0).
```

B.1 Command Categories

B.1.1 admin

```
CATEGORY: admin - Cylc installation, testing, and example suites.

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

COMMANDS:
  check-examples .... (ADMIN) Check all example suites validate
  import-examples ... (ADMIN) import example suites your user database
  test-db ........... (ADMIN) Automated suite database test
  test-suite ........ (ADMIN) Automated cylc scheduler test
```

B.1.2 all

```
CATEGORY: all - The complete command set.

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

COMMANDS:
  alias ...................... Register an alternative name for a suite
  block ...................... Tell suites to ignore interventions until unblocked
  check-examples ............. (ADMIN) Check all example suites validate
  checkvars .................. Check required environment variables en masse
  conditions ................. Print the GNU General Public License v3.0
  copy|cp .................... Copy a suite or a group of suites
  cycletime .................. Cycle time arithmetic and filename templating
  depend ..................... Add prerequisites to tasks in a running suite
  diff|compare ............... Compare two suite definitions and print differences
  documentation|browse ....... Display cylc documentation (User Guide etc.)
  dump ....................... Print the state of tasks in a running suite
  edit ....................... Edit suite definitions, optionally inlined
  failed|task-failed ......... Release task lock and report failure
  gcontrol ................... Suite Control GUI (a.k.a. gcontrol)
```



```
  get-config ................. Parse a suite and report configuration values
  get-directory .............. Retrieve suite definition directory paths
  graph ...................... Plot suite dependency graphs and runtime hierarchies
  gui|gcylc .................. Main Graphical User Interface (a.k.a. gcylc)
  hold ....................... Hold (pause) suites or individual tasks
  housekeeping ............... Parallel archiving and cleanup on cycle time offsets
  import-examples ............ (ADMIN) import example suites your user database
  insert ..................... Insert tasks into a running suite
  jobscript .................. Generate a task job script and print it to stdout
  list|ls .................... Print suite tasks and runtime hierarchies
  lockclient|lc .............. Manual suite and task lock management
  lockserver ................. The cylc lockserver daemon
  log ........................ Print or view filtered suite logs
  message|task-message ....... Report progress and completion of outputs
  monitor .................... An in-terminal suite monitor (see also gcylc)
  nudge ...................... Cause the cylc task processing loop to be invoked
  ping ....................... Check that a suite is running
  print ...................... Print registered suites
  purge ...................... Remove task trees from a running suite
  refresh .................... Report invalid registrations and update suite titles
  register ................... Register a suite for use
  release|unhold ............. Release (unpause) suites or individual tasks
  remove|kill ................ Remove tasks from a running suite
  reregister|rename .......... Change the name of a suite
  reset ...................... Manually set tasks to the waiting, ready, or succeeded states
  restart .................... Restart a suite from a previous state
  run|start .................. Start a suite at a given cycle time
  scan ....................... Scan a host for running suites and lockservers
  scp-transfer ............... Scp-based file transfer for cylc suites
  search|grep ................ Search in suite definitions
  set-runahead ............... Change the runahead limit in a running suite.
  set-verbosity .............. Change a running suite's logging verbosity
  show ....................... Print task state (prerequisites and outputs etc.)
  started|task-started ....... Acquire a task lock and report started
  stop|shutdown .............. Shut down running suites
  submit|single .............. Run a single task just as its parent suite would
  succeeded|task-succeeded ... Release task lock and report succeeded
  test-db .................... (ADMIN) Automated suite database test
  test-suite ................. (ADMIN) Automated cylc scheduler test
  trigger .................... Manually trigger or re-trigger a task
  unblock .................... Tell a blocked suite to resume complying with interventions
  unregister ................. Unregister and optionally delete suites
  validate ................... Parse and validate suite definitions
  view ....................... View suite definitions, inlined and Jinja2 processed
  warranty ................... Print the GPLv3 disclaimer of warranty
```

B.1.3 control

```
CATEGORY: control - Suite start up, monitoring, and control.

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

COMMANDS:
  block ............ Tell suites to ignore interventions until unblocked
  depend ........... Add prerequisites to tasks in a running suite
  gcontrol ......... Suite Control GUI (a.k.a. gcontrol)
  gui|gcylc ........ Main Graphical User Interface (a.k.a. gcylc)
  hold ............. Hold (pause) suites or individual tasks
  insert ........... Insert tasks into a running suite
  nudge ............ Cause the cylc task processing loop to be invoked
  purge ............ Remove task trees from a running suite
  release|unhold ... Release (unpause) suites or individual tasks
  remove|kill ...... Remove tasks from a running suite
  reset ............ Manually set tasks to the waiting, ready, or succeeded states
  restart .......... Restart a suite from a previous state
  run|start ........ Start a suite at a given cycle time
  set-runahead ..... Change the runahead limit in a running suite.
  set-verbosity .... Change a running suite's logging verbosity
```

```
  stop|shutdown .... Shut down running suites
  trigger .......... Manually trigger or re-trigger a task
  unblock .......... Tell a blocked suite to resume complying with interventions
```

B.1.4 database

```
CATEGORY: db|database - Suite registration, copying, deletion, etc.

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

COMMANDS:
  alias ............... Register an alternative name for a suite
  copy|cp ............. Copy a suite or a group of suites
  get-directory ....... Retrieve suite definition directory paths
  gui|gcylc ........... Main Graphical User Interface (a.k.a. gcylc)
  print ............... Print registered suites
  refresh ............. Report invalid registrations and update suite titles
  register ............ Register a suite for use
  reregister|rename ... Change the name of a suite
  unregister .......... Unregister and optionally delete suites
```

B.1.5 discovery

```
CATEGORY: discovery - Detect running suites.

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 hook

```
CATEGORY: hook - Suite and task event hook scripts.

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

COMMANDS:
  email-suite ... A suite event hook script that sends email alerts
  email-task .... A task event hook script that sends email alerts
```

B.1.7 information

```
CATEGORY: information - Interrogate suite definitions and running suites.

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

COMMANDS:
  documentation|browse ... Display cylc documentation (User Guide etc.)
  dump ................... Print the state of tasks in a running suite
  get-config ............. Parse a suite and report configuration values
  gui|gcylc .............. Main Graphical User Interface (a.k.a. gcylc)
  list|ls ................ Print suite tasks and runtime hierarchies
  log .................... Print or view filtered suite logs
  monitor ................ An in-terminal suite monitor (see also gcylc)
  nudge .................. Cause the cylc task processing loop to be invoked
  show ................... Print task state (prerequisites and outputs etc.)
```

B.1.8 license

```
CATEGORY: license|GPL - Software licensing information (GPL v3.0).

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

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

B.1.9 preparation

```
CATEGORY: preparation - Suite editing, validation, visualization, etc.

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

COMMANDS:
  diff|compare ... Compare two suite definitions and print differences
  edit ........... Edit suite definitions, optionally inlined
  graph .......... Plot suite dependency graphs and runtime hierarchies
  gui|gcylc ...... Main Graphical User Interface (a.k.a. gcylc)
  jobscript ...... Generate a task job script and print it to stdout
  list|ls ........ Print suite tasks and runtime hierarchies
  search|grep .... Search in suite definitions
  validate ....... Parse and validate suite definitions
  view ........... View suite definitions, inlined and Jinja2 processed
```

B.1.10 task

```
CATEGORY: task - The task messaging interface.

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

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

B.1.11 utility

```
CATEGORY: utility - Cycle arithmetic and templating, housekeeping, etc.

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

COMMANDS:
  checkvars ....... Check required environment variables en masse
  cycletime ....... Cycle time arithmetic and filename templating
  housekeeping .... Parallel archiving and cleanup on cycle time offsets
  lockclient|lc ... Manual suite and task lock management
  lockserver ...... The cylc lockserver daemon
  scp-transfer .... Scp-based file transfer for cylc suites
```

B.2 Commands

B.2.1 alias

```
Usage: cylc [db] alias [OPTIONS] REG1 REG2

Register an alias REG2 for suite REG1. Using an alias is equivalent to
using the full suite name, except for the following caveat: aliases are
```

```
stored in your local suite db and aliased suites run under their full
name; therefore you can't interact with remote suites via an alias
unless you use '--use-ssh' (for [control] category commands), which
re-invokes the control command on the remote suite host (where the alias
is known).

  $ cylc alias global.ensemble.parallel.test3 bob
  $ cylc edit bob
  $ cylc run  bob
  $ cylc show bob # etc.

Arguments:
   REG1                 Target suite name
   REG2                 An alias for REG1

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
```

B.2.2 block

```
Usage: cylc [control] block [OPTIONS] REG

A blocked suite refuses to comply with intervention commands until deliberately
unblocked. This is a crude safety device to guard against accidental
intervention in your own suites (if you are running multiple suites at once
a simple typo on the command line could target the wrong suite).

Arguments:
   REG                  Suite name

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.3 check-examples

```
USAGE: cylc [admin] check-examples

Check that all cylc example suites validate successfully.
```

B.2.4 checkvars

```
Usage: cylc checkvars [OPTIONS] VARNAMES

Check that each member of a list of environment variables is defined,
and then optionally check their values according to the chosen
```



```
commandline option. Note that THE VARIABLES MUST BE EXPORTED AS THIS
SCRIPT NECESSARILY EXECUTES IN A SUBSHELL.

All of the input variables are checked in turn and the results printed.
If any problems are found then, depending on use of '-w,--warn-only',
this script either aborts with exit status 1 (error) or emits a stern
warning and exits with status 0 (success).

Arguments:
   VARNAMES     Space-separated list of environment variable names.

Options:
  -h, --help            show this help message and exit
  -d, --dirs-exist      Check that the variables refer to directories that
                        exist.
  -c, --create-dirs     Attempt to create the directories referred to by the
                        variables, if they do not already exist.
  -p, --create-parent-dirs
                        Attempt to create the parent directories of files
                        referred to by the variables, if they do not already
                        exist.
  -f, --files-exist     Check that the variables refer to files that exist.
  -i, --int             Check that the variables refer to integer values.
  -s, --silent          Do not print the result of each check.
  -w, --warn-only       Print a warning instead of aborting with error status.
```

B.2.5 conditions

```
USAGE: cylc [license] warranty [--help]
Cylc is release under the GNU General Public License v3.0
This command prints the GPL v3.0 license in full.

Options:
  --help   Print this usage message.
```

B.2.6 copy

```
Usage: cylc [db] copy|cp [OPTIONS] REG REG2 TOPDIR

Copy suite or group REG to TOPDIR, and register the copy as REG2.

Consider the following three suites:

% cylc db print '^foo'     # printed in flat form
foo.bag     | "Test Suite Zero" | /home/bob/zero
foo.bar.qux | "Test Suite Two"  | /home/bob/two
foo.bar.baz | "Test Suite One"  | /home/bob/one

% cylc db print -t '^foo'  # printed in tree from
foo
 |-bag    "Test Suite Zero" | /home/bob/zero
 `-bar
   |-baz  "Test Suite One"  | /home/bob/one
   `-qux  "Test Suite Two"  | /home/bob/two

These suites are stored in a flat directory structure under /home/bob,
but they are organised in the suite database as a group 'foo' that
contains the suite 'foo.bag' and a group 'foo.bar', which in turn
contains the suites 'foo.bar.baz' and 'foo.bar.qux'.

When you copy suites with this command, the target registration names
are determined by TARGET and the name structure underneath SOURCE, and
the suite definition directories are copied into a directory tree under
TOPDIR whose structure reflects the target registration names. If this
is not what you want, you can copy suite definition directories manually
and then register the copied directories manually with 'cylc register'.

To copy suites between different databases use one or both of the
--db-to, --db-from options.  If only one is used the other database
```

```
(source or target) will be the default database, which may in turn
be specified with the plain --db option.

EXAMPLES (using the three suites above):

% cylc db copy foo.bar.baz red /home/bob       # suite to suite
  Copying suite definition for red
% cylc db print "^red"
  red | "Test Suite One" | /home/bob/red

% cylc copy foo.bar.baz blue.green /home/bob   # suite to group
  Copying suite definition for blue.green
% cylc db pr "^blue"
  blue.green | "Test Suite One" | /home/bob/blue/green

% cylc copy foo.bar orange /home/bob           # group to group
  Copying suite definition for orange.qux
  Copying suite definition for orange.baz
% cylc db pr "^orange"
  orange.qux | "Test Suite Two" | /home/bob/orange/qux
  orange.baz | "Test Suite One" | /home/bob/orange/baz

Arguments:
   REG                  Source suite name
   REG2                  Target suite name
   TOPDIR                  Top level target directory.

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  --db-from=DB    Source suite database, specified as for --db. Use --db-to,
                  or --db, or your default DB as the target.
  --db-to=DB      Target suite database, specified as for --db. Use --db-from,
                  or --db', or your default DB as the source.
```

B.2.7 cycletime

```
Usage: cylc [util] cycletime [OPTIONS] [CYCLE]

Arithmetic cycle time offset computation, and filename templating.

Examples:

1) print offset from an explicit cycle time:
  % cylc [util] cycletime --offset-hours=6 2010082318
  2010082400

2) print offset from $CYLC_TASK_CYCLE_TIME (as in suite tasks):
  % export CYLC_TASK_CYCLE_TIME=2010082318
  % cylc cycletime --offset-hours=-6
  2010082312

3) cycle time filename templating, explicit template:
  % export CYLC_TASK_CYCLE_TIME=201008
  % cylc cycletime --offset-years=2 --template=foo-YYYYMM.nc
  foo-201208.nc

4) cycle time filename templating, template in a variable:
  % export CYLC_TASK_CYCLE_TIME=201008
  % export MYTEMPLATE=foo-YYYYMM.nc
  % cylc cycletime --offset-years=2 --template=MYTEMPLATE
  foo-201208.nc
```



```
Arguments:
   [CYCLE]    YYYY[MM[DD[HH[mm[ss]]]]], default $CYLC_TASK_CYCLE_TIME

Options:
  -h, --help            show this help message and exit
  --offset-hours=HOURS  Add N hours to CYCLE (may be negative)
  --offset-days=DAYS    Add N days to CYCLE (N may be negative)
  --offset-months=MONTHS
                        Add N months to CYCLE (N may be negative)
  --offset-years=YEARS  Add N years to CYCLE (N may be negative)
  --template=TEMPLATE   Filename template string or variable
  --print-year          Print only YYYY of result
  --print-month         Print only MM of result
  --print-day           Print only DD of result
  --print-hour          Print only HH of result
```

B.2.8 depend

```
Usage: cylc [control] depend [OPTIONS] REG TASK DEP

Add new dependencies on the fly to tasks in running suite REG. If DEP
is a task ID the target TASK will depend on that task finishing,
otherwise DEP can be an explicit quoted message such as
  "Data files uploaded for 2011080806"
(presumably there will be another task in the suite, or you will insert
one, that reports that message as an output).

Prerequisites added on the fly are not propagated to the successors
of TASK, and they will not persist in TASK across a suite restart.

Arguments:
   REG                  Suite name
   TASK                  Target task
   DEP                  New dependency

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.9 diff

```
Usage: cylc [prep] diff|compare [OPTIONS] REG REG2

Compare two suite definitions and display any differences.

Differencing is done after parsing the suite.rc files so it takes
account of default values that are not explicitly defined, it disregards
the order of configuration items, and it sees any include-file content
after inlining has occurred.

Note that seemingly identical suites normally differ due to inherited
default configuration values (e.g. the default job submission log
directory.

Files in the suite bin directory and other sub-directories of the
```

```
suite definition directory are not currently differenced.

Arguments:
   REG1                 Suite name
   REG1                 Suite name

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  -n, --nested    print suite.rc section headings in nested form.
```

B.2.10 documentation

```
Usage: cylc [info] documentation|browse [OPTIONS]

Display cylc documentation:
 1/ The User Guide in PDF format (default)
 2/ The User Guide in HTML format
 3/ The change log (plain text format)
 4/ The cylc internet homepage.

Required environment variables:
  $PDF_READER    (e.g. evince)
  $HTML_READER   (e.g. firefox)
  $EDITOR        (e.g. vim)
  $GEDITOR       (e.g. 'gvim -f')


Arguments:

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  --pdf           Display the PDF User Guide (DEFAULT).
  --html          Display the multi-page HTML User Guide.
  --html-single   Display the single page HTML User Guide.
  --www           Display the cylc internet homepage.
  --log           Display the cylc repository change log.
  -g              Use $GEDITOR instead of $EDITOR to view the change log.
```

B.2.11 dump

```
Usage: cylc [info] dump [OPTIONS] REG

Print current suite state information (e.g. the state of each task). For
small suites 'watch cylc [info] dump SUITE' is an effective non-GUI real
time monitor (but see also 'cylc monitor').

For more information about a specific task, such as the current state of
its prerequisites and outputs, see 'cylc [info] show'.

Examples:
 Display the state of all running tasks, sorted by cycle time:
 % cylc [info] dump --tasks --sort SUITE | grep running
```

```
 Display the state of all tasks in a particular cycle:
 % cylc [info] dump -t SUITE | grep 2010082406

Arguments:
   REG                   Suite name

Options:
  -h, --help             show this help message and exit
  --owner=USER           User account name (defaults to $USER).
  --host=HOST            Host name (defaults to localhost).
  -v, --verbose          Verbose output mode.
  --debug                Turn on exception tracebacks.
  --db=DB                Suite database: 'u:USERNAME' for another user's
                         default database, or PATH to an explicit location.
                         Defaults to $HOME/.cylc/DB.
  -o, --override         Override cylc version compatibilty checking.
  --use-ssh              Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                         Network connection timeout in case of hung ports
                         (default 1 second).
  -p FILE, --passphrase=FILE
                         Suite passphrase file (if not in a default location)
  -g, --global           Global information only.
  --tasks                Task states only.
  -s, --sort             Task states only; sort by cycle time instead of name.
```

B.2.12 edit

```
Usage: cylc [prep] edit [OPTIONS] REG

Edit suite definitions without having to move to their directory
locations, and with optional reversible inlining of include-files. Note
that Jinja2 suites can only be edited in raw form but the processed
version can be viewed with 'cylc [prep] view -p'.

1/cylc [prep] edit REG
Change to the suite definition directory and edit the suite.rc file.

2/ cylc [prep] edit -i,--inline REG
Edit the suite with include-files inlined between special markers. The
original suite.rc file is temporarily replaced so that the inlined
version is "live" during editing (i.e. you can run suites during
editing and cylc will pick up changes to the suite definition). The
inlined file is then split into its constituent include-files
again when you exit the editor. Include-files can be nested or
multiply-included; in the latter case only the first inclusion is
inlined (this prevents conflicting changes made to the same file).

3/ cylc [prep] edit --cleanup REG
Remove backup files left by previous INLINED edit sessions.

INLINED EDITING SAFETY: The suite.rc file and its include-files are
automatically backed up prior to an inlined editing session. If the
editor dies mid-session just invoke 'cylc edit -i' again to recover from
the last saved inlined file. On exiting the editor, if any of the
original include-files are found to have changed due to external
intervention during editing you will be warned and the affected files
will be written to new backups instead of overwriting the originals.
Finally, the inlined suite.rc file is also backed up on exiting
the editor, to allow recovery in case of accidental corruption of the
include-file boundary markers in the inlined file.

The edit process is spawned in the foreground as follows:
  $(G)EDITOR suite.rc
$GEDITOR or $EDITOR, and $TMDPIR, must be in your environment.

Examples:
   export EDITOR=vim
   export GEDITOR='gvim -f'      # -f: do not detach from parent shell!!
   export EDITOR='xterm -e vim'  # for gcylc, if gvim is not available
```



```
   export GEDITOR=emacs
   export EDITOR='emacs -nw'
You can set both $GEDITOR and $EDITOR to a GUI editor if you like, but
$GEDITOR at least *must* be a GUI editor, or an in-terminal invocation
of a non-GUI editor, if you want to spawn editing sessions via gcylc.

Arguments:
   REG                  Suite name

Options:
  -h, --help       show this help message and exit
  --owner=USER     User account name (defaults to $USER).
  --host=HOST      Host name (defaults to localhost).
  -v, --verbose    Verbose output mode.
  --debug          Turn on exception tracebacks.
  --db=DB          Suite database: 'u:USERNAME' for another user's default
                   database, or PATH to an explicit location. Defaults to
                   $HOME/.cylc/DB.
  -o, --override   Override cylc version compatibilty checking.
  -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.2.13 email-suite

```
USAGE: cylc [hook] email-suite EVENT SUITE MESSAGE

This is a simple suite event hook script that sends an email.
The command line arguments are supplied automatically by cylc.

For example, to get an email alert when a suite shuts down:

# SUITE.RC
[cylc]
   [[environment]]
      MAIL_ADDRESS = foo@bar.baz.waz
   [[event hooks]]
      events = shutdown
      script = cylc email-suite

See the Suite.rc Reference (Cylc User Guide) for more information
on suite and task event hooks and event handler scripts.
```

B.2.14 email-task

```
USAGE: cylc [hook] email-task EVENT SUITE TASKID MESSAGE

This is a simple task event hook handler script that sends an email.
The command line arguments are supplied automatically by cylc.

For example, to get an email alert whenever any task fails:

# SUITE.RC
[cylc]
   [[environment]]
      MAIL_ADDRESS = foo@bar.baz.waz
[runtime]
   [[root]]
      [[[event hooks]]]
         events = failed
         script = cylc email-task

See the Suite.rc Reference (Cylc User Guide) for more information
on suite and task event hooks and event handler scripts.
```

B.2.15 failed


```
Usage: cylc [task] failed [OPTIONS] [REASON]

This command is part of the cylc task messaging interface, used by
running tasks to communicate progress to their parent suite.

The failed command reports failure of task execution (and releases the
task lock to the lockserver if necessary). It is automatically called in
case of an error trapped by the task job script, but it can also be
called explicitly for self-detected failures if necessary.

Suite and task identity are determined from the task execution
environment supplied by the suite (or by the single task 'submit'
command, in which case case the message is just printed to stdout).

See also:
    cylc [task] message
    cylc [task] started
    cylc [task] succeeded

Arguments:
    REASON        - message explaining why the task failed.

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

B.2.16 gcontrol

```
Usage: cylc gcontrol [OPTIONS] REG
gcontrol [OPTIONS] REG

The cylc Suite Control GUI - can also be launched from the main GUI
(cylc gui / gcylc)

NOTE: daemonize important suites with the POSIX nohup command:
    $ nohup gcontrol [OPTIONS] REG &

Arguments:
   REG                  Suite name

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -w VIEWS, --views=VIEWS
                        Initial view panes for the suite control GUI. choose
                        one or two, comma separated, from 'dot', 'text', and
                        'graph'; the default is 'dot,text'
```

B.2.17 get-config

```
Usage: cylc [info] get-config REG [ITEMS...]

Print configuration items parsed from a suite definition (including any
defaults not explicitly set in the suite.rc file). This enables scripts
to control suites or process their output without assuming directory
locations and so on.
```



```
Config items containing spaces must be quoted. If the requested config
item is not a single value the corresponding internal data structure (a
nested Python dict) will be printed verbatim.

#_____________________________________________________________________EXAMPLE
# excerpt from a suite definition registered as foo.bar.baz:
title = local area implementation of modelX
[cylc]
    [[lockserver]]
        enable = True
[runtime]
    [[modelX]]
        command scripting = run-model.sh
        [[[environment]]]
            OUTPUT_DIR=/oper/live/modelX/output
#_____________________________________________________________________

$ cylc get-config foo.bar.baz title
$ local area implementation of modelX

$ cylc get-config foo.bar.baz cylc lockserver enable
$ True

$ cylc get-config foo.bar.baz runtime modelX environment OUTPUT_DIR
$ /oper/live/modelX/output

Arguments:
   REG                  Suite name
   [ITEMS...]                  suite.rc section and item hierarchy

Options:
  -h, --help         show this help message and exit
  --owner=USER       User account name (defaults to $USER).
  --host=HOST        Host name (defaults to localhost).
  -v, --verbose      Verbose output mode.
  --debug            Turn on exception tracebacks.
  --db=DB            Suite database: 'u:USERNAME' for another user's default
                     database, or PATH to an explicit location. Defaults to
                     $HOME/.cylc/DB.
  -o, --override     Override cylc version compatibilty checking.
  -d, --directories  Print all configured suite directory paths.
  -m, --mark-output  Prefix output lines with '!cylc!' to aid in automatic
                     parsing (output can be contaminated by stdout from login
                     scripts, for example, for remote invocation).
```

B.2.18 get-directory

```
Usage: cylc [db] get-directory REG

Retrieve and print the directory location of suite REG.
Tip: here's how to move to a suite definition directory:
  $ cd $(cylc get-dir REG).

Arguments:
   REG                  Suite name

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
```

B.2.19 graph


```
Usage: 1/ cylc [prep] graph [OPTIONS] REG [START [STOP]]
     Plot the suite.rc dependency graph for REG.
       2/ cylc [prep] graph [OPTIONS] -f,--file FILE
     Plot the specified dot-language graph file.

Plot cylc dependency graphs in a pannable, zoomable viewer.

The viewer updates automatically when the suite.rc file is saved during
editing. By default the full cold start graph is plotted; you can omit
cold start tasks with the '-w,--warmstart' option.  Specify the optional
initial and final cycle time arguments to override the suite.rc defaults.
If you just override the intitial cycle, only that cycle will be plotted.

GRAPH VIEWER CONTROLS:
    * Left-click to center the graph on a node.
    * Left-drag to pan the view.
    * Zoom buttons, mouse-wheel, or ctrl-left-drag to zoom in and out.
    * Shift-left-drag to zoom in on a box.
    * Also: "Best Fit" and "Normal Size".
  Family (namespace) grouping controls:
    Toolbar:
    * "group" - group all families up to root.
    * "ungroup" - recursively ungroup all families.
    Right-click menu:
    * "group" - close this node's parent family.
    * "ungroup" - open this family node.
    * "recursive ungroup" - ungroup all families below this node.

Arguments:
   REG                  Suite name
   [START]                  Initial cycle time to plot (default=2999010100)
   [STOP]                 Final cycle time to plot (default=2999010123)

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  -w, --warmstart       Plot the mid-stream warm start (raw start) dependency
                        graph (the default is cold start).
  -n, --namespaces      Plot the suite namespace inheritance hierarchy (task
                        run time properties).
  -f FILE, --file=FILE  View a specific dot-language graphfile.
  --output=FILE         Write out an image file, format determined by file
                        extension. The file will be rewritten if the suite
                        definition is changed while the viewer is running.
                        Available formats depend on your graphviz build and
                        may include png, jpg, gif, svg, pdf, ps, etc.
```

B.2.20 gui

```
Usage: cylc gui|gcylc [OPTIONS]
gcylc [OPTIONS]

This is the cylc Graphical User Interface; it is functionally equivalent
to the Command Line Interface in most respects (see 'cylc help').
Right-click on suites or groups to access cylc functionality from
suite editing and graphing through to suite control and monitoring.

The '-t,--timeout' timeout option affects port scanning to detect
running suites.

NOTE: to daemonize suites launched from the GUI use POSIX nohup command:
    $ nohup gcylc [OPTIONS] &
```

```
Arguments:

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
```

B.2.21 hold

```
Usage: cylc [control] hold [OPTIONS] REG [TASK]

Holding a suite stops it from submitting tasks that are ready to run,
until it is released. Holding a waiting TASK in a suite prevents it
from running or spawning successors, until it is released.

See also 'cylc [control] release'.

Arguments:
   REG                  Suite name
   [TASK]                  Task to hold (NAME%CYCLE)

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.22 housekeeping

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

Parallel archiving and cleanup of files or directories with names
that contain a cycle time. Matched items are grouped into batches in
which members are processed in parallel, by spawned sub-processes.
Once all batch members have completed, the next batch is processed.

OPERATE ('delete', 'move', or 'copy') on items (files or directories)
matching a Python-style regular expression MATCH in directory SOURCE
whose names contain a cycle time (as YYYYMMDDHH, or YYYYMMDD and HH
separately) more than OFFSET (integer hours) earlier than a base cycle
time (which can be $CYLC_TASK_CYCLE_TIME if called by a cylc task, or
otherwise specified on the command line).

FILE is a housekeeping config file containing one or more of lines of:

   VARNAME=VALUE
   # comment
```



```
   SOURCE    MATCH    OPERATION   OFFSET   [TARGET]

(example: $CYLC_DIR/conf/housekeeping.eg)

MATCH must be a Python-style regular expression (NOT A SHELL GLOB
EXPRESSION!) to match the names of items to be operated on AND to
extract the cycle time from the names via one or two parenthesized
sub-expressions - '(\d{10})' for YYYYMMDDHH, '(\d{8})' and '(\d{2})'
for YYYYMMDD and HH in either order. Partial matching can be used
(partial: 'foo-(\d{10})'; full: '^foo-(\d{10})$'). Any additional
parenthesized sub-expressions, e.g. for either-or matching, MUST
be of the (?:...) type to avoid creating a new match group.

SOURCE and TARGET must be on the local filesystem and may contain
environment varables such as $HOME or ${FOO} (e.g. as defined in the
suite.rc file for suite housekeeping tasks). Variables defined in
the housekeeping file itself can also be used, as above.

TARGET may contain the strings YYYYMMDDHH, YYYY, MM, DD, HH; these
will be replaced with the extracted cycle time for each matched item,
e.g. $ARCHIVE/oper/YYYYMM/DD.

If TARGET is specified for the 'delete' operation, matched items in
SOURCE will not be deleted unless an identical item is found in
TARGET. This can be used to check that important files have been
successfully archived before deleting the originals.

The 'move' and 'copy' operations are aborted if the TARGET/item already
exists, but a warning is emitted if the source and target items are not
identical.

To implement a simple ROLLING ARCHIVE of cycle-time labelled files or
directories: just use 'delete' with OFFSET set to the archive length.

SAFE ARCHIVING: The 'move' operation is safe - it uses Python's
shutils.move() which renames files on the local disk partition and
otherwise copies before deleting the original. But for extra safety
consider two-step archiving and cleanup:
1/ copy files to archive, then
2/ delete the originals only if identicals are found in the archive.

Options:
  -h, --help            show this help message and exit
  --cycletime=YYYYMMDDHH
                        Cycle time, defaults to $CYLC_TASK_CYCLE_TIME
  --mode=MODE           Octal umask for creating new destination directories.
                        E.g. 0775 for drwxrwxr-x
  -o LIST, --only=LIST  Only action config file lines matching any member of a
                        comma-separated list of regular expressions.
  -e LIST, --except=LIST
                        Only action config file lines NOT matching any member
                        of a comma-separated list of regular expressions.
  -v, --verbose         print the result of every action
  -d, --debug           print item matching output.
  -c, --cheapdiff       Assume source and target identical if the same size
  -b INT, --batchsize=INT
                        Batch size for parallel processing of matched files.
                        Members of each batch (matched items) are processed in
                        parallel; when a batch completes, the next batch
                        starts. Defaults to a batch size of 1, i.e. sequential
                        processing.
```

B.2.23 import-examples

```
USAGE: cylc [admin] import-examples TOPDIR

Copy the cylc example suites to TOPDIR and register them for use.

Arguments:
```

```
   TOPDIR     Example suite destination directory: TOPDIR/examples/.
```

B.2.24 insert

```
Usage: cylc [control] insert [OPTIONS] REG TASK[%STOP]

Insert a task into a running suite. Inserted tasks will spawn successors
as normal unless they are 'one-off' tasks.
See also 'cylc [task] submit', for running single tasks without the scheduler.

Arguments:
   REG                  Suite name
   TASK[%STOP]                  Task to insert (NAME%TAG)[%STOPTAG]

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.25 jobscript

```
USAGE: cylc [prep] jobscript [OPTIONS] REG TASK

Generate a task job script and print it to stdout.

Here's how to capture the script in the vim editor:
  % cylc jobscript REG TASK | vim -
Emacs unfortunately cannot read from stdin:
  % cylc jobscript REG TASK > tmp.sh; emacs tmp.sh

This command wraps 'cylc [control] submit --dry-run'.
Other options (e.g. for suite host and owner) are passed
through to the submit command.

Options:
  -h,--help   - print this usage message.
 (see also 'cylc submit --help')

Arguments:
  REG         - Registered suite name.
  TASK        - Task ID (NAME%TAG)
```

B.2.26 list

```
Usage: cylc [info|prep] list|ls [OPTIONS] REG

Print a suite's task list or runtime namespace inheritance hierarchy.
To graph the namespace hierarchy, see 'cylc graph'.

Arguments:
   REG                  Suite name

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
```



```
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  -t, --tree      Print the full runtime inheritance hierarchy.
  -b, --box       (with -t,--tree) Use unicode box characters.
```

B.2.27 lockclient

```
Usage: cylc [util] lockclient|lc [OPTIONS]

This is the command line client interface to the cylc lockserver daemon,
for server interrogation and manual lock management.

Use of the lockserver is optional (see suite.rc documentation)

Manual lock acquisition is mainly for testing purposes, but manual
release may be required to remove stale locks if a suite or task dies
without cleaning up after itself.

See also:
    cylc lockserver

Options:
  -h, --help            show this help message and exit
  --acquire-task=SUITE:TASK%CYCLE
                        Acquire a task lock.
  --release-task=SUITE:TASK%CYCLE
                        Release a task lock.
  --acquire-suite=SUITE
                        Acquire an exclusive suite lock.
  --acquire-suite-nonex=SUITE
                        Acquire a non-exclusive suite lock.
  --release-suite=SUITE
                        Release a suite and associated task locks
  -p, --print           Print all locks.
  -l, --list            List all locks (same as -p).
  -c, --clear           Release all locks.
  -f, --filenames       Print lockserver PID, log, and state filenames.
  -t SECONDS, --timeout=SECONDS
                        Network connection timeout (default 1 second).
```

B.2.28 lockserver

```
Usage: cylc [util] lockserver [-f CONFIG] ACTION

The cylc lockserver daemon brokers suite and task locks for a single
user. These locks are analogous to traditional lock files, but they work
even for tasks that start and finish executing on different hosts. Suite
locks prevent multiple instances of the same suite from running at the
same time (even if registered under different names) unless the suite
allows that. Task locks do the same for individual tasks (even if
submitted outside of their suite using 'cylc submit').

The command line user interface for interrogating the daemon, and
for manual lock management, is 'cylc lockclient'.

Use of the lockserver is optional (see suite.rc documentation).

The lockserver reads a config file that specifies the location of the
daemon's process ID, state, and log files. The default config file
is '$CYLC_DIR/conf/lockserver.conf'. You can specify an alternative
config file on the command line, but then all subsequent interaction
with the daemon via the lockclient command must also specify the same
file (this is really only for testing purposes). The default process ID,
state, and log files paths are relative to $HOME so this should be
```

```
sufficient for all users.

The state file records currently held locks and, if it exists at
startup, is used to initialize the lockserver (i.e. suite and task locks
are not lost if the lockserver is killed and restarted). All locking
activitiy is recorded in the log file.

Arguments:
  ACTION   -  'start', 'stop', 'status', 'restart', or 'debug'
              In debug mode the server does not daemonize so its
              the stdout and stderr streams are not lost.

Options:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --config-file=CONFIGFILE
                        Config file (default $CYLC_DIR/lockserver.conf
```

B.2.29 log

```
Usage: cylc [info] log [OPTIONS] REG
Print and filter cylc suite (not task) log files.

Arguments:
   REG                  Suite name

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  -l, --location        Print the suite log file location and exit. This is
                        equivalent to 'cylc get-config SUITE cylc logging
                        directory'.
  -t TASK, --task=TASK  Filter the log for messages from a specific task
  -f RE, --filter=RE    Filter the log with a Python-style regular expression
                        e.g. '\[(foo|bar).*(started|succeeded)'
  -r INT, --rotation=INT
                        Rotation number (to view older, rotated logs)
```

B.2.30 message

```
Usage: cylc [task] message [OPTIONS] MESSAGE

This command is part of the cylc task messaging interface, used by
running tasks to communicate progress to their parent suite.

Suite and task identity are determined from the task execution
environment supplied by the suite (or by the single task 'submit'
command, in which case case the message is just printed to stdout).

See also:
    cylc [task] started
    cylc [task] succeeded
    cylc [task] failed

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.
  -v, --verbose         Verbose output mode.
```

B.2.31 monitor

```
Usage: cylc [info] monitor [OPTIONS] REG

A terminal-based suite monitor that updates the current state of all
tasks in real time. It is effective even for quite large suites if
'--align' is not used. Being a passive monitor that cannot intervene in
a suite's operation, it is allowed to monitor suites owned by others
and/or running on remote hosts.

Arguments:
   REG                  Suite name

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -a, --align           Align columns by task name. This option is only useful
                        for small suites.
```

B.2.32 nudge

```
Usage: cylc [control] nudge [OPTIONS] REG

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.

Arguments:
   REG                  Suite name

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```


B.2.33 ping

```
Usage: cylc [discover] ping [OPTIONS] REG [TASK]

If suite REG (or task TASK in it) is running, exit (silently, unless
-v,--verbose is specified); else print an error message and exit with
error status. For tasks, success means the task proxy is currently in
the 'running' state.

Arguments:
   REG                  Suite name
   [TASK]                  Task NAME%TAG (TAG is cycle time or "a:INT")

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
  --print-ports         Print cylc's configured port range.
```

B.2.34 print

```
Usage: cylc [db] print [OPTIONS] [REGEX]

Print suite database registrations.

Note on result 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 printing purposes there is an implicit wildcard at the end of
each pattern ('foo' is the same as 'foo.*'); use the string end marker
to prevent this ('foo$' matches only literal 'foo').

Arguments:
   [REGEX]                 Suite name regular expression pattern

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  -t, --tree      Print registrations in nested tree form.
  -b, --box       Use unicode box drawing characters in tree views.
  -a, --align     Align columns.
  -x              don't print suite definition directory paths.
  -y              Don't print suite titles.
  --fail          Fail (exit 1) if no matching suites are found.
```

B.2.35 purge

```
Usage: cylc [control] purge [OPTIONS] REG TASK STOP
```

```
Remove an entire tree of dependent tasks, across multiple cycles, from a
running suite. The root task will be forced to spawn and will then be
removed, then so will every task that depends on it, and every task that
depends on those, and so on until the given stop cycle time.

WARNING: THIS COMMAND IS DANGEROUS but in case of disaster you can
restart the suite from the automatic pre-purge state dump (the filename
will be logged by cylc before the purge is actioned.)

UNDERSTANDING HOW PURGE WORKS: cylc identifies tasks that depend on
the root task, and then on its downstream dependents, and then on
theirs, etc., by simulating what would happen if the root task were to
trigger: it artificially sets the root task to the "succeeded" state
then negotatiates dependencies and artificially sets any tasks whose
prerequisites get satisfied to "succeeded"; then it negotiates
dependencies again, and so on until the stop cycle is reached or nothing
new triggers. Finally it marks "virtually triggered" tasks for removal.
Consequently:
 * Dependent tasks will only be identified as such if they have already
   spawned into the root cycle, otherwise they will be missed by the
   purge. To avoid this, wait until all tasks that depend on the root
   have caught up to it before purging.
 * If you purge a task that has already finished, only it and its own
   successors will be purged (other downstream tasks will already have
   triggered if they were able to).
[development note: post cylc-3.0 we could potentially use the suite
graph to determine downstream tasks to remove, without doing this
internal triggering simulation.]

Arguments:
   REG                  Suite name
   TASK                  Task (NAME%CYCLE) to start purge
   STOP                  Cycle (inclusive!) to stop purge

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.36 refresh

```
Usage: cylc [db] refresh [OPTIONS] [REGEX]

Check a suite database for invalid registrations (no suite definition
directory or suite.rc file) and refresh suite titles in case they have
changed since the suite was registered. Explicit wildcards must be
used in the match pattern (e.g. 'f' will not match 'foo.bar' unless
you use 'f.*').

Arguments:
   [REGEX]                  Suite name match pattern

Options:
  -h, --help        show this help message and exit
  --owner=USER      User account name (defaults to $USER).
  --host=HOST       Host name (defaults to localhost).
  -v, --verbose     Verbose output mode.
```

```
  --debug           Turn on exception tracebacks.
  --db=DB           Suite database: 'u:USERNAME' for another user's default
                    database, or PATH to an explicit location. Defaults to
                    $HOME/.cylc/DB.
  -o, --override    Override cylc version compatibilty checking.
  -u, --unregister  Automatically unregister invalid registrations.
```

B.2.37 register

```
Usage: cylc [db] register [OPTIONS] REG PATH

Register the suite definition located in PATH as REG.

Suite names are hierarchical, delimited by '.' (foo.bar.baz); they
may contain letters, digits, underscore, and hyphens. Colons are not
allowed because directory paths incorporating the suite name are
sometimes needed in PATH variables.

EXAMPLES:

For suite definition directories /home/bob/(one,two,three,four):

% cylc db reg bob         /home/bob/one
% cylc db reg foo.bag     /home/bob/two
% cylc db reg foo.bar.baz /home/bob/three
% cylc db reg foo.bar.waz /home/bob/four

% cylc db pr '^foo'             # print in flat form
  bob         | "Test Suite One"   | /home/bob/one
  foo.bag     | "Test Suite Two"   | /home/bob/two
  foo.bar.baz | "Test Suite Four"  | /home/bob/three
  foo.bar.waz | "Test Suite Three" | /home/bob/four

% cylc db pr -t '^foo'          # print in tree form
  bob        "Test Suite One"   | /home/bob/one
  foo
   |-bag     "Test Suite Two"   | /home/bob/two
   `-bar
     |-baz   "Test Suite Three" | /home/bob/three
     `-waz   "Test Suite Four"  | /home/bob/four

Arguments:
   REG                  Suite name
   PATH                  Suite definition directory

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
```

B.2.38 release

```
Usage: cylc [control] release|unhold [OPTIONS] REG [TASK]

Release a suite or a single task from a hold, allowing it run as normal.

Holding a suite stops it from submitting tasks that are ready to run,
until it is released. Holding a waiting TASK in a suite prevents it
from running or spawning successors, until it is released.

See also 'cylc [control] hold'.

Arguments:
```

```
   REG                  Suite name
   [TASK]                  Task to release (NAME%CYCLE)

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.39 remove

```
Usage: cylc [control] remove|kill [OPTIONS] REG TARGET

Remove a single task, or all tasks with a common TAG (cycle time or
asynchronous 'a:INT'), from a running suite.

Target tasks will be forced to spawn successors before being removed if
they have not done so already, unless you use '--no-spawn'.

Arguments:
   REG                  Suite name
   TARGET                  NAME%CYCLE or NAME%a:TAG to remove a single task;
               CYCLE or a:TAG to remove all tasks with the same tag.

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
  --no-spawn            Do not spawn successors before removal.
```

B.2.40 reregister

```
Usage: cylc [db] reregister|rename [OPTIONS] REG1 REG2

Change the name of a suite (or group of suites) from REG1 to REG2.
Example:
  cylc db rereg foo.bar.baz test.baz

Arguments:
   REG1                 original name
   REG2                 new name

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



```
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
```

B.2.41 reset

```
Usage: cylc [control] reset [OPTIONS] REG 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)
 Or:
 5/ force it to spawn if it hasn't done so already (--spawn)

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.

Forcing a task to spawn a successor may be necessary in the case of a
failed "sequential task" that cannot be re-run successfully after fixing
the problem, because sequential tasks do not spawn until they succeed.
Alternatively, you could force the failed task to the succeeded state,
or insert a new instance into the suite at the next cycle time.

Arguments:
   REG                  Suite name
   TASK                  Target task ID

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -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.
  --spawn               Force a task to spawn its successor if it hasn't
                        already.
```

B.2.42 restart

```
Usage: cylc [control] restart [OPTIONS] REG [FILE]

Restart a cylc suite from a previous state.
(To start a suite from a specific cycle time, see 'cylc run REG').

The most recent previous state is used by default but other state dumps
can be specified on the command line (e.g. cylc writes a special state
dump and logs its filename before actioning any intervention command).
```

```
By default:
 1/ 'submitted', 'running', or 'failed' states will be resubmitted.
     Use '--no-reset' to prevent this.
 2/ 'held' tasks will be released.
    Use '--no-release' to prevent this.
 3/ A final cycle time set prior to shutdown will be ignored on restart.
    Use '--keep-stopcycle' to prevent this.

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

NOTE: suites can be (re)started on remote hosts or other user accounts
if passwordless ssh is set up. The ssh tunnel will remain open to
receive the suite stdout and stderr streams. To control the running
suite from the local host requires the suite passphrase to be installed.
If they exist /etc/profile and $HOME/.profile will be sourced on the
remote host prior to attempting to run the suite.

Arguments:
   REG                  Suite name
   [FILE]                  Optional state dump file (in the suite state
                        dump directory, or give full path).

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --no-reset            Do not reset failed tasks to ready on restarting
  --no-release          Do not release held tasks on restarting.
  --keep-stopcycle      Do not ignore a previously set final cycle time on
                        restarting
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second). For an initial check that the
                        suite isn't already running.
  --until=CYCLE         Shut down after all tasks have PASSED this cycle time.
  --hold                Hold (don't run tasks) immediately on starting.
  --hold-after=CYCLE    Hold (don't run tasks) AFTER this cycle time.
  -s, --simulation-mode
                        Use dummy tasks that masquerade as the real thing, and
                        accelerate the wall clock: get the scheduling right
                        without having to run the real suite tasks.
  --fail=NAME%TAG       (SIMULATION MODE) get the specified task to report
                        failure and then abort.
  --timing              Turn on main task processing loop timing, which may be
                        useful for testing very large suites of 1000+ tasks.
  --gcylc               (for use by gcylc only).
```

B.2.43 run

```
Usage: cylc [control] run|start [OPTIONS] REG [START]

Start a suite running at a specified initial cycle time.
(To restart a suite from a previous state, see 'cylc restart REG').

The following are all equivalent if no intercycle dependence exists:
  1/ Cold start (default)     : use special cold-start tasks
  2/ Warm start (-w,--warm)   : assume a previous cycle
  3/ Raw  start (-r,--raw)    : assume nothing

1/ COLD START -- at start up, insert designated cold-start tasks in the
waiting state, to satisfy any initial dependence on a previous cycle.
```

```
2/ WARM START -- at start up, insert designated cold-start tasks in the
succeeded state, to stand in for a previous cycle (from a previous run).

3/ RAW START -- do not insert cold-start tasks at all.

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

NOTE: suites can be (re)started on remote hosts or other user accounts
if passwordless ssh is set up. The ssh tunnel will remain open to
receive the suite stdout and stderr streams. To control the running
suite from the local host requires the suite passphrase to be installed.
If they exist /etc/profile and $HOME/.profile will be sourced on the
remote host prior to attempting to run the suite.

Arguments:
   REG                  Suite name
   [START]                  Initial cycle time, optional if defined in the
                        suite.rc file (in which case the command line
                        takes priority and a suite.rc final cycle time
                        will be ignored); not required if the
                        suite contains no cycling tasks.

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  -w, --warm            Warm start the suite
  -r, --raw             Raw start the suite
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second). For an initial check that the
                        suite isn't already running.
  --until=CYCLE         Shut down after all tasks have PASSED this cycle time.
  --hold                Hold (don't run tasks) immediately on starting.
  --hold-after=CYCLE    Hold (don't run tasks) AFTER this cycle time.
  -s, --simulation-mode
                        Use dummy tasks that masquerade as the real thing, and
                        accelerate the wall clock: get the scheduling right
                        without having to run the real suite tasks.
  --fail=NAME%TAG       (SIMULATION MODE) get the specified task to report
                        failure and then abort.
  --timing              Turn on main task processing loop timing, which may be
                        useful for testing very large suites of 1000+ tasks.
  --gcylc               (for use by gcylc only).
```

B.2.44 scan

```
Usage: cylc [discover] scan [OPTIONS]

Detect (by port scanning) running cylc suites and lockservers, and
print the results. The simple space-delimited output format is designed
for easy parsing: "SUITE OWNER HOST PORT".

Here's one way to parse 'cylc scan' output by shell script:
_____________________________________________________________
#!/bin/bash
# parse suite, owner, host, port from 'cylc scan' output lines
OFIS=$IFS
IFS=$'
'; for LINE in $( cylc scan ); do
    # split on space and assign tokens to positional parameters:
    IFS=$' '; set $LINE
    echo "$1 - $2 - $3 - $4"
```

```
done
IFS=$OFIS
---------------------------------------------------------------
"Connection Denied" indicates another user's suite.

Arguments:

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  --print-ports         Print cylc's configured port range.
```

B.2.45 scp-transfer

```
Usage: cylc [util] scp-transfer [OPTIONS]

An scp wrapper for transferring a list of files and/or directories
at once. The source and target scp URLs can be local or remote (scp
can transfer files between two remote hosts). Passwordless ssh must
be configured appropriately.

ENVIRONMENT VARIABLE INPUTS:
$SRCE  - list of sources (files or directories) as scp URLs.
$DEST  - parallel list of targets as scp URLs.
The source and destination lists should be space-separated.

We let scp determine the validity of source and target URLs.
Target directories are created pre-copy if they don't exist.

Options:
 -v     - verbose: print scp stdout.
 --help - print this usage message.
```

B.2.46 search

```
Usage: cylc [prep] search|grep [OPTIONS] REG PATTERN [PATTERN2...]

Search for pattern matches in suite definitions and any files in the
suite bin directory. Matches are reported by line number and suite
section. An unquoted list of PATTERNs will be converted to an OR'd
pattern. Note that the order of command line arguments conforms to
normal cylc command usage ('command SUITE REG PATTERN [PATTERN2...] ') not shell grep usage.

Note that this command performs a text search on the suite definition,
it does not search the data structure that results from parsing the
suite definition - so it will not report implicit default settings.

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

Arguments:
   REG                  Suite name
   PATTERN                  Python-style regular expression
   [PATTERN2...]                  Addtional search patterns

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
```

```
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  -x              Do not search in the suite bin directory
```

B.2.47 set-runahead

```
Usage: cylc [control] set-runahead [OPTIONS] REG [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 runahead
limit will be set - DO NOT DO THIS for for any cycling suite that has
no near stop cycle set and is not constrained by clock-triggered
tasks.

Arguments:
   REG                  Suite name
   [HOURS]                  Runahead limit (default: no limit)

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.48 set-verbosity

```
Usage: cylc [control] set-verbosity [OPTIONS] REG 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.

Arguments:
   REG                  Suite name
   LEVEL                  debug, info, warning, error, or critical

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
```

```
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.49 show

```
Usage: cylc [info] show [OPTIONS] REG [NAME[%TAG]]

Interrogate a running suite for its title and task list, task
descriptions, current state of task prerequisites and outputs and, for
clock-triggered tasks, whether or not the trigger time is up yet.

Arguments:
   REG                  Suite name
   [NAME[%TAG]]                 Task name or ID

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
```

B.2.50 started

```
Usage: cylc [task] started [OPTIONS]

This command is part of the cylc task messaging interface, used by
running tasks to communicate progress to their parent suite.

The started command reports commencement of task execution (and it
acquires a task lock from the lockserver if necessary). It is
automatically written to the top of task job scripts by cylc and
therefore does not need to be called explicitly by task scripting.

Suite and task identity are determined from the task execution
environment supplied by the suite (or by the single task 'submit'
command, in which case case the message is just printed to stdout).

See also:
    cylc [task] message
    cylc [task] succeeded
    cylc [task] failed

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

B.2.51 stop

```
Usage: cylc [control] stop|shutdown [OPTIONS] REG [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.

Arguments:
   REG                  Suite name
   [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
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
  --now                 Shut down immediately; see WARNING above.
```

B.2.52 submit

```
Usage: cylc [task] submit|single [OPTIONS] REG 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 (i.e. only if
the same task, NAME%TAG, is not currently running in the suite).

Arguments:
   REG                  Suite name
   TASK                  Target task (NAME%TAG)

Options:
  -h, --help        show this help message and exit
  --owner=USER      User account name (defaults to $USER).
  --host=HOST       Host name (defaults to localhost).
  -v, --verbose     Verbose output mode.
  --debug           Turn on exception tracebacks.
  --db=DB           Suite database: 'u:USERNAME' for another user's default
                    database, or PATH to an explicit location. Defaults to
                    $HOME/.cylc/DB.
  -o, --override    Override cylc version compatibilty checking.
  -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.2.53 succeeded

```
Usage: cylc [task] succeeded [OPTIONS]

This command is part of the cylc task messaging interface, used by
running tasks to communicate progress to their parent suite.

The succeeded command reports successful completion of task execution
(and releases the task lock to the lockserver if necessary). It is
automatically written to the end of task jobs scripts by cylc, except in
the case of detaching tasks (suite.rc: 'manual completion = True'), in
which case it must be called explicitly by final task scripting.

Suite and task identity are determined from the task execution
environment supplied by the suite (or by the single task 'submit'
command, in which case case the message is just printed to stdout).

See also:
    cylc [task] message
    cylc [task] started
    cylc [task] failed

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

B.2.54 test-db

```
USAGE: cylc [admin] test-db [--help]
A thorough test of suite registration database functionality.
Options:
  --help   Print this usage message.
```

B.2.55 test-suite

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

Run an automated test of core cylc functionality using a new copy of
$CYLC_DIR/examples/admin. This should be used to check that new
developments in the cylc codebase have not introduced serious bugs.

Currently the test does the following:
  - Copies the 'intro' example suite definition directory;
  - Registers the new suite;
  - Starts the suite at T=06Z, with task X set to fail at 12Z;
  - Unlocks the running suite;
  - Sets a stop time at 12Z (i.e. T+30 hours);
  - Waits for the suite to stall as result of X failing;
  - Inserts a new coldstart task at 06Z (T+24 hours);
  - Purges the failed X and dependants through to 00Z (T+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.
  - Runs a single task 'prep' outside of the suite.

Options:
  -h, --help   Print this help message and exit.
```

B.2.56 trigger

```
Usage: cylc [control] trigger [OPTIONS] REG TASK

Get a task to trigger immediately (unless the suite is paused,
in which case it will trigger when normal operation is resumed).
This is effected by setting the task to the 'ready' state (all
prerequisites satisfied) and, for clock-triggered tasks, ignoring
the designated trigger time.

Arguments:
   REG                  Suite name
   TASK                  Target task
```



```
Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.57 unblock

```
Usage: cylc [control] unblock [OPTIONS] REG

Unblock a blocked suite, which will refuse to comply with intervention
commands until deliberately unblocked. This is a crude safety device to guard
against accidental intervention in your own suites (if you are running multiple
suites at once a simple typo on the command line could target the wrong suite).

Arguments:
   REG                  Suite name

Options:
  -h, --help            show this help message and exit
  --owner=USER          User account name (defaults to $USER).
  --host=HOST           Host name (defaults to localhost).
  -v, --verbose         Verbose output mode.
  --debug               Turn on exception tracebacks.
  --db=DB               Suite database: 'u:USERNAME' for another user's
                        default database, or PATH to an explicit location.
                        Defaults to $HOME/.cylc/DB.
  -o, --override        Override cylc version compatibilty checking.
  --use-ssh             Use ssh to re-invoke the command on the suite host.
  -t SEC, --timeout=SEC
                        Network connection timeout in case of hung ports
                        (default 1 second).
  -p FILE, --passphrase=FILE
                        Suite passphrase file (if not in a default location)
  -f, --force           Do not ask for confirmation before acting.
```

B.2.58 unregister

```
Usage: cylc [db] unregister [OPTIONS] REGEX

Remove one or more suites from your suite database. The REGEX pattern
must match whole suite names to avoid accidental de-registration of
partial matches (e.g. 'bar.baz' will not match 'foo.bar.baz').

Associated suite definition directories will not be deleted unless the
'-d,--delete' option is used.

Arguments:
   REGEX                  Regular expression to match suite names.

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
```



```
                database, or PATH to an explicit location. Defaults to
                $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  -d, --delete    Delete the suite definition directory too (!DANGEROUS!).
  -f, --force     Don't ask for confirmation before deleting suite
                  definitions.
  --dry-run       Just show what I would do.
```

B.2.59 validate

```
Usage: cylc [prep] validate [OPTIONS] REG

Validate a suite definition against the official specification
held in $CYLC_DIR/conf/suiterc.spec.

If the suite definition uses include-files reported line numbers
will be wrong because the parser sees an inlined version of the suite.
Use 'cylc view -i SUITE' to see the inlined version for comparison.

Arguments:
   REG                 Suite name

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
                  database, or PATH to an explicit location. Defaults to
                  $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
```

B.2.60 view

```
Usage: cylc [prep] view [OPTIONS] REG

View a read-only temporary copy of suite NAME's suite.rc file, in your
editor, after optional include-file inlining and Jinja2 preprocessing.

The edit process is spawned in the foreground as follows:
  $(G)EDITOR suite.rc
$GEDITOR or $EDITOR, and $TMDPIR, must be in your environment.

Examples:
   export EDITOR=vim
   export GEDITOR='gvim -f'      # -f: do not detach from parent shell!!
   export EDITOR='xterm -e vim'  # for gcylc, if gvim is not available
   export GEDITOR=emacs
   export EDITOR='emacs -nw'
You can set both $GEDITOR and $EDITOR to a GUI editor if you like, but
$GEDITOR at least *must* be a GUI editor, or an in-terminal invocation
of a non-GUI editor, if you want to spawn editing sessions via gcylc.

For remote host or owner, the suite will be printed to stdout unless
the '-g,--gui' flag is used to spawn a remote GUI edit session.

See also 'cylc [prep] edit'.

Arguments:
   REG                 Suite name

Options:
  -h, --help      show this help message and exit
  --owner=USER    User account name (defaults to $USER).
  --host=HOST     Host name (defaults to localhost).
  -v, --verbose   Verbose output mode.
  --debug         Turn on exception tracebacks.
  --db=DB         Suite database: 'u:USERNAME' for another user's default
```



```
                database, or PATH to an explicit location. Defaults to
                $HOME/.cylc/DB.
  -o, --override  Override cylc version compatibilty checking.
  -i, --inline    Inline any include-files.
  -j, --jinja2    View the suite after Jinja2 template processing. This
                necessarily implies '-i' as well.
  -m, --mark      (With '-i') Mark inclusions in the left margin.
  -l, --label     (With '-i') Label file inclusions with the file name. Line
                numbers will not correspond to those reported by the parser.
  -s, --single    (With '-i') Inline only the first instances of any multiply-
                included files. 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).
  -g, --gui       Use GUI editor $GEDITOR instead of $EDITOR. This option is
                automatically used when an editing session is spawned by
                gcylc.
  --stdout        Print the suite definition to stdout.
```

B.2.61 warranty

```
USAGE: cylc [license] warranty [--help]
   Cylc is released under the GNU General Public License v3.0
This command prints the GPL v3.0 disclaimer of warranty.
Options:
  --help   Print this usage message.
```

C The Cylc Lockserver

Each cylc user can optionally run his/her own lockserver to prevent accidental invocation of multiple instances of the same suite or task at the same time. The suite and task locks brokered by the lockserver are analogous to traditional lock files, but they work across a network, even for distributed suites containing tasks that start executing on one host and finish on another.

Accidental invocation of multiple instances of the same suite or task at the same time can have serious consequences, so use of the lockserver should be considered for important operational suites, but it may be considered an unnecessary complication for general less critical usage, so it is currently disabled by default.

To enable the lockserver:

```
# SUITE.RC
use lockserver = True
```

The suite will now abort at start-up if it cannot connect to the lockserver. To start your lockserver daemon,

```
% cylc lockserver start
```

To check that it is running,

```
% cylc lockserver status
```

For detailed usage information,

```
% cylc lockserver --help
```

There is a command line client interface,

```
% cylc lockclient --help
```

for interrogating the lockserver and managing locks manually (e.g. releasing locks if a suite was killed before it could clean up after itself).

To watch suite locks being acquired and released as a suite runs,

```
% watch cylc lockclient --print
```


D The Suite Control GUI Graph View

The graph view in the gcontrol 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 (up to several hundred tasks or more) *but* because the graphviz engine does a new global layout every time the graph changes the layout is often not very stable. This may not be a solvable problem even in principle as it seems likely that making continual incremental changes to an existing graph without redoing the global layout would inevitably result in a horrible mess.

The following features of the graph view, however, help mitigate the the jumping 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)
- In most suites, toggling the off-white "base graph" nodes (which do not represent current live task proxies and are just present to fill out the graph structure) will dramatically reduce the size of the graph (but it will split into several disconnected sub-trees).
- 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).

E Simulation Mode

If you run a suite in simulation mode its tasks will be replaced by dummy tasks that just print a message and then exit after a few seconds, and the suite will run quickly on an accelerated clock. As far as cylc is concerned this is essentially identical to real operation. Simulation mode can be used to get the scheduling of a complex suite right without having to run real tasks, and to test recovery strategies for simulated task failure scenarios.

By configuring how the accelerated clock is initialized you can watch suites catch up and transition from delayed to real time operation.

Remote task hosting and job submission configuration is ignored in simulation mode so you can simulate any suite, even outside of its normal operating environment.

A more realistic but less portable simulation mode, with full remote job submission etc., can be achieved by simply commenting out task command scripting so that it defaults to dummy command scripting, and running the suite in live mode (note however that this won't work for tasks with internal outputs unless you provide your own dummy command scripting that reports the internal outputs finished).

Simulation mode was, and remains, an important aid to cylc development because it allows testing of all aspects of scheduling without having to run real tasks in real time. Prior to cylc-3 it was also a critical aid to suite construction, quickly identifying any mismatch between the user-defined task prerequisites and outputs across a suite. Post cylc-3.0 this is less important because task prerequisites and outputs are now implicitly defined by the suite dependency graph and, short of a bug in cylc, the suite will run according to the graph.

E.1 Clock Rate and Offset

You can set the simulation mode 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 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 mode, or vice versa - any attempt to do the former would certainly be a mistake (because the simulation mode dummy tasks do not generate any of the real outputs depended on by downstream live tasks), and the latter, while feasible, would corrupt the live state dump and turn it over to simulation mode. The easiest way to test a live suite in simulation mode, if you don't want to obliterate the current state dump by doing a cold or warm start (as opposed to a restart from the previous state) is to take a quick copy of the suite and run the copy in simulation mode. However, if you really want to run a live suite forward in simulation mode without copying it, do this:

1. Backup the live state dump file.
2. Edit the mode line in the state dump file and restart in simulation mode.
3. Later, restart the live suite from the restored live state dump backup.

F Cylc Development History

F.1 Pre-3.0

Early versions of cylc were focused on developing and testing the new scheduling algorithm, and the suite design interface at the time was essentially the quickest route to that end. A suite was a collection of "task definition files" that encoded the prerequisites and outputs of each task in a direct reflection of cylc's internal task proxies. This way of defining suites exposed cylc's self-organising nature to the user, and it did have some nice properties. For instance a group of tasks could be transferred directly from one suite to another by simply copying the taskdef files over (and checking that prerequisite and output messages were consistent with the new suite). However, ensuring consistency of prerequisites and outputs across a large suite could be tedious; a few edge cases associated with suite start-up and forecast model restart dependencies were, arguably, difficult to understand; and the global structure of a suite was not readily apparent until run time (although to counter this cylc 2.x could generate run-time resolved dependency graphs very quickly in simulation mode).

F.2 Version 3.0

Version 3.0 implemented an entirely new suite design interface in which one defines the suite dependency graph, execution environment, and command scripting for each task, in a single structured, validated, configuration file - the suite.rc file. This *really* makes suite structure apparent at a glance, and task prerequisites and outputs (and some other important parameters besides) no longer need to be specified by the user because they are implied by the graph.

F.3 Version 4.0

Version 4.0 has the following major improvements over cylc-3.x, along with many refinements:

- Suite registration has been generalized from GROUP:NAME to a hierarchy of arbitrary depth, e.g foo.bar.baz, allowing suites to be organized in a tree-like structure.

- The suite.rc file has been extensively reorganized. In particular it now defines a *namespace hierarchy* of families and tasks to allow common runtime properties to be grouped naturally among related tasks.
- The Jinja2 template engine is now supported, adding general programmability to cylc suite definitions.

G Pyro

Pyro (Python Remote Objects) is a widely used open source objected oriented Remote Procedure Call technology developed by Irmen de Jong.

Earlier versions of cylc used the Pyro Nameserver to marshal communication between client programs (tasks, commands, viewers, etc.) and their target suites. This worked well, but in principle it provided a route for one suite or user on the subnet to bring down all running suites by killing the nameserver. Consequently cylc now uses Pyro simply as a lightweight object oriented wrapper for direct network socket communication between client programs and their target suites - all suites are thus entirely isolated from one another.

H Acknowledgements

Bernard Miville and Phil Andrews (NIWA) for discussion, bug finding, and many suggestions that have improved cylc's usability and functionality; David Matthews and Matthew Shin (Met Office UK) for the same, and much code development as well.

I GNU GENERAL PUBLIC LICENSE v3.0

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

   The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright.

Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

   (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

   (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

   (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

   (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

   A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

   A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For

a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been

terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a

patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special

requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.