Cylc Rose Suite Design Best Practice Guide

Version 1.0 - 23 March 2017

Last updated for: Cylc-7.2.0 and Rose-2017.02.0

Hilary Oliver, Dave Matthews, Andy Clark, and Contributors

Contents

1 Introduction

This document provides guidance on making complex Cylc + Rose workflows that are clear, maintainable, and portable. Note that best practice advice may evolve over time with the capabilities of Rose and Cylc.

Content is drawn from the Rose and Cylc user guides, earlier Met Office suite design and operational suite review documents, experience with real suites across the Unified Model Consortium, and discussion among members of the UM TISD (Technical Infrastructure Suite Design) working group.

We start with the most general topics (coding style, general principles), move on to more advanced topics (efficiency and maintainability, portable suites), and end with some pointers to future developments.

A good working knowledge of Cylc and Rose is assumed.

- Cylc: http://cylc.github.io/cylc/documentation.html
- Rose: http://metomi.github.io/rose/doc/rose.html

Note: *for non-Rose users: this document comes out of the Unified Model Consortium wherein Cylc is used within the Rose* suite management framework. *However, the bulk of the information in this guide is about Cylc suite design; which parts are Rose-specific should be clear from context.*

2 Style Guidelines

Coding style is largely subjective, but for collaborative development of complex systems it is important to settle on a clear and consistent style to avoid getting into a mess. The following style rules are recommended.

2.1 Tab Characters

Do not use tab characters. Tab width depends on editor settings, so a mixture of tabs and spaces in the same file can render to a mess.

Use `grep -InPr "\t" *` to find tabs recursively in files in a directory.

In *vim* use `%retab` to convert existing tabs to spaces, and set `expandtab` to automatically convert new tabs.

In *emacs* use *whitespace-cleanup*.

In *gedit*, use the *Draw Spaces* plugin to display tabs and spaces.

2.2 Trailing Whitespace

Trailing whitespace is untidy, it makes quick reformatting of paragraphs difficult, and it can result in hard-to-find bugs (space after intended line continuation markers).

To remove existing trailing whitespace in a file use a `sed` or `perl` one-liner:

```
$ perl -pi -e "s/ +$//g" /path/to/file
# or:
$ sed --in-place 's/[[:space:]]\+$//' path/to/file
```

Or do a similar search-and-replace operation in your editor. Editors like *vim* and *emacs* can also be configured to highlight or automatically remove trailing whitespace on the fly.

2.3 Indentation

Consistent indentation makes a suite definition more readable, it shows section nesting clearly, and it makes block re-indentation operations easier in text editors. Indent suite.rc syntax four spaces per nesting level:

2.3.1 Config Items

```
[SECTION]
    # A comment.
    title = the quick brown fox
    [[SUBSECTION]]
        # Another comment.
        a short item = value1
        a very very long item = value2
```

Don't align `item = value` pairs on the = character like this:

```
[SECTION]  # Avoid this.
             a short item = value1
    a very very long item = value2
```

or like this:

```
[SECTION]  # Avoid this.
    a short item          = value1
    a very very long item = value2
```

because the whole block may need re-indenting after a single change, which will pollute your revision history with spurious changes.

Comments should be indented to the same level as the section or item they refer to, and trailing comments should be preceded by two spaces, as shown above.

2.3.2 Script String Lines

Script strings are written verbatim to task job scripts so they should really be indented from the left margin:

```
[runtime]
    [[foo]]
        # Recommended.
        post-script = """
if [[ $RESULT == "bad" ]]; then
    echo Goodbye World!
    exit 1
fi"""
```

Indentation is *mostly* ignored by the bash interpreter, but is useful for readability. It is *mostly* harmless to indent internal script lines as if part of the Cylc syntax, or even out to the triple quotes:

```
[runtime]
    [[foo]]
        # OK, but...
        post-script = """
            if [[ $RESULT == "bad" ]]; then
                echo Goodbye World!
                exit 1
            fi"""
```

On parsing the triple quoted value, Cylc will remove any common leading whitespace from each line using the logic of Python's textwrap.dedent so the script block would end up being the same as the previous example. However, you should watch your line length (see 2.6) when you have many levels of indentations.

Take care when indenting here documents:

```
[runtime]
    [[foo]]
        script = """
        cat >> log.txt <<_EOF_
            The quick brown fox jumped
            over the lazy dog.
        _EOF_
                """
```

In the above, each line in `log.txt` would end up with 4 leading white spaces. The following will give you lines with no white spaces.

```
[runtime]
    [[foo]]
        script = """
        cat >> log.txt <<_EOF_
        The quick brown fox jumped
        over the lazy dog.
        _EOF_
                """
```

2.3.3 Graph String Lines

Multiline `graph` strings can be entirely free-form:

```
[scheduling]
    [[dependencies]]
        graph = """
     # Main workflow:
  FAMILY:succeed-all => bar & baz => qux

     # Housekeeping:
  qux => rose_arch => rose_prune"""
```

Whitespace is ignored in graph string parsing, however, so internal graph lines can be indented as if part of the suite.rc syntax, or even out to the triple quotes, if you feel it aids readability (but watch line length with large indents; see 2.6):

```
[scheduling]
    [[dependencies]]
        graph = """
           # Main workflow:
           FAMILY:succeed-all => bar & baz => qux

           # Housekeeping:
           qux => rose_arch => rose_prune"""
```

Both styles are acceptable; choose one and use it consistently.

2.3.4 Jinja2 Code

A suite.rc file with embedded Jinja2 code is essentially a Jinja2 program to generate a Cylc suite definition. It is not possible to consistently indent the Jinja2 as if it were part of the suite.rc syntax (which to the Jinja2 processor is just arbitrary text), so it should be indented from the left margin on its own terms:

```
[runtime]
    [[OPS]]
{% for T in OPS_TASKS %}
    {% for M in range(M_MAX) %}
    [[ops_{{T}}_{{M}}]]
        inherit = OPS
    {% endfor %}
{% endfor %}
```

2.4 Comments

Comments should be minimal, but not too minimal. If context and clear task and variable names will do, leave it at that. Extremely verbose comments tend to get out of sync with the code they describe, which can be worse than having no comments.

Avoid long lists of numbered comments - future changes may require mass renumbering.

Avoid page-width "section divider" comments, especially if they are not strictly limited to the standard line length (see 2.6).

Indent comments to the same level as the config items they describe.

2.5 Titles, Descriptions, And URLs

Document the suite and its tasks with `title`, `description`, and `url` items instead of comments. These can be displayed, or linked to, by the GUI at runtime.

2.6 Line Length And Continuation

Keep to the standard maximum line length of 79 characters where possible. Very long lines affect readability and make side-by-side diffs hard to view.

Backslash line continuation markers can be used anywhere in the suite.rc file but should be avoided if possible because they are easily broken by invisible trailing whitespace.

Continuation markers are not needed in graph strings where trailing trigger arrows imply line continuation:

```
[scheduling]
    [[dependencies]]
        # No line continuation marker is needed here.
        graph = """prep => one => two => three =>
                   four => five six => seven => eight"""
[runtime]
    [[MY_TASKS]]
    # A line continuation marker *is* needed here:
    [[one, two, three, four, five, six, seven, eight, nine, ten, \
      eleven, twelve, thirteen ]]
        inherit = MY_TASKS
```

2.7 Task Naming Conventions

Use `UPPERCASE` for family names and `lowercase` for tasks, so you can distinguish them at a glance.

Choose a convention for multi-component names and use it consistently. Put the most general name components first for natural grouping in the GUI, e.g. `obs_sonde`, `obs_radar` (not `sonde_obs` etc.)

Within your convention keep names as short as possible.

2.7.1 UM System Task Names

For UM System suites we recommend the following full task naming convention:

```
model_system_function[_member]
```

For example, `glu_ops_process_scatwind` where `glu` refers to the global (deterministic model) update run, `ops` is the system that owns the task, and `process_scatwind` is the function it performs. The optional `member` suffix is intended for use with ensembles as needed.

Within this convention keep names as short as possible, e.g. use `fcst` instead of `forecast`.

UM forecast apps should be given names that reflect their general science configuration rather than geographic domain, to allow use on other model domains without causing confusion.

2.8 Rose Config Files

Use `rose config-dump` to load and re-save new Rose .conf files. This puts the files in a standard format (ordering of lines etc.) to ensure that spurious changes aren't generated when you next use `rose edit`.

See also 4.3 on optional app config files.

3 Basic Principles

This section covers general principles that should be kept in mind when writing any suite. More advanced topics are covered later: *Efficiency And Maintainability* (section 4) and *Portable Suites* (section 5).

3.1 UTC Mode

Cylc has full timezone support if needed, but real time NWP suites should use UTC mode to avoid problems at the transition between local standard time and daylight saving time, and to enable the same suite to run the same way in different timezones.

```
[cylc]
    UTC mode = True
```

3.2 Fine Or Coarse-Grained Suites

Suites can have many small simple tasks, fewer large complex tasks, or anything in between. A task that runs many distinct processes can be split into many distinct tasks. The fine-grained approach is more transparent and it allows more task level concurrency and quicker failure recovery - you can rerun just what failed without repeating anything unnecessarily.

3.2.1 rose bunch

One caveat to our fine-graining advice is that submitting a large number of small tasks at once may be a problem on some platforms. If you have many similar concurrent jobs you can use `rose bunch` to pack them into a single task with incremental rerun capability: retriggering the task will rerun just the component jobs that did not successfully complete earlier.

3.3 Monolithic Or Interdependent Suites

When writing suites from scratch you may need to decide between putting multiple loosely connected sub-workflows into a single large suite, or constructing a more modular system of smaller suites that depend on each other through inter-suite triggering. Each approach has its pros and cons, depending on your requirements and preferences with respect to the complexity and manageability of the resulting system.

The `cylc gscan` GUI lets you monitor multiple suites at a time, and you can define virtual groups of suites that collapse into a single state summary.

3.3.1 Inter-Suite Triggering

A task in one suite can explicitly trigger off of a task in another suite. The full range of possible triggering conditions is supported, including custom message triggers. Remote triggering involves repeatedly querying ("polling") the remote suite run database, not the suite server program, so it works even if the other suite is down at the time.

There is special graph syntax to support triggering off of a task in another suite, or you can call the underlying `cylc suite-state` command directly in task scripting.

In real time suites you may want to use clock-triggers to delay the onset of inter-suite polling until roughly the expected completion time of the remote task.

3.4 Self-Contained Suites

All files generated by Cylc during a suite run are confined to the *suite run directory* `$HOME/cylc-run/<SUITE>`. However, Cylc has no control over the locations of the programs, scripts, and files, that are executed, read, or generated by your tasks at runtime. It is up to you to ensure that all of this is confined to the suite run directory too, as far as possible.

Self-contained suites are more robust, easier to work with, and more portable. Multiple instances of the same suite (with different suite names) should be able to run concurrently under the same user account without mutual interference.

3.4.1 Avoiding External Files

Suites that use external scripts, executables, and files beyond the essential system libraries and utilities are vulnerable to external changes: someone else might interfere with these files without telling you.

In some case you may need to symlink to large external files anyway, if space or copy speed is a problem, but otherwise suites with private copies of all the files they need are more robust.

3.4.2 Installing Files At Start-up

Use `rose suite-run` *file creation mode* or `R1` install tasks to copy files to the self-contained suite run directory at start-up. Install tasks are preferred for time-consuming installations because they don't slow the suite start-up process, they can be monitored in the GUI, they can run directly on target platforms, and you can rerun them later without restarting the suite. If you are using symbolic links to install files under your suite directory it is recommended that the linking should be set up to fail if the source is missing e.g. by using *mode=symlink+* for file installation in a rose app.

3.4.3 Confining Ouput To The Run Directory

Output files should be confined to the suite run directory tree. Then all output is easy to find, multiple instances of the same suite can run concurrently without interference, and other users should be able to copy and run your suite with few modifications. Cylc provides a `share` directory for generated files that are used by several tasks in a suite (see 3.16.3). Archiving tasks can use `rose arch` to copy or move selected files to external locations as needed (see 3.14).

3.5 Task Host Selection

At sites with multiple task hosts to choose from, use `rose host-select` to dynamically select appropriate task hosts rather than hard coding particular hostnames. This enables your suite to adapt to particular machines being down or heavily overloaded by selecting from a group of hosts based on a series of criteria. `rose host-select` will only return hosts that can be contacted by non-interactive SSH.

3.6 Task Scripting

Non-trivial task scripting should be held in external files rather than inlined in the suite.rc. This keeps the suite definition tidy, and it allows proper shell-mode text editing and independent testing of task scripts.

For automatic access by task jobs, task-specific scripts should be kept in Rose app bin directories, and shared scripts kept in (or installed to) the suite bin directory.

3.6.1 Coding Standards

When writing your own task scripts make consistent use of appropriate coding standards such as:

- PEP8 for Python - https://www.python.org/dev/peps/pep-0008/
- Google Shell Style Guide for Bash - https://google.github.io/styleguide/shell.xml

3.6.2 Basic Functionality

In consideration of future users who may not be expert on the internals of your suite and its tasks, all task scripts should:

- Print clear usage information if invoked incorrectly (and via the standard options `-h, --help`).
- Print useful diagnostic messages in case of error. For example, if a file was not found, the error message should contain the full path to the expected location.
- Always return correct shell exit status - zero for success, non-zero for failure. This is used by Cylc job wrapper code to detect success and failure and report it back to the suite server program.
- In shell scripts use `set -u` to abort on any reference to an undefined variable. If you really need an undefined variable to evaluate to an empty string, make it explicit: `FOO=${FOO:-}`.
- In shell scripts use `set -e` to abort on any error without having to failure-check each command explicitly.
- In shell scripts use `set -o pipefail` to abort on any error within a pipe line. Note that all commands in the pipe line will still run, it will just exit with the right most non-zero exit status.
- For examples and more details on the above three `set` commands, see https://vaneyckt.io/posts/safer_bash_scripts_with_set_euxo_pipefail/

3.7 Rose Apps

Rose apps allow all non-shared task configuration - which is not relevant to workflow automation - to be moved from the suite definition into app config files. This makes suites tidier and easier to understand, and it allows `rose edit` to provide a unified metadata-enhanced view of the suite and its apps (see 3.8).

Rose apps are a clear winner for tasks with complex configuration requirements. It matters less for those with little configuration, but for consistency and to take full advantage of `rose edit` it makes sense to use Rose apps for most tasks.

When most tasks are Rose apps, set the app-run command as a root-level default, and override it for the occasional non Rose app task:

```
[runtime]
    [[root]]
        script = rose task-run -v
    [[rose-app1]]
        #...
    [[rose-app2]]
        #...
    [[hello-world]]  # Not a Rose app.
        script = echo "Hello World"
```

3.8 Rose Metadata Compliance

Rose metadata drives page layout and sort order in `rose edit`, plus help information, input validity checking, macros for advanced checking and app version upgrades, and more.

To ensure the suite and its constituent applications are being run as intended it should be valid against any provided metadata: launch the `rose edit` GUI or run `rose macro --validate` on the command line to highlight any errors, and correct them prior to use. If errors are flagged incorrectly you should endeavour to fix the metadata.

When writing a new suite or application, consider creating metadata to facilitate ease of use by others.

3.9 Task Independence

Essential dependencies must be encoded in the suite graph, but tasks should not rely unnecessarily on the action of other tasks. For example, tasks should create their own output directories if they don't already exist, even if they would normally be created by an earlier task in the workflow. This makes it is easier to run tasks alone during development and testing.

3.10 Clock-Triggered Tasks

Tasks that wait on real time data should use clock-triggers to delay job submission until the expected data arrival time:

```
[scheduling]
    initial cycle point = now
    [[special tasks]]
        # Trigger 5 min after wall-clock time is equal to cycle point.
        clock-trigger = get-data(PT5M)
    [[dependencies]]
        [[[T00]]]
            graph = get-data => process-data
```

Clock-triggered tasks typically have to handle late data arrival. Task execution *retry delays* can be used to simply retrigger the task at intervals until the data is found, but frequently retrying small tasks probably should not go to a batch scheduler, and multiple task failures will be logged for what is a essentially a normal condition (at least it is normal until the data is really late).

Rather than using task execution retry delays to repeatedly trigger a task that checks for a file, it may be better to have the task itself repeatedly poll for the data (see 3.11 for example).

3.11 Rose App File Polling

Rose apps have built-in polling functionality to check repeatedly for the existence of files before executing the main app. See the `[poll]` section in Rose app config documentation. This is a good way to implement check-and-wait functionality in clock-triggered tasks (3.10), for example.

It is important to note that frequent polling may be bad for some filesystems, so be sure to configure a reasonable interval between polls.

3.12 Task Execution Time Limits

Instead of setting job wall clock limits directly in batch scheduler directives, use the `execution time limit` suite config item. Cylc automatically derives the correct batch scheduler directives from this, and it is also used to run `background` and `at` jobs via the `timeout` command, and to poll tasks that haven't reported in finished by the configured time limit.

3.13 Restricting Suite Activity

It may be possible for large suites to overwhelm a job host by submitting too many jobs at once:

- Large suites that are not sufficiently limited by real time clock triggering or inter-cycle dependence may generate a lot of *runahead* (this refers to Cylc's ability to run multiple cycles at once, restricted only by the dependencies of individual tasks).
- Some suites may have large families of tasks whose members all become ready at the same time.

These problems can be avoided with *runahead limiting* and *internal queues*, respectively.

3.13.1 Runahead Limiting

By default Cylc allows a maximum of three cycle points to be active at the same time, but this value is configurable:

```
[scheduling]
    initial cycle point = 2020-01-01T00
    # Don't allow any cycle interleaving:
    max active cycle points = 1
```

3.13.2 Internal Queues

Tasks can be assigned to named internal queues that limit the number of members that can be active (i.e. submitted or running) at the same time:

```
[scheduling]
    initial cycle point = 2020-01-01T00
    [[queues]]
        # Allow only 2 members of BIG_JOBS to run at once:
        [[[big_jobs_queue]]]
            limit = 2
            members = BIG_JOBS
    [[dependencies]]
        [[[T00]]]
            graph = pre => BIG_JOBS
[runtime]
    [[BIG_JOBS]]
    [[foo, bar, baz, ...]]
        inherit = BIG_JOBS
```

3.14 Suite Housekeeping

Ongoing cycling suites can generate an enormous number of output files and logs so regular housekeeping is very important. Special housekeeping tasks, typically the last tasks in each cycle, should be included to archive selected important files and then delete everything at some offset from the current cycle point.

The Rose built-in apps `rose_arch` and `rose_prune` provide an easy way to do this. They can be configured easily with file-matching patterns and cycle point offsets to perform various housekeeping operations on matched files.

3.15 Complex Jinja2 Code

The Jinja2 template processor provides general programming constructs, extensible with custom Python filters, that can be used to *generate* the suite definition. This makes it possible to write flexible multi-use suites with structure and content that varies according to various input switches. There is a cost to this flexibility however: excessive use of Jinja2 can make a suite hard to understand and maintain. It is difficult to say exactly where to draw the line, but we recommend erring on the side of simplicity and clarity: write suites that are easy to understand and therefore easy to modify for other purposes, rather than extremely complicated suites that attempt do everything out of the box but are hard to maintain and modify.

Note that use of Jinja2 loops for generating tasks is now deprecated in favour of built-in parameterized tasks - see 4.2.2.

3.16 Shared Configuration

Configuration that is common to multiple tasks should be defined in one place and used by all, rather than duplicated in each task. Duplication is a maintenance risk because changes have to be made consistently in several places at once.

3.16.1 Jinja2 Variables

In simple cases you can share by passing a Jinja2 variable to all the tasks that need it:

```
{% set JOB_VERSION = 'A23' %}
[runtime]
    [[foo]]
        script = run-foo --version={{JOB_VERSION}}
    [[bar]]
        script = run-bar --version={{JOB_VERSION}}
```

3.16.2 Inheritance

Sharing by inheritance of task families is recommended when more than a few configuration items are involved.

The simplest application of inheritance is to set set global defaults in the `[[runtime]][root]` namespace that is inherited by all tasks. However, this should only be done for settings that really are used by the vast majority of tasks. Over-sharing of via root, particularly of environment variables, is a maintenance risk because it can be very difficult to be sure which tasks are *using* which global variables.

Any `[runtime]` settings can be shared - scripting, host and batch scheduler configuration, environment variables, and so on - from single items up to complete task or app configurations. At the latter extreme, it is quite common to have several tasks that inherit the same complete job configuration followed by minor task-specific additions:

```
[runtime]
    [[FILE-CONVERT]]
        script = convert-netcdf
        #...
```

```
    [[convert-a]]
        inherit = FILE-CONVERT
        [[[environment]]]
              FILE_IN = file-a
    [[convert-b]]
        inherit = FILE-CONVERT
        [[[environment]]]
              FILE_IN = file-b
```

Inheritance is covered in more detail from an efficiency perspective in Section 4.1.

3.16.3 Shared Task IO Paths

If one task uses files generated by another task (and both see the same filesystem) a common IO path should normally be passed to both tasks via a shared environment variable. As far as Cylc is concerned this is no different to other shared configuration items, but there are some additional aspects of usage worth addressing here.

Primarily, for self-containment (see 3.4) shared IO paths should be under the *suite share directory*, the location of which is passed to all tasks as `$CYLC_SUITE_SHARE_PATH`.

The `rose task-env` utility can provide additional environment variables that refer to static and cyclepoint-specific locations under the suite share directory.

```
[runtime]
    [[my-task]]
        env-script = $(eval rose task-env -T P1D -T P2D)
```

For a current cycle point of `20170105` this will make the following variables available to tasks:

```
ROSE_DATA=$CYLC_SUITE_SHARE_PATH/data
ROSE_DATAC=$CYLC_SUITE_SHARE_PATH/cycle/20170105
ROSE_DATACP1D=$CYLC_SUITE_SHARE_PATH/cycle/20170104
ROSE_DATACP2D=$CYLC_SUITE_SHARE_PATH/cycle/20170103
```

Subdirectories of `$ROSE_DATAC` etc. should be agreed between different sub-systems of the suite; typically they are named for the file-generating tasks, and the file-consuming tasks should know to look there.

The share-not-duplicate rule can be relaxed for shared files whose names are agreed by convention, so long as their locations under the share directory are proper shared suite variables. For instance the Unified Model uses a large number of files whose conventional names (`glu_snow`, for example) can reasonably be expected not to change, so they are typically hardwired into app configurations (as `$ROSE_DATA/glu_snow`, for example) to avoid cluttering the suite definition.

Here two tasks share a workspace under the suite share directory by inheritance:

```
# Sharing an I/O location via inheritance.
[scheduling]
    [[dependencies]]
        graph = write_data => read_data
[runtime]
    [[root]]
        env-script = $(eval rose task-env)
    [[WORKSPACE]]
        [[[environment]]]
            DATA_DIR = ${ROSE_DATA}/png
    [[write_data]]
        inherit = WORKSPACE
        script = """
mkdir -p $DATA_DIR
write-data.exe -o ${DATA_DIR}"""
    [[read_data]]
```

```
        inherit = WORKSPACE
        script = read-data.exe -i ${DATA_DIR}
```

In simple cases where an appropriate family does not already exist paths can be shared via Jinja variables:

```
# Sharing an I/O location with Jinja2.
{% set DATA_DIR = '$ROSE_DATA/stuff' %}
[scheduling]
    [[dependencies]]
        graph = write_data => read_data
[runtime]
    [[write_data]]
        script = """
mkdir -p {{DATA_DIR}}
write-data.exe -o {{DATA_DIR}}"""
    [[read_data]]
        script = read-data.exe -i {{DATA_DIR}}
```

For completeness we note that it is also possible to configure multiple tasks to use the same work directory so they can all share files in `$PWD`. (Cylc executes task jobs in special work directories that by default are unique to each task). This may simplify the suite slightly, and it may be useful if you are unfortunate enough to have executables that are designed for IO in `$PWD`, *but it is not recommended.* There is a higher risk of interference between tasks; it will break `rose task-run` incremental file creation mode; and `rose task-run --new` will in effect delete the work directories of tasks other than its intended target.

```
# Shared work directory: tasks can read and write in $PWD - use with caution!
[scheduling]
    initial cycle point = 2018
    [[dependencies]]
        [[[P1Y]]]
            graph = write_data => read_data
[runtime]
    [[WORKSPACE]]
        work sub-directory = $CYLC_TASK_CYCLE_POINT/datadir
    [[write_data]]
        inherit = WORKSPACE
        script = write-data.exe
    [[read_data]]
        inherit = WORKSPACE
        script = read-data.exe
```

3.17 Varying Behaviour By Cycle Point

To make a cycling job behave differently at different cycle points you *could* use a single task with scripting that reacts to the cycle point it finds itself running at, but it is better to use different tasks (in different cycling sections) that inherit the same base job configuration. This results in a more transparent suite that can be understood just by inspecting the graph:

```
# Run the same job differently at different cycle points.
[scheduling]
    initial cycle point = 2020-01-01T00
    [[dependencies]]
        [[[T00]]]
            graph = pre => long_fc => post
        [[[T12]]]
            graph = pre => short_fc => post
[runtime]
    [[MODEL]]
        script = run-model.sh
    [[long_fc]]
        inherit = MODEL
        [[[job]]]
```

```
            execution time limit = PT30M
        [[[environment]]]
            RUN_LEN = PT48H
    [[short_fc]]
        inherit = MODEL
        [[[job]]]
            execution time limit = PT10M
        [[[environment]]]
            RUN_LEN = PT12H
```

The few differences between `short_fc` and `long_fc`, including batch scheduler resource requests, can be configured after common settings are inherited.

3.17.1 At Start-Up

Similarly, if a cycling job needs special behaviour at the initial (or any other) cycle point, just use a different logical task in an `R1` graph and have it inherit the same job as the general cycling task, not a single task with scripting that behaves differently if it finds itself running at the initial cycle point.

3.18 Automating Failure Recovery

3.18.1 Job Submission Retries

When submitting jobs to a remote host, use job submission retries to automatically resubmit tasks in the event of network outages. Note this is distinct from job retries for job execution failure (just below).

Job submission retries should normally be host (or host-group for `rose host-select`) specific, not task-specific, so configure them in a host (or host-group) specific family. The following suite.rc fragment configures all HPC jobs to retry on job submission failure up to 10 times at 1 minute intervals, then another 5 times at 1 hour intervals:

```
[runtime]
    [[HPC]]  # Inherited by all jobs submitted to HPC.
        [[[job]]]
            submission retry delays = 10*PT1M, 5*PT1H
```

3.18.2 Job Execution Retries

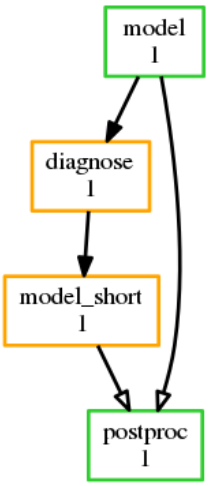
Automatic retry on job execution failure is useful if you have good reason to believe that a simple retry will usually succeed. This may be the case if the job host is known to be flaky, or if the job only ever fails for one known reason that can be fixed on a retry. For example, if a model fails occasionally with a numerical instability that can be remedied with a short timestep rerun, then an automatic retry may be appropriate:

```
[runtime]
    [[model]]
        script = """
if [[ $CYLC_TASK_TRY_NUMBER > 1 ]]; then
    SHORT_TIMESTEP=true
else
    SHORT_TIMESTEP=false
fi
model.exe"""
        [[[job]]]
            execution retry delays = 1*PT0M
```

3.18.3 Failure Recovery Workflows

For recovery from failures that require explicit diagnosis you can configure alternate routes through the workflow, together with *suicide triggers* that remove the unused route. In the following example, if the model fails a diagnosis task will trigger; if it determines the cause of the failure is a known numerical instability (e.g. by parsing model job logs) it will succeed, triggering a short timestep run. Postprocessing can proceed from either the original or the short-step model run, and suicide triggers remove the unused path from the workflow:

```
[scheduling]
    [[dependencies]]
        graph = """
            model | model_short => postproc
            model:fail => diagnose => model_short
              # Clean up with suicide triggers:
            model => ! diagnose & ! model_short
            model_short => ! model"""
```



3.19 Include Files

Include-files should not be overused, but they can sometimes be useful (e.g. see Portable Suites 5):

```
#...
{% include 'inc/foo.rc' %}
```

(Technically this inserts a Jinja2-rendered file template). Cylc also has a native include mechanism that pre-dates Jinja2 support and literally inlines the include-file:

```
#...
%include 'inc/foo.rc'
```

The two methods normally produce the same result, but use the Jinja2 version if you need to construct an include-file name from a variable (because Cylc include-files get inlined before Jinja2 processing is done):

```
#...
{% include 'inc/' ~ SITE ~ '.rc' %}
```

4 Efficiency And Maintainability

Efficiency (in the sense of *economy of suite definition*) and maintainability go hand in hand. This section describes techniques for clean and efficient construction of complex workflows that are easy to understand, maintain, and modify.

4.1 The Task Family Hierarchy

A properly designed family hierarchy fulfills three purposes in Cylc:

- efficient sharing of all configuration common to groups of related tasks
- efficient bulk triggering, for clear scheduling graphs
- clean suite visualization and monitoring, because families are collapsible in the GUIs

4.1.1 Sharing By Inheritance

Duplication is a maintenance risk because changes have to be repeated in multiple places without mistakes. On the other hand, unnecessary sharing of items via global variables is also bad because it is hard to be sure which tasks are using which variables. A properly designed runtime inheritance hierarchy can give every task exactly what it needs, and nothing that it doesn't need.

If a group of related tasks has some configuration in common, it can be factored out into a task family inherited by all.

```
[runtime]
    [[OBSPROC]]
        # Settings common to all obs processing tasks.
    [[obs1]]
        inherit = OBSPROC
    [[obs2]]
        inherit = OBSPROC
```

If several families have settings in common, they can in turn can inherit from higher-level families.

Multiple inheritance allows efficient sharing even for overlapping categories of tasks. For example consider that some obs processing tasks in the following suite run parallel jobs and some serial:

```
[runtime]
    [[SERIAL]]
        # Serial job settings.
    [[PARALLEL]]
        # Parallel job settings.
    [[OBSPROC]]
        # Settings for all obs processing tasks.
    [[obs1, obs2, obs3]]
        # Serial obs processing tasks.
        inherit = OBSPROC, SERIAL
    [[obs4, obs5]]
        # Parallel obs processing tasks.
        inherit = OBSPROC, PARALLEL
```

Note that suite parameters should really be used to define family members efficiently - see Section 4.2.

Cylc provides tools to help make sense of your inheritance hierarchy:

- `cylc graph -n/--namespaces` - plot the full multiple inheritance graph (not the dependency graph)
- `cylc get-config SUITE` - print selected sections or items after inheritance processing

- `cylc graph SUITE` - plot the dependency graph, with collapsible first-parent families (see 4.1.4)
- `cylc list -t/--tree SUITE` - print the first-parent inheritance hierarchy
- `cylc list -m/--mro SUITE` - print the inheritance precedence order for each runtime namespace

4.1.2 Family Triggering

Task families can be used to simplify the scheduling graph wherever many tasks need to trigger at once:

```
[scheduling]
    [[dependencies]]
        graph = pre => MODELS
[runtime]
    [[MODELS]]
    [[model1, model2, model3, ...]]
        inherit = MODELS
```

To trigger *off of* many tasks at once, family names need to be qualified by `<state>-all` or `<state>-any` to indicate the desired member-triggering semantics:

```
[scheduling]
    [[dependencies]]
        graph = """pre => MODELS
                MODELS:succeed-all => post"""
```

Note that this can be simplified further because Cylc ignores trigger qualifiers like `:succeed-all` on the right of trigger arrows to allow chaining of dependencies:

```
[scheduling]
    [[dependencies]]
        graph = pre => MODELS:succeed-all => post
```

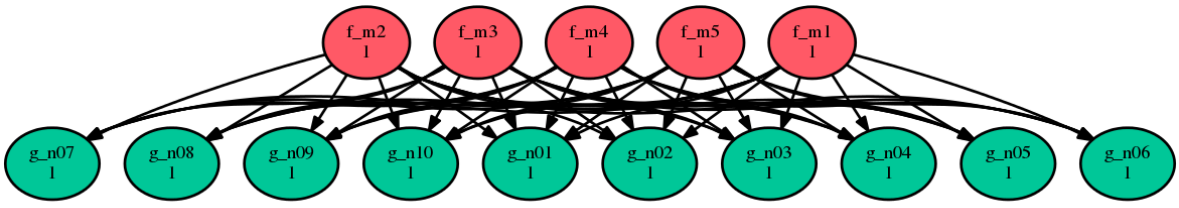
4.1.3 Family-to-Family Triggering

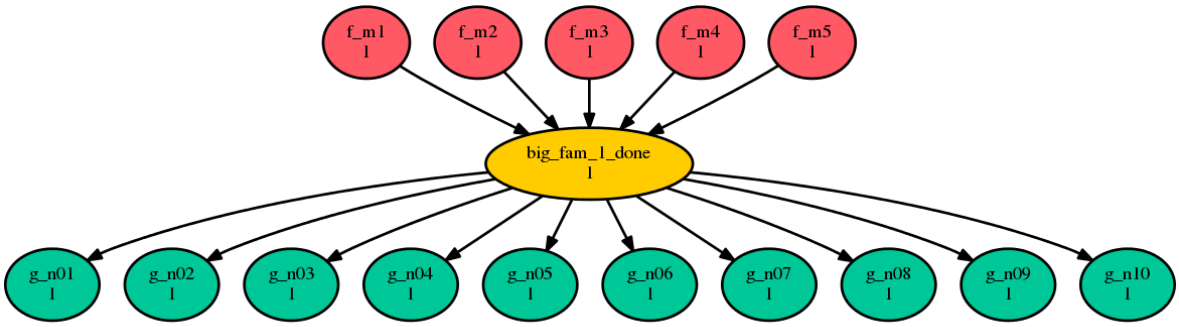
```
[scheduling]
    [[dependencies]]
        graph = BIG_FAM_1:succeed-all => BIG_FAM_2
```

This means every member of `BIG_FAM_2` depends on every member of `BIG_FAM_1` succeeding. For very large families this can create so many dependencies that it affects the performance of Cylc at run time, as well as cluttering graph visualizations with unnecessary edges. Instead, interpose a dummy task that signifies completion of the first family:

```
[scheduling]
    [[dependencies]]
        graph = BIG_FAM_1:succeed-all => big_fam_1_done => BIG_FAM_2
```

For families with `M` and `N` members respectively, this reduces the number of dependencies from `M*N` to `M+N` without affecting the scheduling.





4.1.4 Task Families And Visualization

First parents in the inheritance hierarchy double as collapsible summary groups for visualization and monitoring. Tasks should generally be grouped into visualization families that reflect their logical purpose in the suite rather than technical detail such as inherited job submission or host settings. So in the example under Section 4.1.1 above all `obs<n>` tasks collapse into `OBSPROC` but not into `SERIAL` or `PARALLEL`.

If necessary you can introduce new namespaces just for visualization:

```
[runtime]
    [[MODEL]]
        # (No settings here - just for visualization).
    [[model1, model2]]
        inherit = MODEL, HOSTX
    [[model3, model4]]
        inherit = MODEL, HOSTY
```

To stop a solo parent being used in visualization, demote it to secondary with a null parent like this:

```
[runtime]
    [[SERIAL]]
    [[foo]]
        # Inherit settings from SERIAL but don't use it in visualization.
        inherit = None, SERIAL
```

4.2 Generating Tasks Automatically

Groups of tasks that are closely related such as an ensemble of model runs or a family of obs processing tasks, or sections of workflow that are repeated with minor variations, can be generated automatically by iterating over some integer range (e.g. `model<n>` for `n = 1..10`) or list of strings (e.g. `obs<type>` for `type = ship, buoy, radiosonde, ...`).

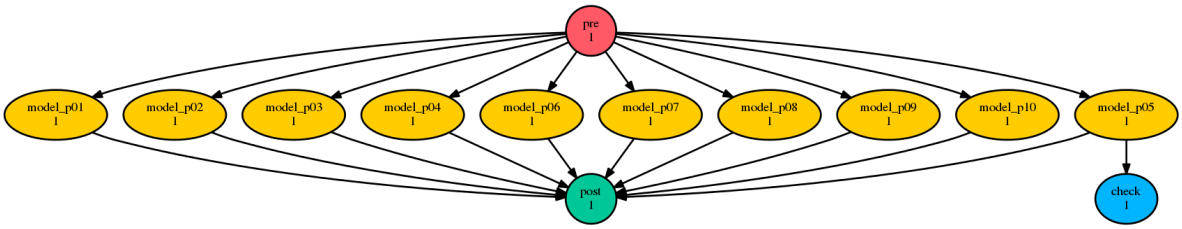
4.2.1 Jinja2 Loops

Task generation was traditionally done in Cylc with explicit Jinja2 loops, like this:

```
# Task generation the old way: Jinja2 loops (NO LONGER RECOMMENDED!)
{% set PARAMS = range(1,11) %}
[scheduling]
    [[dependencies]]
        graph = """
{% for P in PARAMS %}
      pre => model_p{{P}} => post
      {% if P == 5 %}
          model_p{{P}} => check
```

```
      {% endif %}
{% endfor %}    """
[runtime]
{% for P in PARAMS %}
    [[model_p{{P}}]]
        script = echo "my parameter value is {{P}}"
    {% if P == 1 %}
        # special case...
    {% endif %}
{% endfor %}
```

Unfortunately this makes a mess of the suite definition, particularly the scheduling graph, and it gets worse with nested loops over multiple parameters.



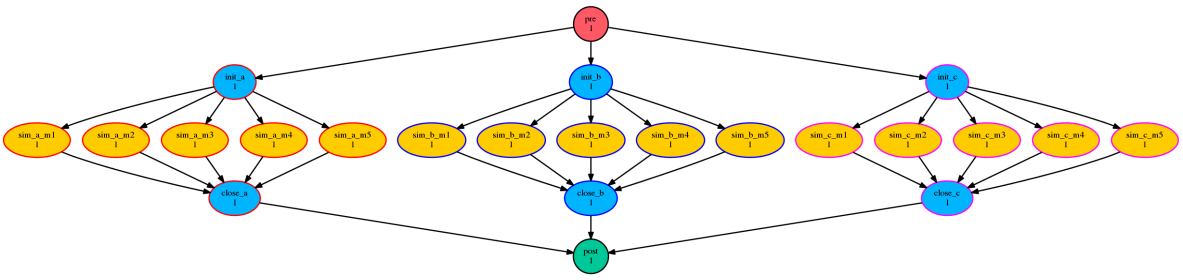
4.2.2 Parameterized Tasks

Cylc-6.11 introduced built-in *suite parameters* for generating tasks without destroying the clarity of the base suite definition. Here's the same example using suite parameters instead of Jinja2 loops:

```
# Task generation the new way: suite parameters.
[cylc]
    [[parameters]]
        p = 1..10
[scheduling]
    [[dependencies]]
        graph = """pre => model<p> => post
                   model<p=5> => check"""
[runtime]
    [[model<p>]]
        script = echo "my parameter value is ${CYLC_TASK_PARAM_p}"
    [[model<p=7>]]
        # special case ...
```

Here `model<p>` expands to `model_p7` for `p=7`, and so on, via the default expansion template for integer-valued parameters, but custom templates can be defined if necessary. Parameters can also be defined as lists of strings, and you can define dependencies between different values: `chunk<p-1> => chunk<p>`. Here's a multi-parameter example:

```
[cylc]
    [[parameters]]
        run = a, b, c
        m = 1..5
[scheduling]
    [[dependencies]]
        graph = pre => init<run> => sim<run,m> => close<run> => post
[runtime]
    [[sim<run,m>]]
```



If family members are defined by suite parameters, then parameterized trigger expressions are equivalent to family `:<state>-all` triggers. For example, this:

```
[cylc]
    [[parameters]]
        n = 1..5
[scheduling]
    [[dependencies]]
        graph = pre => model<n> => post
[runtime]
    [[MODELS]]
    [[model<n>]]
        inherit = MODELS
```

is equivalent to this:

```
[cylc]
    [[parameters]]
        n = 1..5
[scheduling]
    [[dependencies]]
        graph = pre => MODELS:succeed-all => post
[runtime]
    [[MODELS]]
    [[model<n>]]
        inherit = MODELS
```

(but future plans for family triggering may make the second case more efficient for very large families).

For more information on parameterized tasks see the Cylc user guide.

4.3 Optional App Config Files

Closely related tasks with few configuration differences between them - such as multiple UM forecast and reconfiguration apps in the same suite - should use the same Rose app configuration with the differences supplied by optional configs, rather than duplicating the entire app for each task.

Optional app configs should be valid on top of the main app config and not dependent on the use of other optional app configs. This ensures they will work correctly with macros and can therefore be upgraded automatically.

Note: *Currently optional configs don't work very well with UM STASH configuration - see Section 6.2.*

Optional app configs can be loaded by command line switch:

```
rose task-run -O key1 -O key2
```

or by environment variable:

```
ROSE_APP_OPT_CONF_KEYS = key1 key2
```

The environment variable is generally preferred in suites because you don't have to repeat and override the root-level script configuration:

```
[runtime]
    [[root]]
        script = rose task-run -v
    [[foo]]
        [[[environment]]]
            ROSE_APP_OPT_CONF_KEYS = key1 key2
```

5 Portable Suites

A *portable* or *interoperable* suite can run "out of the box" at different sites, or in different environments such as research and operations within a site. For convenience we just use the term *site portability*.

Lack of portability is a major barrier to collaborative development when sites need to run more or less the same workflow, because it is very difficult to translate changes manually between large, complicated suites.

Most suites are riddled with site-specific details such as local build configurations, file paths, host names, and batch scheduler directives, etc.; but it is possible to cleanly factor all this out to make a portable suite. Significant variations in workflow structure can even be accommodated quite easily. If the site workflows are *too different*, however, you may decide that it is appropriate for each site to maintain separate suites.

The recommended way to do this, which we expand on below, is:

- Put all site-specific settings in include-files loaded at the end of a generic "core" suite definition.
- Use "optional optional" app config files for site-specific variations in the core suite's Rose apps.
- (Make minimal use of inlined site switches too, if necessary).
- When referencing files, reference them within the suite structure and use an install task to link external files in.

The result should actually be tidier than the original in one respect: all the messy platform-specific resource directives etc., will be hidden away in the site include-files.

5.1 The Jinja2 SITE Variable

First a suite Jinja2 variable called `SITE` should be set to the site name, either in `rose-suite.conf`, or in the suite definition itself (perhaps automatically, by querying the local environment in some way).

```
#!Jinja2
{% set SITE = "niwa" %}
#...
```

This will be used to select site-specific configuration, as described below.

5.2 Site Include-Files

If a section heading in a suite.rc file is repeated the items under it simply add to or override those defined under the same section earlier in the file (but note Section 6.1). For example, this task definition:

```
[runtime]
    [[foo]]
        script = run-foo.sh
        [[[remote]]]
            host = hpc1.niwa.co.nz
```

can equally be written like this:

```
[runtime]  # Part 1 (site-agnostic).
    [[foo]]
        script = run-foo.sh
[runtime]  # Part 2 (site-specific).
    [[foo]]
        [[[remote]]]
            host = hpc1.niwa.co.nz
```

(Note that if Part 2 had also defined `script` the new value would override the original. It can sometimes be useful to set a widely used default and override it in a few cases, but be aware that this can make it more difficult to determine the origin of affected values.)

In this way all site-specific `[runtime]` settings, with their respective sub-section headings, can be moved to the end of the file, and then out into an include-file (file inclusion is essentially just literal inlining):

```
#...
{% set SITE = "niwa" %}

# Core site-agnostic settings:
#...
[runtime]
    [[foo]]
        script = run-foo.sh
#...

# Site-specific settings:
{% include 'site/' ~ SITE ~ '.rc' %}
```

where the site include-file `site/niwa.rc` contains:

```
# site/niwa.rc
[runtime]
    [[foo]]
        [[[remote]]]
            host = hpc1.niwa.co.nz
```

5.3 Site-Specific Graphs

Repeated `graph` strings under the same graph section headings are always additive (graph strings are the only exception to the normal repeat item override semantics). So, for instance, this graph:

```
[scheduling]
    initial cycle point = 2025
    [[dependencies]]
        [[[P1Y]]]
            graph = "pre => model => post => niwa_archive"
```

can be written like this:

```
[scheduling]
    initial cycle point = 2025
    [[dependencies]]
        [[[P1Y]]]
            graph = "pre => model => post"
        [[[P1Y]]]
            graph = "post => niwa_archive"
```

and again, the site-specific part can be taken out to a site include-file:

```
#...
{% set SITE = "niwa" %}

# Core site-agnostic settings.
#...
[scheduling]
    initial cycle point = 2025
    [[dependencies]]
        [[[P1Y]]]
            graph = "pre => model => post"
#...
# Site-specific settings:
{% include 'site/' ~ SITE ~ '.rc' %}
```

where the site include-file `site/niwa.rc` contains:

```
# site/niwa.rc
[scheduling]
    [[dependencies]]
        [[[P1Y]]]
            graph = "post => niwa_archive"
```

Note that the site-file graph needs to define the dependencies of the site-specific tasks, and thus their points of connection to the core suite - which is why the core task `post` appears in the graph here (if `post` had any site-specific runtime settings, to get it to run at this site, they would also be in the site-file).

5.4 Inlined Site-Switching

It may be tempting to use inlined switch blocks throughout the suite instead of site include-files, but *this is not recommended* - it is verbose and untidy (the greater the number of supported sites, the bigger the mess) and it exposes all site configuration to all users:

```
#...
[runtime]
    [[model]]
        script = run-model.sh
{# Site switch blocks not recommended:#}
{% if SITE == 'niwa' %}
        [[[job]]]
            batch system = loadleveler
        [[[directives]]]
            # NIWA Loadleveler directives...
{% elif SITE == 'metoffice' %}
        [[[job]]]
            batch system = pbs
        [[[directives]]]
            # Met Office PBS directives...
{% elif SITE == ... %}
            #...
{% else %}
    {{raise('Unsupported site: ' ~ SITE)}}
{% endif %}
    #...
```

Inlined switches can be used, however, to configure exceptional behaviour at one site without requiring the other sites to duplicate the default behaviour. But be wary of accumulating too many of these switches:

```
# (core suite.rc file)
#...
{% if SITE == 'small' %}
   {# We can't run 100 members... #}
   {% set ENSEMBLE_SIZE = 25 %}
{% else %}
   {# ...but everyone else can! #}
   {% set ENSEMBLE_SIZE = 100 %}
{% endif %}
#...
```

Inlined switches can also be used to temporarily isolate a site-specific change to a hitherto non site-specific part of the suite, thereby avoiding the need to update all site include-files before getting agreement from the suite owner and collaborators.

5.5 Site-Specific Suite Variables

It can sometimes be useful to set site-specific values of suite variables that aren't exposed to users via `rose-suite.conf`. For example, consider a suite that can run a special post-processing workflow of some kind at sites where IDL is available. The IDL-dependence switch can be set per site like this:

```
#...
{% from SITE ~ '-vars.rc' import HAVE_IDL, OTHER_VAR %}
graph = """
  pre => model => post
{% if HAVE_IDL %}
      post => idl-1 => idl-2 => idl-3
{% endif %}
        """
```

where for `SITE = niwa` the file `niwa-vars.rc` contains:

```
{# niwa-vars.rc #}
{% set HAVE_IDL = True %}
{% set OTHER_VAR = "the quick brown fox" %}
```

Note we are assuming there are significantly fewer options (IDL or not, in this case) than sites, otherwise the IDL workflow should just go in the site include-files of the sites that need it.

5.6 Site-Specific Optional Suite Configs

During development and testing of a portable suite you can use an optional Rose suite config file to automatically set site-specific suite inputs and thereby avoid the need to make manual changes every time you check out and run a new version. The site switch itself has to be set of course, but there may be other settings too such as model parameters for a standard local test domain. Just put these settings in `opt/rose-suite-niwa.conf` (for site "niwa") and run the suite with `rose suite-run -O niwa`.

5.7 Site-Agnostic File Paths in App Configs

Where possible apps should be configured to reference files within the suite structure itself rather than outside of it. This makes the apps themselves portable and it becomes the job of the install task to ensure all required source files are available within the suite structure e.g. via symlink into the share directory. Additionally, by moving the responsibility of linking files into the suite to an install task you gain the added benefit of knowing if a file is missing at the start of a suite rather than part way into a run.

5.8 Site-Specific Optional App Configs

Typically a few but not all apps will need some site customization, e.g. for local archive configuration, local science options, or whatever. To avoid explicit site-customization of individual task-run command lines use Rose's built-in *optional optional app config* capability:

```
[runtime]
    [[root]]
        script = rose task-run -v -O '({{SITE}})'
```

Normally a missing optional app config is considered to be an error, but the round parentheses here mean the named optional config is optional - i.e. use it if it exists, otherwise ignore.

With this setting in place we can simply add a `opt/rose-app-niwa.conf` to any app that needs customization at `SITE = niwa`.

5.9 An Example

The following small suite is not portable because all of its tasks are submitted to a NIWA HPC host; two task are entirely NIWA-specific in that they respectively install files from a local database and upload products to a local distribution system; and one task runs a somewhat NIWA-specific configuration of a model. The remaining tasks are site-agnostic apart from local job host and batch scheduler directives.

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 2017-01-01
    [[dependencies]]
        [[[R1]]]
            graph = install_niwa => preproc
        [[[P1D]]]
            graph = """
                preproc & model[-P1D] => model => postproc => upload_niwa
                postproc => idl-1 => idl-2 => idl-3"""
[runtime]
    [[root]]
        script = rose task-run -v
    [[HPC]]  # NIWA job host and batch scheduler settings.
        [[[remote]]]
            host = hpc1.niwa.co.nz
        [[[job]]]
            batch system = loadleveler
        [[[directives]]]
            account_no = NWP1623
            class = General
            job_type = serial  # (most jobs in this suite are serial)
    [[install_niwa]]  # NIWA-specific file installation task.
        inherit = HPC
    [[preproc]]
        inherit = HPC
    [[model]]  # Run the model on a local test domain.
        inherit = HPC
        [[[directives]]]  # Override the serial job_type setting.
            job_type = parallel
        [[[environment]]]
            SPEED = fast
    [[postproc]]
        inherit = HPC
    [[upload_niwa]]  # NIWA-specific product upload.
        inherit = HPC
```

To make this portable, refactor it into a core suite.rc file that contains the clean site-independent workflow configuration and loads all site-specific settings from an include-file at the end:

```
# suite.rc: CORE SITE-INDEPENDENT CONFIGURATION.
{% set SITE = 'niwa' %}
{% from 'site/' ~ SITE ~ '-vars.rc' import HAVE_IDL %}
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 2017-01-01
    [[dependencies]]
        [[[P1D]]]
            graph = """
preproc & model[-P1D] => model => postproc
{% if HAVE_IDL %}
    postproc => idl-1 => idl-2 => idl-3
{% endif %}
                    """
[runtime]
    [[root]]
        script = rose task-run -v -O '({{SITE}})'
```

```
    [[preproc]]
        inherit = HPC
    [[preproc]]
        inherit = HPC
    [[model]]
        inherit = HPC
        [[[environment]]]
            SPEED = fast
{% include 'site/' ~ SITE ~ '.rc' %}
```

plus site files `site/niwa-vars.rc`:

```
# site/niwa-vars.rc: NIWA SITE SETTINGS FOR THE EXAMPLE SUITE.
{% set HAVE_IDL = True %}
```

and `site/niwa.rc`:

```
# site/niwa.rc: NIWA SITE SETTINGS FOR THE EXAMPLE SUITE.
[scheduling]
    [[dependencies]]
        [[[R1]]]
            graph = install_niwa => preproc
        [[[P1D]]]
            graph = postproc => upload_niwa
[runtime]
    [[HPC]]
        [[[remote]]]
            host = hpc1.niwa.co.nz
        [[[job]]]
            batch system = loadleveler
        [[[directives]]]
            account_no = NWP1623
            class = General
            job_type = serial  # (most jobs in this suite are serial)
    [[install_niwa]]  # NIWA-specific file installation.
    [[model]]
        [[[directives]]]  # Override the serial job_type setting.
            job_type = parallel
    [[upload_niwa]]  # NIWA-specific product upload.
```

and finally, an optional app config file for the local model domain:

```
app/model/rose-app.conf  # Main app config.
app/model/opt/rose-app-niwa.conf  # NIWA site settings.
```

Some points to note:

- It is straightforward to extend support to a new site by copying an existing site file(s) and adapting it to the new job host and batch scheduler etc.
- Batch system directives should be considered site-specific unless all supported sites have the same batch system and the same host architecture (including CPU clock speed and memory size etc.).
- We've assumed that all tasks run on a single HPC host at both sites. If that's not a valid assumption the `HPC` family inheritance relationships would have to become site-specific.
- Core task runtime configuration aren't needed in site files at all if their job host and batch system settings can be defined in common families that are (`HPC` in this case).

5.10 Collaborative Development Model

Official releases of a portable suite should be made from the suite trunk.

Changes should be developed on feature branches so as not to affect other users of the suite.

Site-specific changes shouldn't touch the core suite.rc file, just the relevant site include-file, and therefore should not need close scrutiny from other sites.

Changes to the core suite.rc file should be agreed by all stakeholders, and should be carefully checked for effects on site include-files:

- Changing the name of tasks or families in the core suite may break sites that add configuration to the original runtime namespace.
- Adding new tasks or families to the core suite may require corresponding additions to the site files.
- Deleting tasks or families from the core suite may require corresponding parts of the site files to be removed. And also, check for site-specific triggering off of deleted tasks or families.

However, if the owner site has to get some changes into the trunk before all collaborating sites have time to test them, version control will of course protect those lagging behind from any immediate ill effects.

When a new feature is complete and tested at the developer's site, the suite owner should check out the branch, review and test it, and if necessary request that other sites do the same and report back. The owner can then merge the new feature to the trunk once satisfied.

All planning and discussion associated with the change should be documented on MOSRS Trac tickets associated with the suite.

5.11 Research-To-Operations Transition

Under this collaborative development model it is *possible* to use the same suite in research and operations, largely eliminating the difficult translation between the two environments. Where appropriate, this can save a lot of work.

Operations-specific parts of the suite should be factored out (as for site portability) into include-files that are only loaded in the operational environment. Improvements and upgrades can be developed on feature branches in the research environment. Operations staff can check out completed feature branches for testing in the operational environment before merging to trunk or refering back to research if problems are found. After sufficient testing the new suite version can be deployed into operations.

Note: *This obviously glosses over the myriad complexities of the technical and scientific testing and validation of suite upgrades; it merely describes what is possible from a suite design and collaborative development perspective.*

6 Roadmap

Several planned future developments in Rose and Cylc may have an impact on suite design.

6.1 List Item Override In Site Include-Files

A few Cylc config items hold lists of task (or family) names, e.g.:

```
[scheduling]
    [[special tasks]]
        clock-trigger = get-data-a, get-data-b
    #...
#...
```

Currently a repeated config item completely overrides a previously set value (apart from graph strings which are always additive). This means a site include-file (for example) can't add a new site-specific clock-triggered task without writing out the complete list of all clock-triggered tasks in the suite, which breaks the otherwise clean separation into core and site files.

In the future we plan to support add, subtract, unset, and override semantics for all items - see https://github.com/cylc/cylc/issues/1363.

6.2 UM STASH in Optional App Configs

A caveat to the advice on use of option app configs in Section 4.3: in general you might need the ability to turn off or modify some STASH requests in the main app, not just add additional site-specific STASH. But overriding STASH in optional configs is fragile because STASH namelists names are automatically generated from a *hash* of the precise content of the namelist. This makes it possible to uniquely identify the same STASH requests in different apps, but if any detail of a STASH request changes in a main app its namelist name will change and any optional configs that refer to it will become divorced from their intended target.

Until this problem is solved we recommend that:

- All STASH in main UM apps should be grouped into sensible *packages* that can be turned on and off in optional configs without referencing the individual STASH request namelists.
- Or all STASH should be held in optional site configs and none in the main app. Note however that STASH is difficult to configure outside of `rose edit`, and the editor does not yet allow you to edit optional configs - see https://github.com/metomi/rose/issues/1685.

6.3 Modular Suite Design

The modular suite design concept is that we should be able to import common workflow segments at install time rather than duplicating them in each suite: https://github.com/cylc/cylc/issues/1829. The content of a suite module will be encapsulated in a protected namespace to avoid clashing with the importing suite, and selected inputs and outputs exposed via a proper interface.

This should aid portable suite design too by enabling site-specific parts of a workflow (local product generation for example) to be stored and imported on-site rather than polluting the source and revision control record of the core suite that everyone sees.

We note that this can already be done to a limited extent by using `rose suite-run` to install suite.rc fragments from an external location. However, as a literal inlining mechanism with no encapsulation or

interface, the internals of the "imported" fragments would have to be compatible with the suite definition in every respect.

See also 3.3 on modular *systems of suites* connected by inter-suite triggering.