Cylc Tutorial

Release 2019.01.0

Metomi

Jan 25, 2019

CONTENTS

Cylc is a workflow engine for running suites of inter-dependent jobs.

This section will cover the Cylc framework and writing basic Cylc suites.

CHAPTER

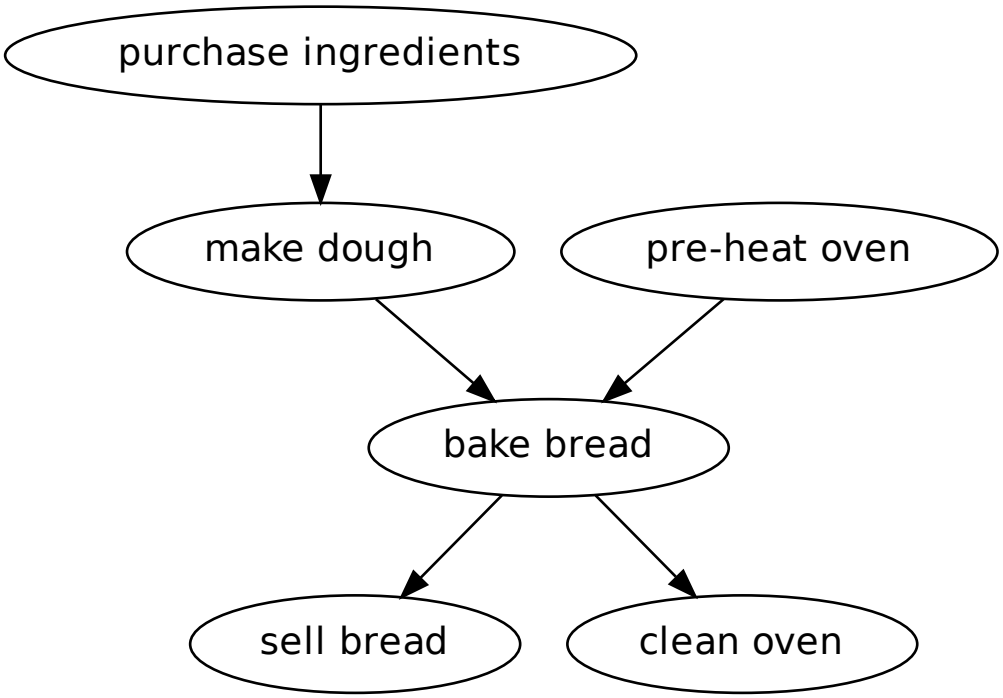
ONE

INTRODUCTION

1.1 What Is A Workflow?

In research, business and other fields we may have processes that we repeat in the course of our work. At its simplest a workflow is a set of steps that must be followed in a particular order to achieve some end goal.

We can represent each "step" in a workflow as a oval and the order with arrows.



1.2 What Is Cylc?

Cylc (pronounced silk) is a workflow engine, a system that automatically executes tasks according to their schedules and dependencies. In a Cylc workflow each step is a computational task, a script to execute. Cylc runs each task as soon as it is appropriate to do so. Cylc was originally developed at NIWA (The National Institute of Water and Atmospheric Research - New Zealand) for running their weather forecasting workflows. Cylc is now developed by an international partnership including members from NIWA and the Met Office (UK). Though initially

developed for meteorological purposes Cylc is a general purpose tool as applicable in business as in scientific research.

CHAPTER

TWO

SCHEDULING

This section looks at how to write workflows in cylc.

2.1 Graphing

In this section we will cover writing basic workflows in cylc.

2.1.1 The `suite.rc` File Format

We refer to a Cylc workflow as a Cylc suite. A Cylc suite is a directory containing a `suite.rc` file. This configuration file is where we define our workflow. The `suite.rc` file uses a nested INI (https://en.wikipedia.org/wiki/INI_file)-based format:

- Comments start with a `#` character.
- Settings are written as `key = value` pairs.
- Settings can be contained within sections.
- Sections are written inside square brackets i.e. `[section-name]`.
- Sections can be nested, by adding an extra square bracket with each level, so a sub-section would be written `[[sub-section]]`, a sub-sub-section `[[[sub-sub-section]]]`, and so on.

```
# Comment
[section]
    key = value
    [[sub-section]]
        another-key = another-value  # Inline comment
        yet-another-key = """
A
Multi-line
String
"""
```

Throughout this tutorial we will refer to settings in the following format:

- `[section]` - refers to the entire section.
- `[section]key` - refers to a setting within the section.
- `[section]key=value` - expresses the value of the setting.
- `[section][sub-section]another-key`. Note we only use one set of square brackets with nested sections.

Tip: It is advisable to indent `suite.rc` files. This indentation, however, is ignored when the file is parsed so settings must appear before sub-sections.

```
[section]
    key = value  # This setting belongs to the section.
    [[sub-section]]
        key = value  # This setting belongs to the sub-section.

    # This setting belongs to the sub-section as indentation is ignored.
    # Always write settings before defining any sub-sections!
    key = value
```

Note

In the `suite.rc` file format duplicate sections are additive, that is to say the following two examples are equivalent:

```
[a]
   c = C
[b]
   d = D
[a]
   e = E
```

```
[a]
   c = C
   e = E
[b]
   d = D
```

Settings, however, are not additive meaning that a duplicate setting will override an earlier value. The following two examples are also equivalent:

```
a = foo
a = bar
```

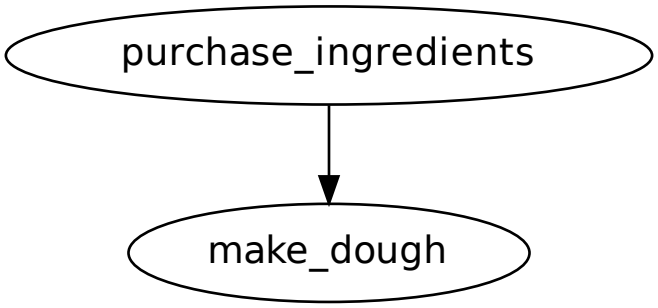
```
a = bar
```

2.1.2 Graph Strings

In Cylc we consider workflows in terms of tasks and dependencies.

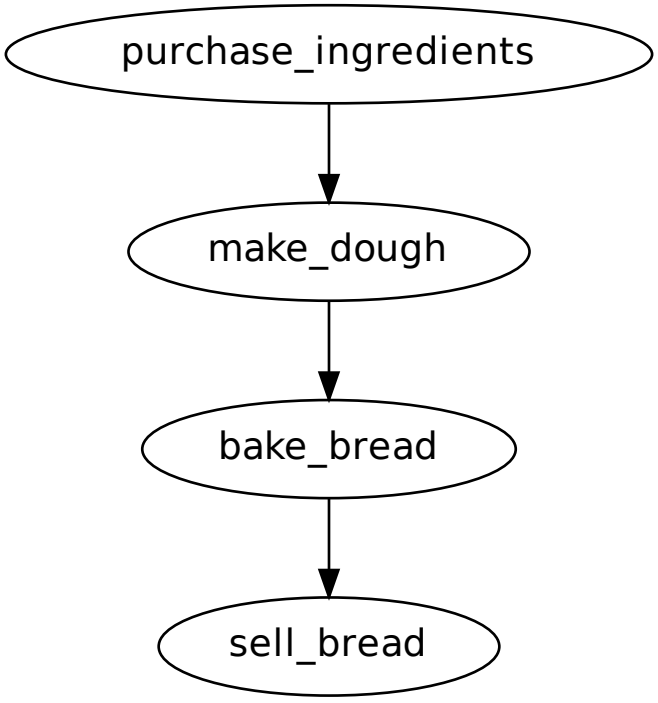
Task are represented as words and dependencies as arrows (=>), so the following text defines two tasks where `make_dough` is dependent on `purchase_ingredients`:

```
purchase_ingredients => make_dough
```



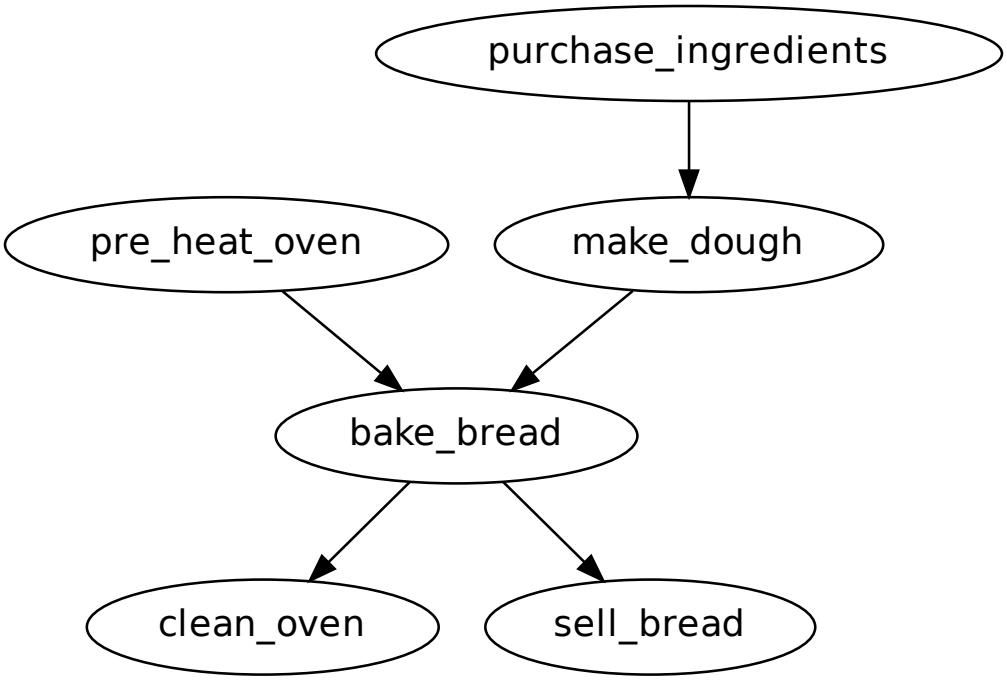
In a Cylc workflow this would mean that `make_dough` would only run when `purchase_ingredients` has succeeded. These dependencies can be chained together:

```
purchase_ingredients => make_dough => bake_bread => sell_bread
```



This line of text is referred to as a graph string. These graph strings can be combined to form more complex workflows:

```
purchase_ingredients => make_dough => bake_bread => sell_bread
pre_heat_oven => bake_bread
bake_bread => clean_oven
```



Graph strings can also contain "and" (`&`) and "or" (`|`) operators, for instance the following lines are equivalent to the ones just above:

```
purchase_ingredients => make_dough
pre_heat_oven & make_dough => bake_bread => sell_bread & clean_oven
```

Collectively these graph strings are referred to as a graph.

Note

The order in which lines appear in the graph section doesn't matter, for instance the following examples are the same as each other:

```
foo => bar
bar => baz
```

```
bar => baz
foo => bar
```

2.1.3 Cylc Graphs

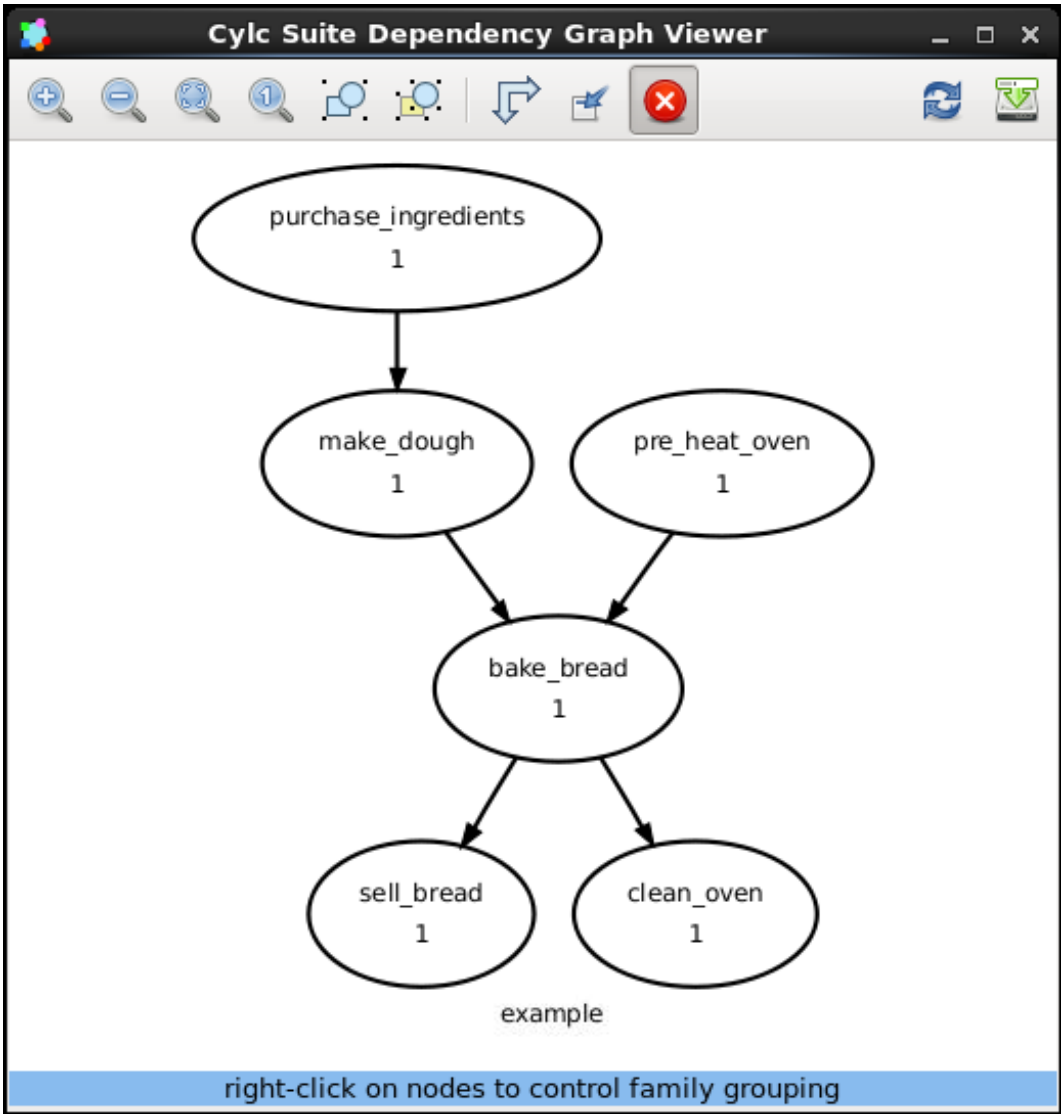
In a Cylc suite the graph is stored under the `[scheduling][dependencies]graph` setting, i.e:

```
[scheduling]
    [[dependencies]]
        graph = """
            purchase_ingredients => make_dough
            pre_heat_oven & make_dough => bake_bread => sell_bread & clean_oven
        """
```

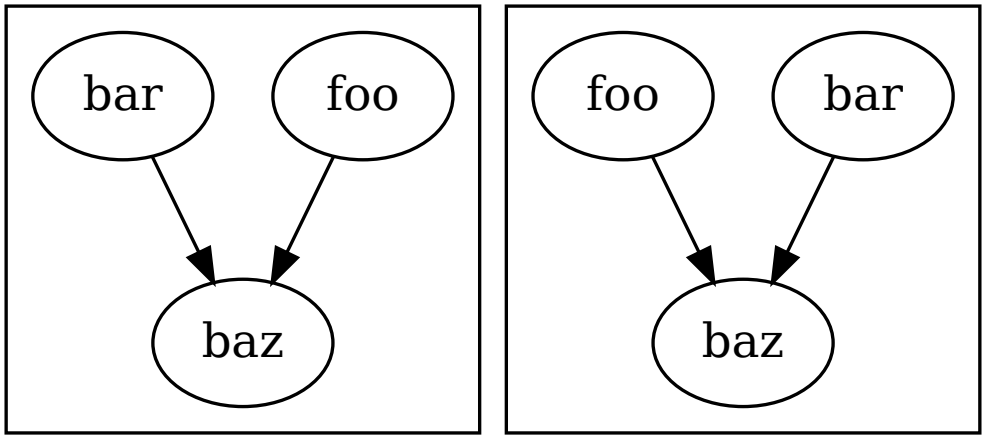
This is a minimal Cylc suite, in which we have defined a graph representing a workflow for Cylc to run. We have not yet provided Cylc with the scripts or binaries to run for each task. This will be covered later in the *runtime tutorial* (page 31).

Cylc provides a GUI for visualising graphs. It is run on the command line using the `cylc graph <path>` command where the path `path` is to the `suite.rc` file you wish to visualise.

When run, `cylc graph` will display a diagram similar to the ones you have seen so far. The number `1` which appears below each task is the cycle point. We will explain what this means in the next section.



Hint: A graph can be drawn in multiple ways, for instance the following two examples are equivalent:



The graph drawn by `cylc graph` may vary slightly from one run to another but the tasks and dependencies will always be the same.

Practical

In this practical we will create a new Cylc suite and write a graph for it to use.

1. **Create a Cylc suite.**

   A Cylc suite is just a directory containing a `suite.rc` file.

   If you don't have one already, create a `cylc-run` directory in your user space i.e:

   ```
   ~/cylc-run
   ```

   Within this directory create a new folder called `graph-introduction`, which is to be our suite directory. Move into it:

   ```
   mkdir ~/cylc-run/graph-introduction
   cd ~/cylc-run/graph-introduction
   ```

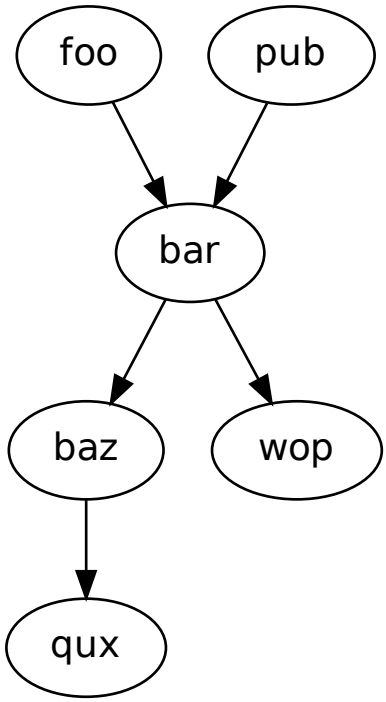
   Inside this directory create a `suite.rc` file and paste in the following text:

   ```
   [scheduling]
       [[dependencies]]
           graph = """
               # Write graph strings here!
           """
   ```

2. **Write a graph.**

   We now have a blank Cylc suite, next we need to define a workflow.

   Edit your `suite.rc` file to add graph strings representing the following graph:



3. **Use `cylc graph` to visualise the workflow.**

   Once you have written some graph strings try using `cylc graph` to display the workflow. Run the following command:

   ```
   cylc graph .
   ```

   Note

   `cylc graph` takes the path to the suite as an argument. As we are inside the suite directory we can run `cylc graph .`.

   If the results don't match the diagram above try going back to the suite.rc file and making changes.

   Tip: In the top right-hand corner of the `cylc graph` window there is a refresh button which will reload the GUI with any changes you have made.

   Solution

   There are multiple correct ways to write this graph. So long as what you see in `cylc graph` matches the above diagram then you have a correct solution.

   Two valid examples:

```
foo & pub => bar => baz & wop
baz => qux
```

```
foo => bar => baz => qux
pub => bar => wop
```

The whole suite should look something like this:

```
[scheduling]
    [[dependencies]]
        graph = """
            foo & pub => bar => baz & wop
            baz => qux
        """
```

2.2 Basic Cycling

In this section we will look at how to write cycling (repeating) workflows.

2.2.1 Repeating Workflows

Often, we will want to repeat the same workflow multiple times. In Cylc this "repetition" is called cycling and each repetition of the workflow is referred to as a cycle.

Each cycle is given a unique label. This is called a cycle point. For now these cycle points will be integers *(they can also be dates as we will see in the next section)*.

To make a workflow repeat we must tell Cylc three things:

The recurrence How often we want the workflow to repeat.

The initial cycle point At what cycle point we want to start the workflow.

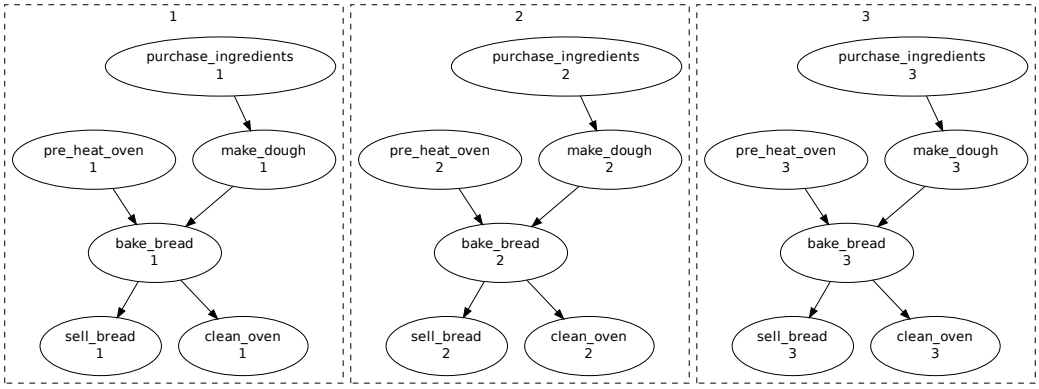
The final cycle point *Optionally* we can also tell Cylc what cycle point we want to stop the workflow.

Let's take the bakery example from the previous section. Bread is produced in batches so the bakery will repeat this workflow for each batch of bread they bake. We can make this workflow repeat with the addition of three lines:

```
 [scheduling]
+    cycling mode = integer
+    initial cycle point = 1
     [[dependencies]]
+        [[[P1]]]
             graph = """
                 purchase_ingredients => make_dough
                 pre_heat_oven & make_dough => bake_bread => sell_bread & clean_
↪oven
             """
```

- The `cycling mode = integer` setting tells Cylc that we want our cycle points to be numbered.
- The `initial cycle point = 1` setting tells Cylc to start counting from 1.
- `P1` is the recurrence. The graph within the `[[[P1]]]` section will be repeated at each cycle point.

The first three cycles would look like this, with the entire workflow repeated at each cycle point:



Note the numbers under each task which represent the cycle point each task is in.

2.2.2 Inter-Cycle Dependencies

We've just seen how to write a workflow that repeats every cycle.

Cylc runs tasks as soon as their dependencies are met so cycles are not necessarily run in order. This could cause problems, for instance we could find ourselves pre-heating the oven in one cycle whist we are still cleaning it in another.

To resolve this we must add dependencies *between* the cycles. We do this by adding lines to the graph. Tasks in the previous cycle can be referred to by suffixing their name with `[-P1]`, for example. So to ensure the `clean_oven` task has been completed before the start of the `pre_heat_oven` task in the next cycle, we would write the following dependency:

```
clean_oven[-P1] => pre_heat_oven
```

This dependency can be added to the suite by adding it to the other graph lines:

```
 [scheduling]
     cycling mode = integer
     initial cycle point = 1
     [[dependencies]]
         [[[P1]]]
             graph = """
                 purchase_ingredients => make_dough
                 pre_heat_oven & make_dough => bake_bread => sell_bread & clean_
↪oven
+                clean_oven[-P1] => pre_heat_oven
             """
```

The resulting suite would look like this:

Adding this dependency "strings together" the cycles, forcing them to run in order. We refer to dependencies between cycles as inter-cycle dependencies.

In the dependency the `[-P1]` suffix tells Cylc that we are referring to a task in the previous cycle. Equally `[-P2]` would refer to a task two cycles ago.

Note that the `purchase_ingredients` task has no arrows pointing at it meaning that it has no dependencies. Consequently the `purchase_ingredients` tasks will all run straight away. This could cause our bakery to run into cash-flow problems as they would be purchasing ingredients well in advance of using them.

To solve this, but still make sure that they never run out of ingredients, the bakery wants to purchase ingredients two batches ahead. This can be achieved by adding the following dependency:

```
 [scheduling]
     cycling mode = integer
     initial cycle point = 1
     [[dependencies]]
         [[[P1]]]
             graph = """
                 purchase_ingredients => make_dough
                 pre_heat_oven & make_dough => bake_bread => sell_bread & clean_
↪oven
                 clean_oven[-P1] => pre_heat_oven
+                sell_bread[-P2] => purchase_ingredients
```

(continues on next page)

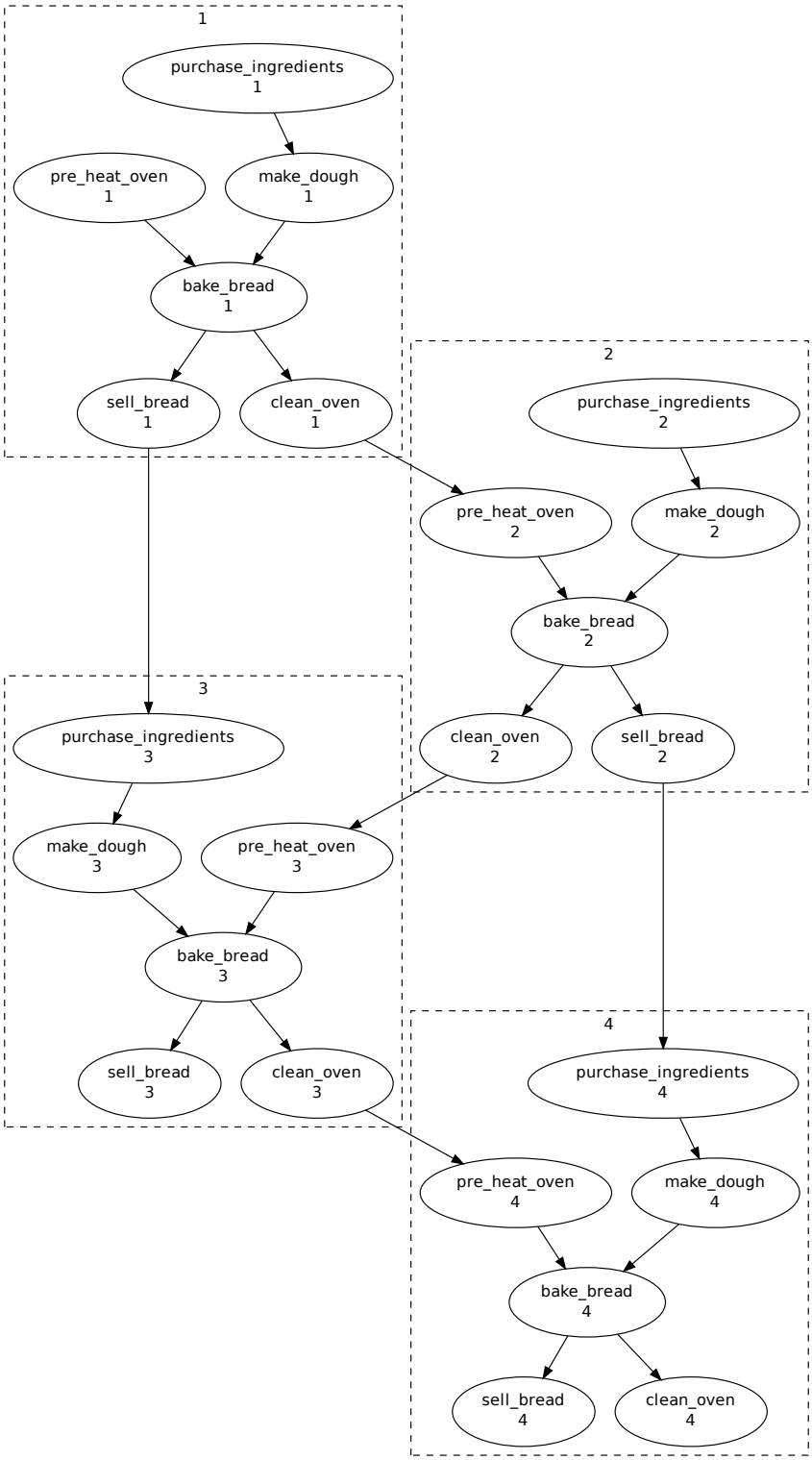
(continued from previous page)

```
        """
```

This dependency means that the `purchase_ingredients` task will run after the `sell_bread` task two cycles before.

Note: The `[-P2]` suffix is used to reference a task two cycles before. For the first two cycles this doesn't make sense as there was no cycle two cycles before, so this dependency will be ignored.

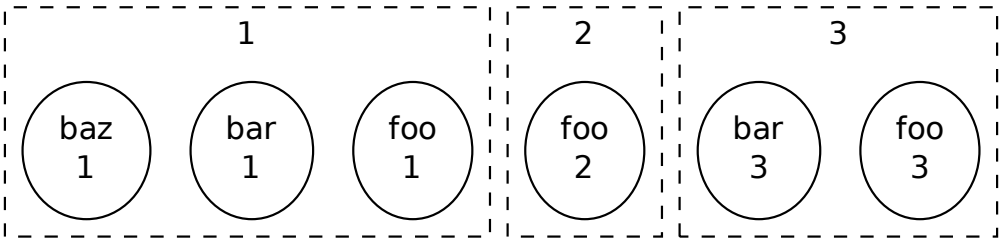
Any inter-cycle dependencies stretching back to before the initial cycle point will be ignored.


1
purchase_ingredients
1
pre_heat_oven
1
make_dough
1
bake_bread
1
sell_bread
1
clean_oven
1
2
purchase_ingredients
2
pre_heat_oven
2
make_dough
2
bake_bread
2
clean_oven
2
sell_bread
2
3
purchase_ingredients
3
make_dough
3
pre_heat_oven
3
bake_bread
3
sell_bread
3
clean_oven
3
4
purchase_ingredients
4
pre_heat_oven
4
make_dough
4
bake_bread
4
sell_bread
4
clean_oven
4

2.2.3 Recurrence Sections

In the previous examples we made the workflow repeat by placing the graph within the `[[[P1]]]` section. Here `P1` is a recurrence meaning repeat every cycle, where `P1` means every cycle, `P2` means every *other* cycle, and so on. To build more complex workflows we can use multiple recurrences:

```
[scheduling]
    cycling mode = integer
    initial cycle point = 1
    [[dependencies]]
        [[[P1]]]  # Repeat every cycle.
            graph = foo
        [[[P2]]]  # Repeat every second cycle.
            graph = bar
        [[[P3]]]  # Repeat every third cycle.
            graph = baz
```



By default recurrences start at the initial cycle point, however it is possible to make them start at an arbitrary cycle point. This is done by writing the cycle point and the recurrence separated by a forward slash (`/`), e.g. `5/P3` means repeat every third cycle starting *from* cycle number 5.

The start point of a recurrence can also be defined as an offset from the initial cycle point, e.g. `+P5/P3` means repeat every third cycle starting 5 cycles *after* the initial cycle point.

Practical

In this practical we will take the suite we wrote in the previous section and turn it into a cycling suite.

If you have not completed the previous practical use the following code for your `suite.rc` file.

```
[scheduling]
    [[dependencies]]
        graph = """
            foo & pub => bar => baz & wop
            baz => qux
        """
```

1. **Create a new suite.**

   Within your `~/cylc-run/` directory create a new (sub-)directory called `integer-cycling` and move into it:

   ```
   mkdir ~/cylc-run/integer-cycling
   cd ~/cylc-run/integer-cycling
   ```

Copy the above code into a `suite.rc` file in that directory.

2. **Make the suite cycle.**

   Add in the following lines.

```
 [scheduling]
+    cycling mode = integer
+    initial cycle point = 1
     [[dependencies]]
+        [[[P1]]]
             graph = """
                 foo & pub => bar => baz & wop
                 baz => qux
             """
```

3. **Visualise the suite.**

   Try visualising the suite using `cylc graph`.

```
cylc graph .
```

Tip: You can get Cylc graph to draw dotted boxes around the cycles by clicking the "Organise by cycle point" button on the toolbar:

Tip: By default `cylc graph` displays the first three cycles of a suite. You can tell `cylc graph` to visualise the cycles between two points by providing them as arguments, for instance the following example would show all cycles between `1` and `5` (inclusive):

```
cylc graph . 1 5 &
```

---

4. **Add another recurrence.**

   Suppose we wanted the `qux` task to run every *other* cycle as opposed to every cycle. We can do this by adding another recurrence.

   Make the following changes to your `suite.rc` file.

```
 [scheduling]
     cycling mode = integer
     initial cycle point = 1
     [[dependencies]]
         [[[P1]]]
             graph = """
                 foo & pub => bar => baz & wop
-                baz => qux
             """
+        [[[P2]]]
+            graph = """
+                baz => qux
+            """
```

   Use `cylc graph` to see the effect this has on the workflow.

5. **Inter-cycle dependencies.**

   Next we need to add some inter-cycle dependencies. We are going to add three inter-cycle dependencies:

(a) Between `wop` from the previous cycle and `pub`.

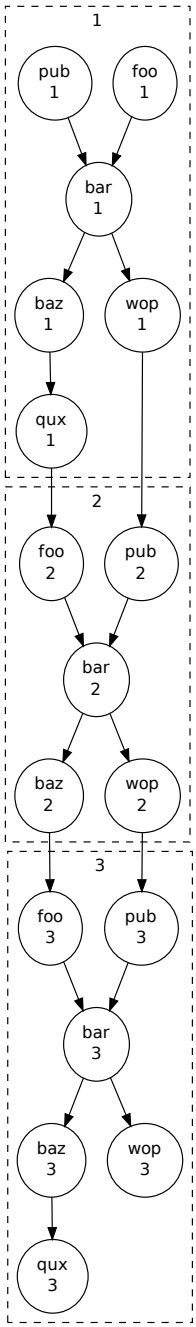
(b) Between `baz` from the previous cycle and `foo` *every odd cycle*.

(c) Between `qux` from the previous cycle and `foo` *every even cycle*.

Have a go at adding inter-cycle dependencies to your `suite.rc` file to make your workflow match the diagram below.

Hint:

- `P2` means every odd cycle.
- `2/P2` means every even cycle.



Solution

```
[scheduling]
    cycling mode = integer
    initial cycle point = 1
    [[dependencies]]
        [[[P1]]]
            graph = """
```

(continues on next page)

(continued from previous page)

```
            foo & pub => bar => baz & wop
            wop[-P1] => pub  # (1)
        """
    [[[P2]]]
        graph = """
            baz => qux
            baz[-P1] => foo  # (2)
        """
    [[[2/P2]]]
        graph = """
            qux[-P1] => foo  # (3)
        """
```

2.3 Date-Time Cycling

In the last section we looked at writing an integer cycling workflow, one where the cycle points are numbered.

Typically workflows are repeated at a regular time interval, say every day or every few hours. To make this easier Cylc has a date-time cycling mode where the cycle points use date and time specifications rather than numbers.

Reminder

Cycle points are labels. Cylc runs tasks as soon as their dependencies are met so cycles do not necessarily run in order.

2.3.1 ISO8601

In Cylc, dates, times and durations are written using the ISO8601 format - an international standard for representing dates and times.

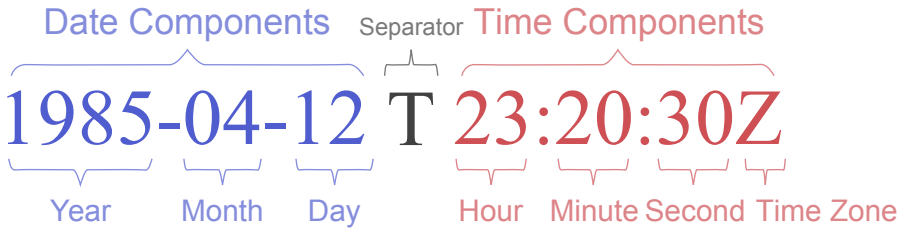
Date-Times

In ISO8601, datetimes are written from the largest unit to the smallest (i.e: year, month, day, hour, minute, second in succession) with the `T` character separating the date and time components. For example, midnight on the 1st of January 2000 is written `20000101T000000`.

For brevity we may omit seconds (and minutes) from the time i.e: `20000101T0000` (`20000101T00`).

For readability we may add hyphen (`-`) characters between the date components and colon (`:`) characters between the time components, i.e: `2000-01-01T00:00`. This is the "extended" format.

Time-zone information can be added onto the end. UTC is written `Z`, UTC+1 is written `+01`, etc. E.G: `2000-01-01T00:00Z`.



Warning: The "basic" (purely numeric except for `T`) and "extended" (written with hyphens and colons) formats cannot be mixed. For example the following date-times are invalid:

```
2000-01-01T0000
20000101T00:00
```

Durations

In ISO8601, durations are prefixed with a `P` and are written with a character following each unit:

- `Y` for year.
- `M` for month.
- `D` for day.
- `W` for week.
- `H` for hour.
- `M` for minute.
- `S` for second.

As with datetimes the components are written in order from largest to smallest and the date and time components are separated by the `T` character. E.G:

- `P1D`: one day.
- `PT1H`: one hour.
- `P1DT1H`: one day and one hour.
- `PT1H30M`: one and a half hours.
- `P1Y1M1DT1H1M1S`: a year and a month and a day and an hour and a minute and a second.

2.3.2 Date-Time Recurrences

In integer cycling, suites' recurrences are written `P1`, `P2`, etc.

In date-time cycling there are two ways to write recurrences:

1. Using ISO8601 durations (e.g. `P1D`, `PT1H`).
2. Using ISO8601 date-times with inferred recurrence.

Inferred Recurrence

A recurrence can be inferred from a date-time by omitting digits from the front. For example, if the year is omitted then the recurrence can be inferred to be annual. E.G:

```
2000-01-01T00   # Datetime - midnight on the 1st of January 2000.

     01-01T00   # Every year on the 1st of January.
        01T00   # Every month on the first of the month.
          T00   # Every day at midnight.
          T-00  # Every hour at zero minutes past (every hour on the hour).
```

Note: To omit hours from a date time we must place a `-` after the `T` character.

Recurrence Formats

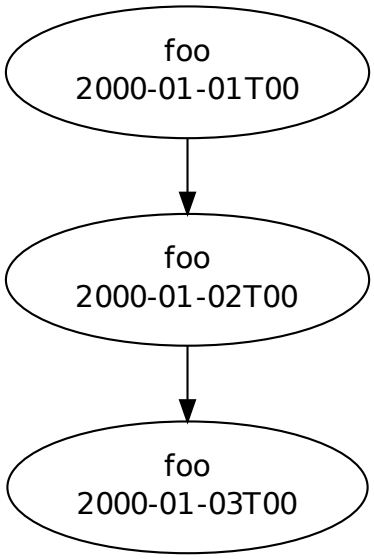
As with integer cycling, recurrences start, by default, at the initial cycle point. We can override this in one of two ways:

1. By defining an arbitrary cycle point (`datetime/recurrence`):
   - `2000/P1Y`: every year starting with the year 2000.
   - `2000-01-01T00/T00`: every day at midnight starting on the 1st of January 2000
   - `2000-01-01T12/T00`: every day at midnight starting on the first midnight after the 1st of January at 12:00 (i.e. `2000-01-02T00`).
2. By defining an offset from the initial cycle point (`offset/recurrence`). This offset is an ISO8601 duration preceded by a plus character:
   - `+P1Y/P1Y`: every year starting one year after the initial cycle point.
   - `+PT1H/T00`: every day starting on the first midnight after the point one hour after the initial cycle point.

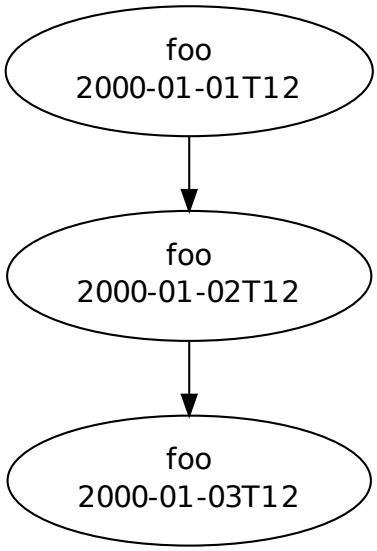
Durations And The Initial Cycle Point

When using durations, beware that a change in the initial cycle point might produce different results for the recurrences.

```
[scheduling]
    initial cycle point = 2000-01-01T00
    [[dependencies]]
        [[[P1D]]]
            graph = foo[-P1D] => foo
```




```
[scheduling]
    initial cycle point = 2000-01-01T12
    [[dependencies]]
        [[[P1D]]]
            graph = foo[-P1D] => foo
```



We could write the recurrence "every midnight" independent from the initial cycle point by:

- Use an *inferred recurrence* (page 22) instead (i.e. `T00`).

- Overriding the recurrence start point (i.e. `T00/P1D`)
- Using the `[scheduling]initial cycle point constraints` setting to constrain the initial cycle point (e.g. to a particular time of day). See the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide) for details.

The Initial & Final Cycle Points

There are two special recurrences for the initial and final cycle points:

- `R1`: repeat once at the initial cycle point.
- `R1/P0Y`: repeat once at the final cycle point.

Inter-Cycle Dependencies

Inter-cycle dependencies are written as ISO8601 durations, e.g:

- `foo[-P1D]`: the task `foo` from the cycle one day before.
- `bar[-PT1H30M]`: the task `bar` from the cycle 1 hour 30 minutes before.

The initial cycle point can be referenced using a caret character `^`, e.g:

- `baz[^]`: the task `baz` from the initial cycle point.

2.3.3 UTC Mode

Due to all of the difficulties caused by time zones, particularly with respect to daylight savings, we typically use UTC (that's the `+00` time zone) in Cylc suites.

When a suite uses UTC all of the cycle points will be written in the `+00` time zone.

To make your suite use UTC set the `[cylc]UTC mode` setting to `True`, i.e:

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

2.3.4 Putting It All Together

Cylc was originally developed for running operational weather forecasting. In this section we will outline a basic (dummy) weather-forecasting suite and explain how to implement it in cylc.

Note: Technically the suite outlined in this section is a nowcasting (https://www.metoffice.gov.uk/learning/science/hours-ahead/nowcasting) suite. We will refer to it as forecasting for simplicity.

A basic weather-forecasting workflow consists of three main steps:

1. Gathering Observations

We gather observations from different weather stations and use them to build a picture of the current weather. Our dummy weather forecast will get wind observations from four weather stations:

- Belmullet
- Camborne
- Heathrow

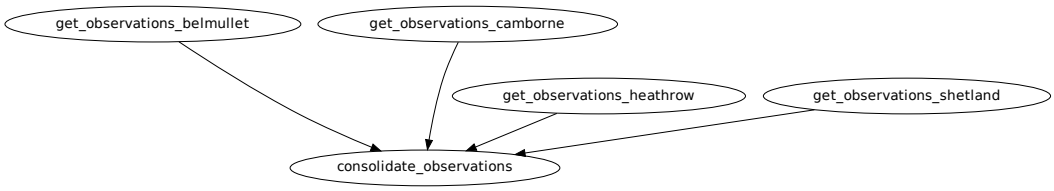
- Shetland

The tasks which retrieve observation data will be called `get_observations_<site>` where `site` is the name of the weather station in question.

Next we need to consolidate these observations so that our forecasting system can work with them. To do this we have a `consolidate_observations` task.

We will fetch wind observations **every three hours starting from the initial cycle point**.

The `consolidate_observations` task must run after the `get_observations<site>` tasks.



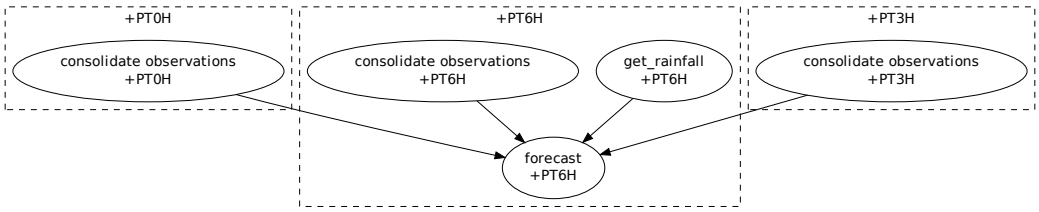
We will also use the UK radar network to get rainfall data with a task called `get_rainfall`.

We will fetch rainfall data **every six hours starting six hours after the initial cycle point**.

2. Running computer models to generate forecast data

We will do this with a task called `forecast` which will run **every six hours starting six hours after the initial cycle point**. The `forecast` task will be dependent on:

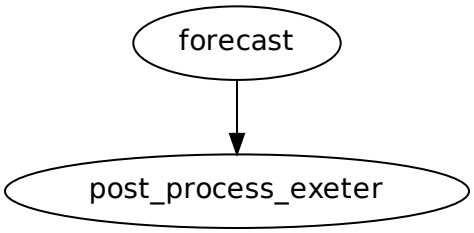
- The `consolidate_observations` task from the previous two cycles as well as from the present cycle.
- The `get_rainfall` task from the present cycle.



3. Processing the data output to produce user-friendly forecasts

This will be done with a task called `post_process_<location>` where `location` is the place we want to generate the forecast for. For the moment we will use Exeter.

The `post_process_exeter` task will run **every six hours starting six hours after the initial cycle point** and will be dependent on the `forecast` task.



Practical

In this practical we will create a dummy forecasting suite using date-time cycling.

1. **Create A New Suite.**

   Within your `~/cylc-run` directory create a new directory called `datetime-cycling` and move into it:

```
mkdir ~/cylc-run/datetime-cycling
cd ~/cylc-run/datetime-cycling
```

   Create a `suite.rc` file and paste the following code into it:

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20000101T00Z
    [[dependencies]]
```

2. **Add The Recurrences.**

   The weather-forecasting suite will require two recurrences. Add sections under the dependencies section for these, based on the information given above.

   Hint: See *Date-Time Recurrences* (page 23).

   Solution

   The two recurrences you need are

   - `PT3H`: repeat every three hours starting from the initial cycle point.
   - `+PT6H/PT6H`: repeat every six hours starting six hours after the initial cycle point.

```
 [cylc]
     UTC mode = True
 [scheduling]
     initial cycle point = 20000101T00Z
     [[dependencies]]
+        [[[PT3H]]]
+        [[[+PT6H/PT6H]]]
```

3. **Write The Graphing.**

   With the help of the graphs and the information above add dependencies to your suite to implement the weather-forecasting workflow.

   You will need to consider the inter-cycle dependencies between tasks.

   Use `cylc graph` to inspect your work.

   Hint

   The dependencies you will need to formulate are as follows:

   - The `consolidate_observations` task is dependent on the `get_observations_<site>` tasks.
   - The `forecast` task is dependent on:
     - the `get_rainfall` task;
     - the `consolidate_observations` tasks from:
       * the same cycle;
       * the cycle 3 hours before (`-PT3H`);
       * the cycle 6 hours before (`-PT6H`).
   - The `post_process_exeter` task is dependent on the `forecast` task.

   To launch `cylc graph` run the command:

   ```
   cylc graph <path/to/suite.rc>
   ```

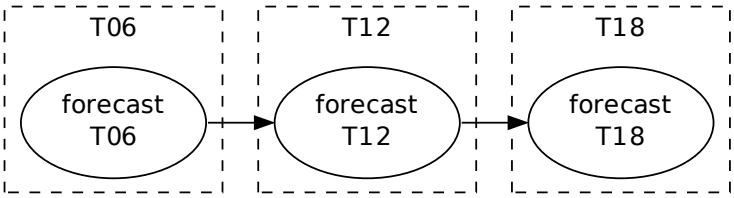
   Solution

   ```
   [cylc]
       UTC mode = True
   [scheduling]
       initial cycle point = 20000101T00Z
       [[dependencies]]
           [[[PT3H]]]
               graph = """
                   get_observations_belmullet => consolidate_observations
                   get_observations_camborne => consolidate_observations
                   get_observations_heathrow => consolidate_observations
                   get_observations_shetland => consolidate_observations
               """
           [[[+PT6H/PT6H]]]
               graph = """
                   consolidate_observations => forecast
                   consolidate_observations[-PT3H] => forecast
                   consolidate_observations[-PT6H] => forecast
                   get_rainfall => forecast => post_process_exeter
               """
   ```

   ---

4. **Inter-Cycle Offsets.**

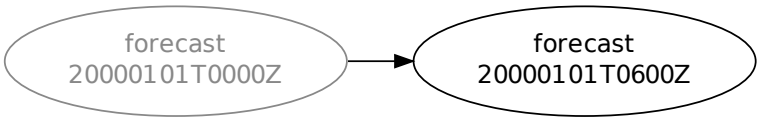
   To ensure the `forecast` tasks for different cycles run in order the `forecast` task will also need to be dependent on the previous run of `forecast`.



We can express this dependency as `forecast[-PT6H] => forecast`.

Try adding this line to your suite then visualising it with `cylc graph`.

You will notice that there is a dependency which looks like this:



Note in particular that the `forecast` task in the 00:00 cycle is grey. The reason for this is that this task does not exist. Remember the forecast task runs every six hours **starting 6 hours after the initial cycle point**, so the dependency is only valid from 12:00 onwards. To fix the problem we must add a new dependency section which repeats every six hours **starting 12 hours after the initial cycle point**.

Make the following changes to your suite and the grey task should disappear:

```
            [[[+PT6H/PT6H]]]
                graph = """
                    ...
-                   forecast[-PT6H] => forecast
                """
+           [[[+PT12H/PT6H]]]
+               graph = """
+                   forecast[-PT6H] => forecast
+               """
```

2.4 Further Scheduling

In this section we will quickly run through some of the more advanced features of Cylc's scheduling logic.

2.4.1 Qualifiers

So far we have written dependencies like `foo => bar`. This is, in fact, shorthand for `foo:succeed => bar`. It means that the task `bar` will run once `foo` has finished successfully. If `foo` were to fail then `bar` would not run. We will talk more about these task states in the Runtime Section.

We refer to the `:succeed` descriptor as a qualifier. There are qualifiers for different task states e.g:

`:start` When the task has started running.

`:fail` When the task finishes if it fails (produces non-zero return code).

`:finish` When the task has completed (either succeeded or failed).

It is also possible to create your own custom qualifiers to handle events within your code (custom outputs).

For more information see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

2.4.2 Clock Triggers

In Cylc, cycle points are just labels. Tasks are triggered when their dependencies are met irrespective of the cycle they are in, but we can force cycles to wait for a particular time before running using clock triggers. This is necessary for certain operational and monitoring systems.

For example in the following suite the cycle `2000-01-01T12Z` will wait until 11:00 on the 1st of January 2000 before running:

```
[scheduling]
    initial cycle point = 2000-01-01T00Z
    [[special tasks]]
        clock-trigger = daily(-PT1H)
    [[dependencies]]
        [[[T12]]]
             graph = daily  # "daily" will run, at the earliest, one hour
                            # before midday.
```

Tip: See the *Clock Triggered Tasks* (page 57) tutorial for more information.

2.4.3 Alternative Calendars

By default Cylc uses the Gregorian calendar for datetime cycling, but Cylc also supports the 360-day calendar (12 months of 30 days each in a year).

```
[scheduling]
    cycling mode = 360day
```

For more information see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

CHAPTER

THREE

RUNTIME

This section covers:

- Associating tasks with scripts and executables.
- Providing executables with runtime configurations.
- Running Cylc suites.

3.1 Introduction

So far we have been working with the `[scheduling]` section. This is where the workflow is defined in terms of tasks and dependencies.

In order to make the workflow runnable we must associate tasks with scripts or binaries to be executed when the task runs. This means working with the `[runtime]` section which determines what runs, as well as where and how it runs.

3.1.1 The Task Section

The runtime settings for each task are stored in a sub-section of the `[runtime]` section. E.g. for a task called `hello_world` we would write settings inside the following section:

```
[runtime]
    [[hello_world]]
```

3.1.2 The `script` Setting

We tell Cylc *what* to execute when a task is run using the `script` setting.

This setting is interpreted as a bash script. The following example defines a task called `hello_world` which writes `Hello World!` to stdout upon execution.

```
[runtime]
    [[hello_world]]
        script = echo 'Hello World!'
```

Note: If you do not set the `script` for a task then nothing will be run.

We can also call other scripts or executables in this way, e.g:

```
[runtime]
    [[hello_world]]
        script = ~/foo/bar/baz/hello_world
```

It is often a good idea to keep our scripts within the Cylc suite directory tree rather than leaving them somewhere else on the system. If you create a `bin` sub-directory within the suite directory this directory will be added to the path when tasks run, e.g:

Listing 1: bin/hello_world

```
#!/usr/bin/bash
echo 'Hello World!'
```

Listing 2: suite.rc

```
[runtime]
    [[hello_world]]
        script = hello_world
```

3.1.3 Tasks And Jobs

When a task is "Run" it creates a job. The job is a bash file containing the script you have told the task to run along with configuration specifications and a system for trapping errors. It is the job which actually gets executed and not the task itself. This "job file" is called the job script.

During its life a typical task goes through the following states:

Waiting Tasks wait for their dependencies to be satisfied before running. In the meantime they are in the "Waiting" state.

Submitted When a task's dependencies have been met it is ready for submission. During this phase the job script is created. The job is then submitted to the specified batch system. There is more about this in the *next section* (page 39).

Running A task is in the "Running" state as soon as the job is executed.

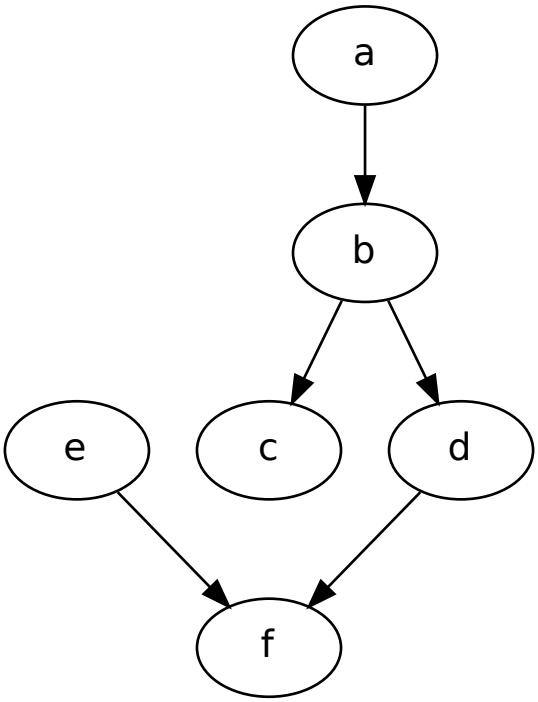
Succeeded If the job submitted by a task has successfully completed (i.e. there is zero return code) then it is said to have succeeded.

These descriptions, and a few more (e.g. failed), are called the task states.

3.1.4 The Cylc GUI

To help you to keep track of a running suite Cylc has a graphical user interface (the Cylc GUI) which can be used for monitoring and interaction.

The Cylc GUI looks quite like `cylc graph` but the tasks are colour-coded to represent their state, as in the following diagram.



This is the “graph view”. The Cylc GUI has two other views called “tree” and “dot”.

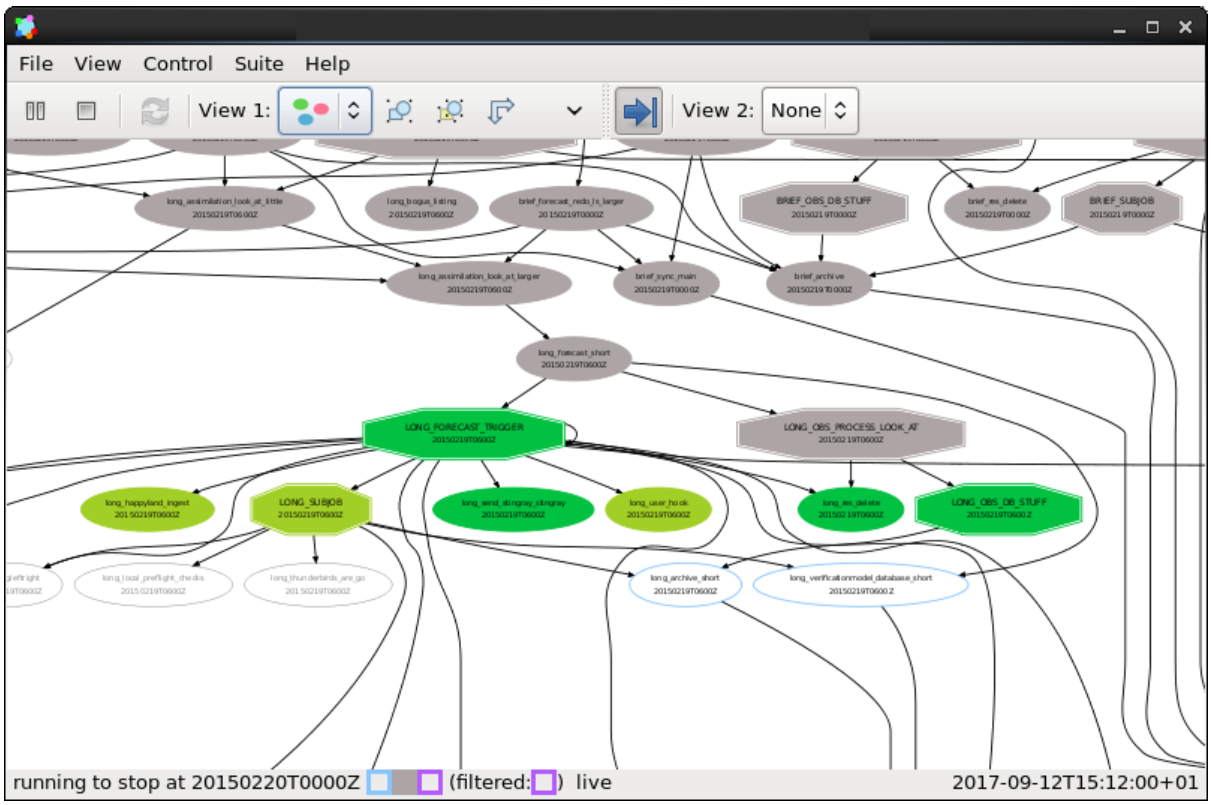


Fig. 1: Screenshot of the Cylc GUI in “Graph View” mode.

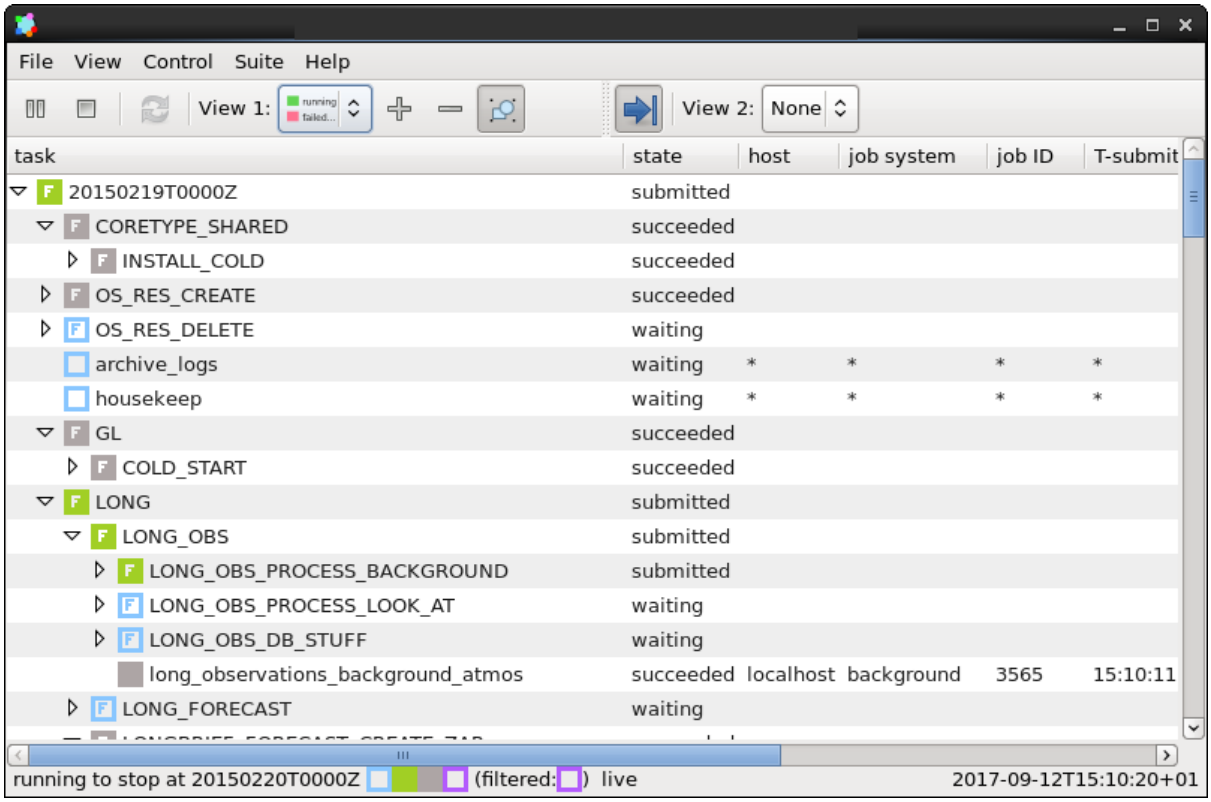


Fig. 2: Screenshot of the Cylc GUI in "Tree View" mode.

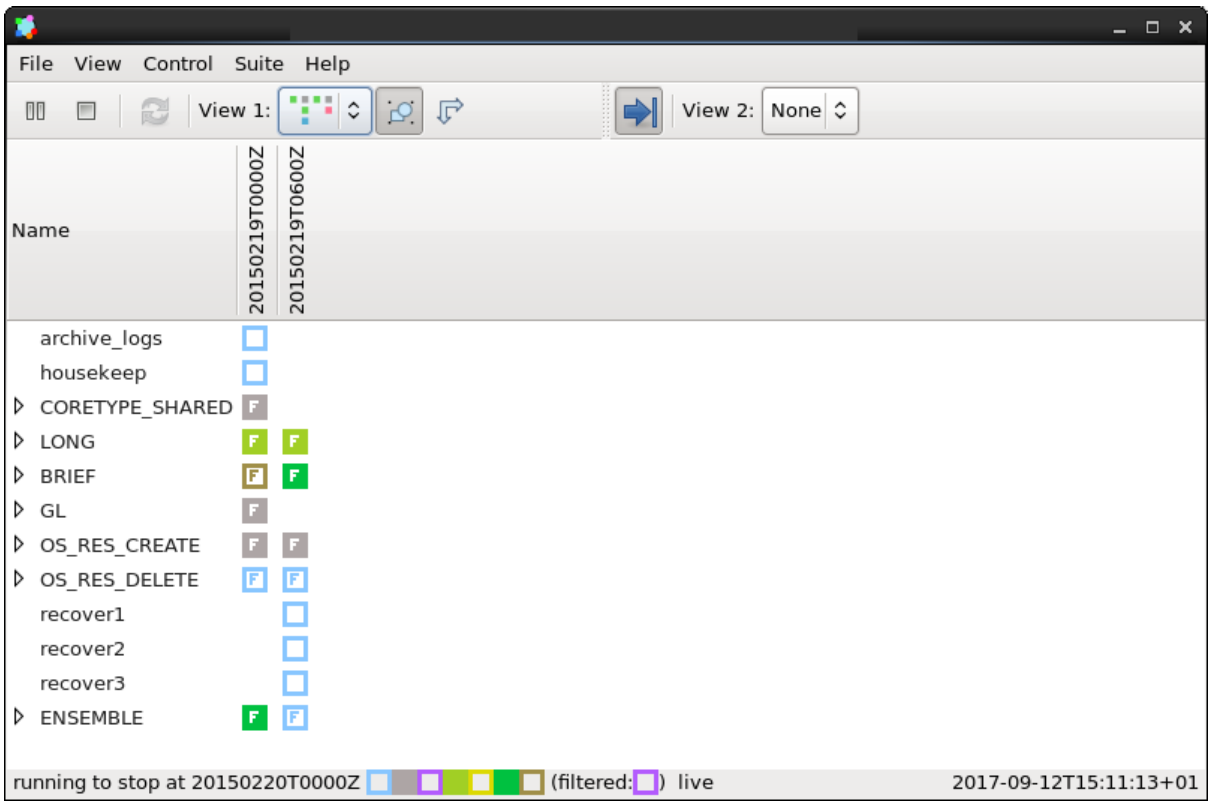


Fig. 3: Screenshot of the Cylc GUI in "Dot View" mode.

3.1.5 Where Do All The Files Go?

The Work Directory

When a task is run Cylc creates a directory for the job to run in. This is called the work directory.

By default the work directory is located in a directory structure under the relevant cycle point and task name:

```
~/cylc-run/<suite-name>/work/<cycle-point>/<task-name>
```

The Job Log Directory

When a task is run Cylc generates a job script which is stored in the job log directory as the file `job`.

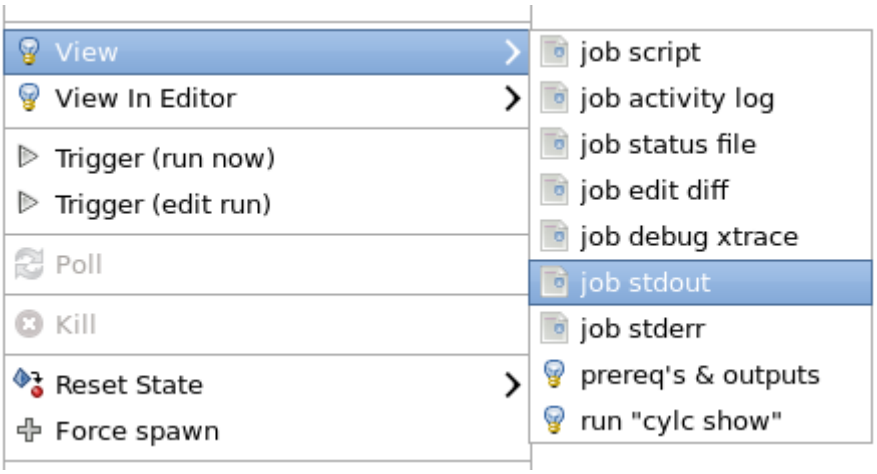
When the job script is executed the stdout and stderr are redirected into the `job.out` and `job.err` files which are also stored in the job log directory.

The job log directory lives in a directory structure under the cycle point, task name and job submission number:

```
~/cylc-run/<suite-name>/log/job/<cycle-point>/<task-name>/<job-submission-num>/
```

The job submission number starts at 1 and increments by 1 each time a task is re-run.

Tip: If a task has run and is still visible in the Cylc GUI you can view its job log files by right-clicking on the task and selecting "View".



3.1.6 Running A Suite

It is a good idea to check a suite for errors before running it. Cylc provides a command which automatically checks for any obvious configuration issues called `cylc validate`, run via:

```
cylc validate <path/to/suite>
```

Here `<path/to/suite>` is the path to the suite's location within the filesystem (so if we create a suite in `~/cylc-run/foo` we would put `~/cylc-run/foo/suite.rc`).

Next we can run the suite using the `cylc run` command.

```
cylc run <name>
```

The `name` is the name of the suite directory (i.e. `<name>` would be `foo` in the above example).

Note: In this tutorial we are writing our suites in the `cylc-run` directory.

It is possible to write them elsewhere on the system. If we do so we must register the suite with Cylc before use.

We do this using the `cylc reg` command which we supply with a name which will be used to refer to the suite in place of the path i.e:

```
cylc reg <name> <path/to/suite>
cylc validate <name>
cylc run <name>
```

The `cylc reg` command will create a directory for the suite in the `cylc-run` directory meaning that we will have separate suite directories and run directories.

3.1.7 Suite Files

Cylc generates files and directories when it runs a suite, namely:

`log/` Directory containing log files, including:

- **`log/db`** The database which Cylc uses to record the state of the suite;
- **`log/job`** The directory where the job log files live;
- **`log/suite`** The directory where the suite log files live. These files are written by Cylc as the suite is run and are useful for debugging purposes in the event of error.

`suite.rc.processed` A copy of the `suite.rc` file made after any Jinja2 (http://jinja.pocoo.org/) has been processed - we will cover this in the *Consolidating Configuration* (page 44) section.

`share/` The share directory is a place where tasks can write files which are intended to be shared within that cycle.

`work/` A directory hierarchy containing task's work directories.

Practical

In this practical we will add some scripts to, and run, the weather forecasting suite from the scheduling tutorial.

1. **Create A New Suite.**

   The following command will copy some files for us to work with into a new suite called `runtime-introduction`:

   ```
   rose tutorial runtime-introduction
   cd ~/cylc-run/runtime-introduction
   ```

   In this directory we have the `suite.rc` file from the *weather forecasting suite* (page 24) with some runtime configuration added to it.

   There is also a script called `get-observations` located in the bin directory.

   Take a look at the `[runtime]` section in the `suite.rc` file.

2. **Run The Suite.**

   First validate the suite by running:

```
cylc validate .
```

Open the Cylc GUI (in the background) by running the following command:

```
cylc gui runtime-introduction &
```

Finally run the suite by executing:

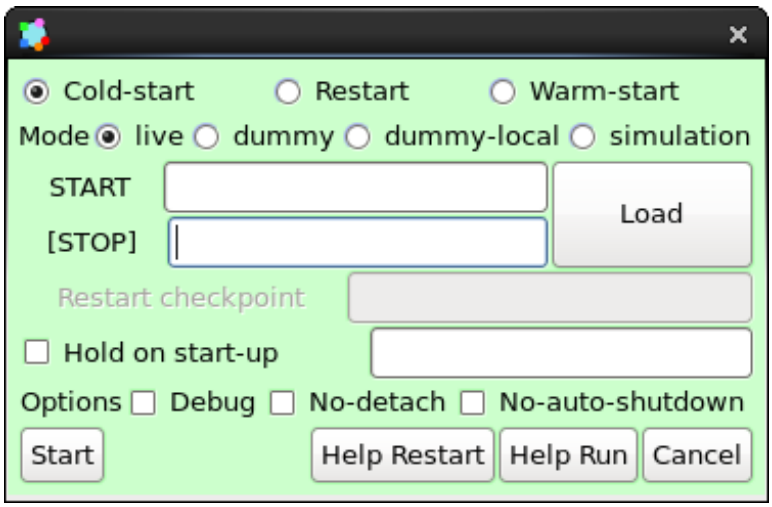
```
cylc run runtime-introduction
```

The tasks will start to run - you should see them going through the "Waiting", "Running" and "Succeeded" states.

When the suite reaches the final cycle point and all tasks have succeeded it will shutdown automatically and the GUI will go blank.

Tip: You can also run a suite from the Cylc GUI by pressing the "play" button.

A box will appear. Ensure that "Cold Start" is selected then press "Start".



---

3. **Inspect A Job Log.**

   Try opening the file `job.out` for one of the `get_observations` jobs in a text editor. The file will be located within the job log directory:

   ```
   log/job/<cycle-point>/get_observations_heathrow/01/job.out
   ```

   You should see something like this:

   ```
   Suite    : runtime-introduction
   Task Job : 20000101T0000Z/get_observations_heathrow/01 (try 1)
   User@Host: username@hostname

   Guessing Weather Conditions
   Writing Out Wind Data
   1970-01-01T00:00:00Z NORMAL - started
   2038-01-19T03:14:08Z NORMAL - succeeded
   ```

   - The first three lines are information which Cylc has written to the file to provide information about the job.

- The last two lines were also written by cylc. They provide timestamps marking the stages in the job's life.
- The lines in the middle are the stdout of the job itself.

4. **Inspect A Work Directory.**

   The `get_rainfall` task should create a file called `rainfall` in its work directory. Try opening this file, recalling that the format of the relevant path from within the work directory will be:

```
work/<cycle-point>/get_rainfall/rainfall
```

   Hint: The `get_rainfall` task only runs every third cycle.

5. **Extension: Explore The Cylc GUI**

   - Try re-running the suite.
   - Try changing the current view(s).

     Tip: You can do this from the "View" menu or from the toolbar:

   - Try pressing the "Pause" button which is found in the top left-hand corner of the GUI.
   - Try right-clicking on a task. From the right-click menu you could try:
     - "Trigger (run now)"
     - "Reset State"

3.2 Runtime Configuration

In the last section we associated tasks with scripts and ran a simple suite. In this section we will look at how we can configure these tasks.

3.2.1 Environment Variables

We can specify environment variables in a task's `[environment]` section. These environment variables are then provided to jobs when they run.

```
[runtime]
    [[countdown]]
        script = seq $START_NUMBER
        [[[environment]]]
            START_NUMBER = 5
```

Each job is also provided with some standard environment variables e.g:

`CYLC_SUITE_RUN_DIR` The path to the suite's run directory *(e.g. ~/cylc-run/suite)*.

`CYLC_TASK_WORK_DIR` The path to the associated task's work directory *(e.g. run-directory/work/cycle/task)*.

`CYLC_TASK_CYCLE_POINT` The cycle point for the associated task *(e.g. 20171009T0950)*.

There are many more environment variables - see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide) for more information.

3.2.2 Job Submission

By default Cylc runs jobs on the machine where the suite is running. We can tell Cylc to run jobs on other machines by setting the `[remote]host` setting to the name of the host, e.g. to run a task on the host `computehost` you might write:

```
[runtime]
    [[hello_computehost]]
        script = echo "Hello Compute Host"
        [[[remote]]]
            host = computehost
```

Cylc also executes jobs as background processes (https://en.wikipedia.org/wiki/Background_process) by default. When we are running jobs on other compute hosts we will often want to use a batch system (job scheduler (https://en.wikipedia.org/wiki/Job_scheduler)) to submit our job. Cylc supports the following batch systems:

- at
- loadleveler
- lsf
- pbs
- sge
- slurm
- moab

Batch systems typically require directives in some form. Directives inform the batch system of the requirements of a job, for example how much memory a given job requires or how many CPUs the job will run on. For example:

```
[runtime]
    [[big_task]]
        script = big-executable

        # Submit to the host "big-computer".
        [[[remote]]]
            host = big-computer

        # Submit the job using the "slurm" batch system.
        [[[job]]]
            batch system = slurm

        # Inform "slurm" that this job requires 500MB of RAM and 4 CPUs.
        [[[directives]]]
            --mem = 500
            --ntasks = 4
```

3.2.3 Timeouts

We can specify a time limit after which a job will be terminated using the `[job]execution time limit` setting. The value of the setting is an ISO8601 duration. Cylc automatically inserts this into a job's directives as appropriate.

```
[runtime]
    [[some_task]]
        script = some-executable
        [[[job]]]
            execution time limit = PT15M  # 15 minutes.
```

3.2.4 Retries

Sometimes jobs fail. This can be caused by two factors:

- Something going wrong with the job's execution e.g:
  - A bug;
  - A system error;
  - The job hitting the `execution time limit`.
- Something going wrong with the job submission e.g:
  - A network problem;
  - The job host becoming unavailable or overloaded;
  - An issue with the directives.

In the event of failure Cylc can automatically re-submit (retry) jobs. We configure retries using the `[job]execution retry delays` and `[job]submission retry delays` settings. These settings are both set to an ISO8601 duration, e.g. setting `execution retry delays` to `PT10M` would cause the job to retry every 10 minutes in the event of execution failure.

We can limit the number of retries by writing a multiple in front of the duration, e.g:

```
[runtime]
    [[some-task]]
        script = some-script
        [[[job]]]
            # In the event of execution failure, retry a maximum
            # of three times every 15 minutes.
            execution retry delays = 3*PT15M
            # In the event of submission failure, retry a maximum
            # of two times every ten minutes and then every 30
            # minutes thereafter.
            submission retry delays = 2*PT10M, PT30M
```

3.2.5 Start, Stop, Restart

We have seen how to start and stop Cylc suites with `cylc run` and `cylc stop` respectively. The `cylc stop` command causes Cylc to wait for all running jobs to finish before it stops the suite. There are two options which change this behaviour:

`cylc stop --kill` When the `--kill` option is used Cylc will kill all running jobs before stopping. *Cylc can kill jobs on remote hosts and uses the appropriate command when a* batch system *is used.*

`cylc stop --now --now` When the `--now` option is used twice Cylc stops straight away, leaving any jobs running.

Once a suite has stopped it is possible to restart it using the `cylc restart` command. When the suite restarts it picks up where it left off and carries on as normal.

```
# Run the suite "name".
cylc run <name>
# Stop the suite "name", killing any running tasks.
cylc stop <name> --kill
# Restart the suite "name", picking up where it left off.
cylc restart <name>
```

Practical

In this practical we will add runtime configuration to the weather-forecasting suite from the scheduling tutorial.

1. **Create A New Suite.**

   Create a new suite by running the command:

   ```
   rose tutorial runtime-tutorial
   cd ~/cylc-run/runtime-tutorial
   ```

   You will now have a copy of the weather-forecasting suite along with some executables and python modules.

1. **Set The Initial And Final Cycle Points.**

   First we will set the initial and final cycle points (see the *datetime tutorial* (page 21) for help with writing ISO8601 datetimes):

   - The final cycle point should be set to the time one hour ago from the present time (with minutes and seconds ignored), *e.g. if the current time is 9:45 UTC then the final cycle point should be at 8:00 UTC.*
   - The initial cycle point should be the final cycle point minus six hours.

   Reminder

   Remember that we are working in UTC mode (the `+00` time zone). Datetimes should end with a `Z` character to reflect this.

   Solution

   You can check your answers by running the following commands (hyphens and colons optional but can't be mixed):

   For the initial cycle point: `rose date --utc --offset -PT7H --format CCYY-MM-DDThh:00Z`

   For the final cycle point: `rose date --utc --offset -PT1H --format CCYY-MM-DDThh:00Z`

   Run `cylc validate` to check for any errors:

   ```
   cylc validate .
   ```

2. **Add Runtime Configuration For The `get_observations` Tasks.**

   In the `bin` directory is a script called `get-observations`. This script gets weather data from the MetOffice DataPoint (https://www.metoffice.gov.uk/datapoint) service. It requires two environment variables:

   `SITE_ID`: A four digit numerical code which is used to identify a weather station, e.g. `3772` is Heathrow Airport.

   `API_KEY`: An authentication key required for access to the service.

   Generate a Datapoint API key:

   ```
   rose tutorial api-key
   ```

   Add the following lines to the bottom of the `suite.rc` file replacing `xxx...` with your API key:

   ```
   [runtime]
       [[get_observations_heathrow]]
           script = get-observations
   ```

(continues on next page)

(continued from previous page)

```
        [[[environment]]]
            SITE_ID = 3772
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

Add three more `get_observations` tasks for each of the remaining weather stations.

You will need the codes for the other three weather stations, which are:

- Camborne - `3808`
- Shetland - `3005`
- Belmullet - `3976`

Solution

```
[runtime]
    [[get_observations_heathrow]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3772
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    [[get_observations_camborne]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3808
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    [[get_observations_shetland]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3005
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    [[get_observations_belmullet]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3976
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

Check the `suite.rc` file is valid by running the command:

```
cylc validate .
```

3. **Test The `get_observations` Tasks.**

   Next we will test the `get_observations` tasks.

   Open the Cylc GUI by running the following command:

   ```
   cylc gui runtime-tutorial &
   ```

   Run the suite either by pressing the play button in the Cylc GUI or by running the command:

   ```
   cylc run runtime-tutorial
   ```

   If all goes well the suite will startup and the tasks will run and succeed. Note that the tasks which do not have a `[runtime]` section will still run though they will not do anything as they do not call any scripts.

   Once the suite has reached the final cycle point and all tasks have succeeded the suite will automatically shutdown.

   The `get-observations` script produces a file called `wind.csv` which specifies the wind speed and direction. This file is written in the task's work directory.

Try and open one of the `wind.csv` files. Note that the path to the work directory is:

```
work/<cycle-point>/<task-name>
```

You should find a file containing four numbers:

- The longitude of the weather station;
- The latitude of the weather station;
- The wind direction (*the direction the wind is blowing towards*) in degrees;
- The wind speed in miles per hour.

Hint

If you run `ls work` you should see a list of cycles. Pick one of them and open the file:

```
work/<cycle-point>/get_observations_heathrow/wind.csv
```

---

4. **Add runtime configuration for the other tasks.**

   The runtime configuration for the remaining tasks has been written out for you in the `runtime` file which you will find in the suite directory. Copy the code in the `runtime` file to the bottom of the `suite.rc` file.

   Check the `suite.rc` file is valid by running the command:

   ```
   cylc validate .
   ```

5. **Run The Suite.**

   Open the Cylc GUI (if not already open) and run the suite.

   Hint

   ```
   cylc gui runtime-tutorial &
   ```

   Run the suite either by:

   - Pressing the play button in the Cylc GUI. Then, ensuring that "Cold Start" is selected within the dialogue window, pressing the "Start" button.
   - Running the command `cylc run runtime-tutorial`.

   ---

6. **View The Forecast Summary.**

   The `post_process_exeter` task will produce a one-line summary of the weather in Exeter, as forecast two hours ahead of time. This summary can be found in the `summary.txt` file in the work directory.

   Try opening the summary file - it will be in the last cycle. The path to the work directory is:

   ```
   work/<cycle-point>/<task-name>
   ```

   Hint

   - `cycle-point` - this will be the last cycle of the suite, i.e. the final cycle point.
   - `task-name` - set this to "post_process_exeter".

   ---

7. **View The Rainfall Data.**

   The `forecast` task will produce a html page where the rainfall data is rendered on a map. This html file is called `job-map.html` and is saved alongside the job log.

   Try opening this file in a web browser, e.g via:

   ```
   firefox <filename> &
   ```

   The path to the job log directory is:

   ```
   log/job/<cycle-point>/<task-name>/<submission-number>
   ```

   Hint

   - `cycle-point` - this will be the last cycle of the suite, i.e. the final cycle point.
   - `task-name` - set this to "forecast".
   - `submission-number` - set this to "01".

3.3 Consolidating Configuration

In the last section we wrote out the following code in the `suite.rc` file:

```
[runtime]
    [[get_observations_heathrow]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3772
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    [[get_observations_camborne]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3808
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    [[get_observations_shetland]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3005
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    [[get_observations_belmullet]]
        script = get-observations
        [[[environment]]]
            SITE_ID = 3976
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

In this code the `script` item and the `API_KEY` environment variable have been repeated for each task. This is bad practice as it makes the configuration lengthy and making changes can become difficult.

Likewise the graphing relating to the `get_observations` tasks is highly repetitive:

```
[scheduling]
    [[dependencies]]
        [[[T00/PT3H]]]
            graph = """
                get_observations_belmullet => consolidate_observations
                get_observations_camborne => consolidate_observations
                get_observations_heathrow => consolidate_observations
```

(continues on next page)

(continued from previous page)

```
                get_observations_shetland => consolidate_observations
            """
```

Cylc offers three ways of consolidating configurations to help improve the structure of a suite and avoid duplication.

3.3.1 Families

Families provide a way of grouping tasks together so they can be treated as one.

Runtime

Families are groups of tasks which share a common configuration. In the present example the common configuration is:

```
script = get-observations
[[[environment]]]
    API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

We define a family as a new task consisting of the common configuration. By convention families are named in upper case:

```
[[GET_OBSERVATIONS]]
    script = get-observations
    [[[environment]]]
        API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

We "add" tasks to a family using the `inherit` setting:

```
[[get_observations_heathrow]]
    inherit = GET_OBSERVATIONS
    [[[environment]]]
        SITE_ID = 3772
```

When we add a task to a family in this way it inherits the configuration from the family, i.e. the above example is equivalent to:

```
[[get_observations_heathrow]]
    script = get-observations
    [[[environment]]]
        API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
        SITE_ID = 3772
```

It is possible to override inherited configuration within the task. For example if we wanted the `get_observations_heathrow` task to use a different API key we could write:

```
[[get_observations_heathrow]]
    inherit = GET_OBSERVATIONS
    [[[environment]]]
        API_KEY = special-api-key
        SITE_ID = 3772
```

Using families the `get_observations` tasks could be written like so:

```
[runtime]
    [[GET_OBSERVATIONS]]
        script = get-observations
        [[[environment]]]
```

(continues on next page)

(continued from previous page)

```
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

    [[get_observations_heathrow]]
        inherit = GET_OBSERVATIONS
        [[[environment]]]
            SITE_ID = 3772
    [[get_observations_camborne]]
        inherit = GET_OBSERVATIONS
        [[[environment]]]
            SITE_ID = 3808
    [[get_observations_shetland]]
        inherit = GET_OBSERVATIONS
        [[[environment]]]
            SITE_ID = 3005
    [[get_observations_belmullet]]
        inherit = GET_OBSERVATIONS
        [[[environment]]]
            SITE_ID = 3976
```

Graphing

Families can be used in the suite's graph, e.g:

```
GET_OBSERVATIONS:succeed-all => consolidate_observations
```

The `:succeed-all` is a special qualifier which in this example means that the `consolidate_observations` task will run once *all* of the members of the `GET_OBSERVATIONS` family have succeeded. This is equivalent to:

```
get_observations_heathrow => consolidate_observations
get_observations_camborne => consolidate_observations
get_observations_shetland => consolidate_observations
get_observations_belmullet => consolidate_observations
```

The `GET_OBSERVATIONS:succeed-all` part is referred to as a family trigger. Family triggers use special qualifiers which are non-optional. The most commonly used ones are:

`succeed-all` Run if all of the members of the family have succeeded.

`succeed-any` Run as soon as any one family member has succeeded.

`finish-all` Run as soon as all of the family members have completed (i.e. have each either succeeded or failed).

For more information on family triggers see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

The `root` Family

There is a special family called *root* (in lowercase) which is used only in the runtime to provide configuration which will be inherited by all tasks.

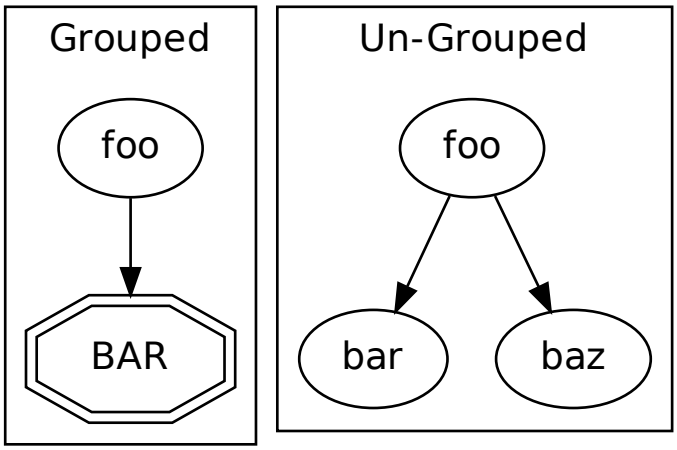
In the following example the task `bar` will inherit the environment variable `FOO` from the `[root]` section:

```
[runtime]
    [[root]]
        [[[environment]]]
            FOO = foo
    [[bar]]
        script = echo $FOO
```

Families and `cylc graph`

By default, `cylc graph` groups together all members of a family in the graph. To un-group a family right click on it and select *UnGroup*.

For instance if the tasks `bar` and `baz` both inherited from `BAR` `cylc graph` would produce:



Practical

In this practical we will consolidate the configuration of the weather-forecasting suite from the previous section.

1. **Create A New Suite.**

   To make a new copy of the forecasting suite run the following commands:

   ```
   rose tutorial consolidation-tutorial
   cd ~/cylc-run/consolidation-tutorial
   ```

   Set the intial and final cycle points as you did in the *previous tutorial* (page 41).

2. **Move Site-Wide Settings Into The `root` Family.**

   The following three environment variables are used by multiple tasks:

   ```
   PYTHONPATH="$CYLC_SUITE_DEF_PATH/lib/python:$PYTHONPATH"
   RESOLUTION = 0.2
   DOMAIN = -12,48,5,61  # Do not change!
   ```

   Rather than manually adding them to each task individually we could put them in the `root` family, making them accessible to all tasks.

   Add a `root` section containing these three environment variables. Remove the variables from any other task's `environment` sections:

   ```
    [runtime]
   +    [[root]]
   ```

(continues on next page)

(continued from previous page)

```
+        [[[environment]]]
+            # Add the `python` directory to the PYTHONPATH.
+            PYTHONPATH="$CYLC_SUITE_DEF_PATH/lib/python:$PYTHONPATH"
+            # The dimensions of each grid cell in degrees.
+            RESOLUTION = 0.2
+            # The area to generate forecasts for (lng1, lat1, lng2, lat2).
+            DOMAIN = -12,48,5,61  # Do not change!
```

```
 [[consolidate_observations]]
     script = consolidate-observations
-    [[[environment]]]
-        # Add the `python` directory to the PYTHONPATH.
-        PYTHONPATH="$CYLC_SUITE_RUN_DIR/lib/python:$PYTHONPATH"
-        # The dimensions of each grid cell in degrees.
-        RESOLUTION = 0.2
-        # The area to generate forecasts for (lng1, lat1, lng2, lat2).
-        DOMAIN = -12,48,5,61  # Do not change!

 [[get_rainfall]]
     script = get-rainfall
     [[[environment]]]
         # The key required to get weather data from the DataPoint service.
         # To use archived data comment this line out.
         API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
-        # Add the `python` directory to the PYTHONPATH.
-        PYTHONPATH="$CYLC_SUITE_RUN_DIR/lib/python:$PYTHONPATH"
-        # The dimensions of each grid cell in degrees.
-        RESOLUTION = 0.2
-        # The area to generate forecasts for (lng1, lat1, lng2, lat2).
-        DOMAIN = -12,48,5,61  # Do not change!

 [[forecast]]
     script = forecast 60 5  # Generate 5 forecasts at 60 minute intervals.
     [[[environment]]]
-        # Add the `python` directory to the PYTHONPATH.
-        PYTHONPATH="$CYLC_SUITE_RUN_DIR/lib/python:$PYTHONPATH"
-        # The dimensions of each grid cell in degrees.
-        RESOLUTION = 0.2
-        # The area to generate forecasts for (lng1, lat1, lng2, lat2)
-        DOMAIN = -12,48,5,61  # Do not change!
         # The path to the files containing wind data (the {variables} will
         # get substituted in the forecast script).
         WIND_FILE_TEMPLATE = $CYLC_SUITE_WORK_DIR/{cycle}/consolidate_
↪observations/wind_{xy}.csv
         # List of cycle points to process wind data from.
         WIND_CYCLES = 0, -3, -6

         # The path to the rainfall file.
         RAINFALL_FILE = $CYLC_SUITE_WORK_DIR/$CYLC_TASK_CYCLE_POINT/get_
↪rainfall/rainfall.csv
         # Create the html map file in the task's log directory.
         MAP_FILE = "${CYLC_TASK_LOG_ROOT}-map.html"
         # The path to the template file used to generate the html map.
         MAP_TEMPLATE = "$CYLC_SUITE_RUN_DIR/lib/template/map.html"

 [[post_process_exeter]]
     # Generate a forecast for Exeter 60 minutes into the future.
     script = post-process exeter 60
-    [[[environment]]]
-        # Add the `python` directory to the PYTHONPATH.
-        PYTHONPATH="$CYLC_SUITE_RUN_DIR/lib/python:$PYTHONPATH"
```

(continues on next page)

(continued from previous page)

```
-         # The dimensions of each grid cell in degrees.
-         RESOLUTION = 0.2
-         # The area to generate forecasts for (lng1, lat1, lng2, lat2).
-         DOMAIN = -12,48,5,61  # Do not change!
```

To ensure that the environment variables are being inherited correctly by the tasks, inspect the `[runtime]` section using `cylc get-config` by running the following command:

```
cylc get-config . --sparse -i "[runtime]"
```

You should see the environment variables from the `[root]` section in the `[environment]` section for all tasks.

Tip: You may find it easier to open the output of this command in a text editor, e.g:

```
cylc get-config . --sparse -i "[runtime]" | gvim -
```

3.3.2 Jinja2

Jinja2 (page 49) is a templating language often used in web design with some similarities to python. It can be used to make a suite definition more dynamic.

The Jinja2 Language

In Jinja2 statements are wrapped with `{%` characters, i.e:

```
{% ... %}
```

Variables are initiated using the `set` statement, e.g:

```
{% set foo = 3 %}
```

Expressions wrapped with `{{` characters will be replaced with the value of the evaluation of the expression, e.g:

```
There are {{ foo }} methods for consolidating the suite.rc file
```

Would result in:

```
There are 3 methods for consolidating the suite.rc file
```

Loops are written with `for` statements, e.g:

```
{% for x in range(foo) %}
   {{ x }}
{% endfor %}
```

Would result in:

```
0
1
2
```

To enable Jinja2 in the `suite.rc` file, add the following shebang (https://en.wikipedia.org/wiki/Shebang_(Unix)) to the top of the file:

```
#!Jinja2
```

For more information see the Jinja2 Tutorial (http://jinja.pocoo.org/docs).

Example

To consolidate the configuration for the `get_observations` tasks we could define a dictionary of station and ID pairs:

```
{% set stations = {'belmullet': 3976,
                   'camborne': 3808,
                   'heathrow': 3772,
                   'shetland': 3005} %}
```

We could then loop over the stations like so:

```
{% for station in stations %}
    {{ station }}
{% endfor %}
```

After processing, this would result in:

```
belmullet
camborne
heathrow
shetland
```

We could also loop over both the stations and corresponding IDs like so:

```
{% for station, id in stations.items() %}
    {{ station }} - {{ id }}
{% endfor %}
```

This would result in:

```
belmullet - 3976
camborne - 3808
heathrow - 3772
shetland - 3005
```

Putting this all together, the `get_observations` configuration could be written as follows:

```
#!Jinja2

{% set stations = {'belmullet': 3976,
                   'camborne': 3808,
                   'heathrow': 3772,
                   'shetland': 3005} %}

[scheduling]
    [[dependencies]]
        [[[T00/PT3H]]]
            graph = """
{% for station in stations %}
                get_observations_{{station}} => consolidate_observations
{% endfor %}
            """
```

```
[runtime]
{% for station, id in stations.items() %}
```

(continues on next page)

(continued from previous page)

```
    [[get_observations_{{station}}]]
        script = get-observations
        [[[environment]]]
            SITE_ID = {{ id }}
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

{% endfor %}
```

Practical

This practical continues on from the families practical.

3. **Use Jinja2 To Avoid Duplication.**

   The `API_KEY` environment variable is used by both the `get_observations` and `get_rainfall` tasks. Rather than writing it out multiple times we will use Jinja2 to centralise this configuration.

   At the top of the `suite.rc` file add the Jinja2 shebang line. Then copy the value of the `API_KEY` environment variable and use it to define an `API_KEY` Jinja2 variable:

   ```
   #!Jinja2

   {% set API_KEY = 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' %}
   ```

   Next replace the key, where it appears in the suite, with `{{ API_KEY }}`:

   ```
    [runtime]
        [[get_observations_heathrow]]
            script = get-observations
            [[[environment]]]
                SITE_ID = 3772
   -            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
   +            API_KEY = {{ API_KEY }}
        [[get_observations_camborne]]
            script = get-observations
            [[[environment]]]
                SITE_ID = 3808
   -            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
   +            API_KEY = {{ API_KEY }}
        [[get_observations_shetland]]
            script = get-observations
            [[[environment]]]
               SITE_ID = 3005
   -            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
   +            API_KEY = {{ API_KEY }}
        [[get_observations_belmullet]]
            script = get-observations
            [[[environment]]]
                SITE_ID = 3976
   -            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
   +            API_KEY = {{ API_KEY }}
       [[get_rainfall]]
           script = get-rainfall
           [[[environment]]]
               # The key required to get weather data from the DataPoint service.
               # To use archived data comment this line out.
   -            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
   +            API_KEY = {{ API_KEY }}
   ```

Check the result with `cylc get-config`. The Jinja2 will be processed so you should not see any difference after making these changes.

3.3.3 Parameterised Tasks

Parameterised tasks (see parameterisation) provide a way of implicitly looping over tasks without the need for Jinja2.

Cylc Parameters

Parameters are defined in their own section, e.g:

```
[cylc]
    [[parameters]]
        world = Mercury, Venus, Earth
```

They can then be referenced by writing the name of the parameter in angle brackets, e.g:

```
[scheduling]
    [[dependencies]]
        graph = start => hello<world> => end
[runtime]
    [[hello<world>]]
        script = echo 'Hello World!'
```

When the `suite.rc` file is read by Cylc, the parameters will be expanded. For example the code above is equivalent to:

```
[scheduling]
    [[dependencies]]
        graph = """
            start => hello_Mercury => end
            start => hello_Venus => end
            start => hello_Earth => end
        """
[runtime]
    [[hello_Mercury]]
        script = echo 'Hello World!'
    [[hello_Venus]]
        script = echo 'Hello World!'
    [[hello_Earth]]
        script = echo 'Hello World!'
```

We can refer to a specific parameter by writing it after an = sign:

```
[runtime]
    [[hello<world=Earth>]]
        script = echo 'Greetings Earth!'
```

Environment Variables

The name of the parameter is provided to the job as an environment variable called `CYLC_TASK_PARAM_<parameter>` where `<parameter>` is the name of the parameter (in the present case `world`):

```
[runtime]
    [[hello<world>]]
        script = echo "Hello ${CYLC_TASK_PARAM_world}!"
```

Parameter Types

Parameters can be either words or integers:

```
[cylc]
    [[parameters]]
        foo = 1..5
        bar = 1..5..2
        baz = pub, qux, bol
```

Hint: Remember that Cylc automatically inserts an underscore between the task and the parameter, e.g. the following lines are equivalent:

```
task<baz=pub>
task_pub
```

Hint: When using integer parameters, to prevent confusion, Cylc prefixes the parameter value with the parameter name. For example:

```
[scheduling]
    [[dependencies]]
        graph = """
            # task<bar> would result in:
            task_bar1
            task_bar3
            task_bar5

            # task<baz> would result in:
            task_pub
            task_qux
            task_bol
        """
```

Using parameters the `get_observations` configuration could be written like so:

```
[scheduling]
   [[dependencies]]
       [[[T00/PT3H]]]
           graph = """
               get_observations<station> => consolidate_observations
           """
[runtime]
    [[get_observations<station>]]
        script = get-observations
        [[[environment]]]
            API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

    [[get_observations<station=belmullet>]]
        [[[environment]]]
            SITE_ID = 3976
    [[get_observations<station=camborne>]]
        [[[environment]]]
            SITE_ID = 3808
    [[get_observations<station=heathrow>]]
        [[[environment]]]
            SITE_ID = 3772
    [[get_observations<station=shetland>]]
```

(continues on next page)

(continued from previous page)

```
        [[[environment]]]
            SITE_ID = 3005
```

For more information see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

Practical

This practical continues on from the Jinja2 practical.

4. **Use Parameterisation To Consolidate The `get_observations` Tasks.**

   Next we will parameterise the `get_observations` tasks.

   Add a parameter called `station`:

```
 [cylc]
     UTC mode = True
+    [[parameters]]
+        station = belmullet, camborne, heathrow, shetland
```

   Remove the four `get_observations` tasks and insert the following code in their place:

```
[[get_observations<station>]]
    script = get-observations
    [[[environment]]]
        API_KEY = {{ API_KEY }}
```

   Using `cylc get-config` you should see that Cylc replaces the `<station>` with each of the stations in turn, creating a new task for each:

```
cylc get-config . --sparse -i "[runtime]"
```

   The `get_observations` tasks are now missing the `SITE_ID` environment variable. Add a new section for each station with a `SITE_ID`:

```
[[get_observations<station=heathrow>]]
    [[[environment]]]
        SITE_ID = 3772
```

Hint: The relevant IDs are:

- Belmullet - `3976`
- Camborne - `3808`
- Heathrow - `3772`
- Shetland - `3005`

Solution

```
[[get_observations<station=belmullet>]]
    [[[environment]]]
        SITE_ID = 3976
[[get_observations<station=camborne>]]
    [[[environment]]]
        SITE_ID = 3808
```

(continues on next page)

(continued from previous page)

```
[[get_observations<station=heathrow>]]
    [[[environment]]]
        SITE_ID = 3772
[[get_observations<station=shetland>]]
    [[[environment]]]
        SITE_ID = 3005
```

Using `cylc get-config` you should now see four `get_observations` tasks, each with a `script`, an `API_KEY` and a `SITE_ID`:

```
cylc get-config . --sparse -i "[runtime]"
```

Finally we can use this parameterisation to simplify the suite's graphing. Replace the `get_observations` lines in the graph with `get_observations<station>`:

```
 [[[PT3H]]]
     # Repeat every three hours starting at the initial cycle point.
     graph = """
-        get_observations_belmullet => consolidate_observations
-        get_observations_camborne => consolidate_observations
-        get_observations_heathrow => consolidate_observations
-        get_observations_shetland => consolidate_observations
+        get_observations<station> => consolidate_observations
     """
```

Hint: The `cylc get-config` command does not expand parameters or families in the graph so you must use `cylc graph` to inspect changes to the graphing.

5. **Use Parameterisation To Consolidate The `post_process` Tasks.**

   At the moment we only have one `post_process` task (`post_process_exeter`), but suppose we wanted to add a second task for Edinburgh.

   Create a new parameter called `site` and set it to contain `exeter` and `edinburgh`. Parameterise the `post_process` task using this parameter.

   Hint: The first argument to the `post-process` task is the name of the site. We can use the `CYLC_TASK_PARAM_site` environment variable to avoid having to write out this section twice.

   Solution

   First we must create the `site` parameter:

```
 [cylc]
     UTC mode = True
     [[parameters]]
         station = belmullet, camborne, heathrow, shetland
+        site = exeter, edinburgh
```

   Next we parameterise the task in the graph:

```
-get_rainfall => forecast => post_process_exeter
+get_rainfall => forecast => post_process<site>
```

   And also the runtime:

```
-[[post_process_exeter]]
+[[post_process<site>]]
      # Generate a forecast for Exeter 60 minutes in the future.
-     script = post-process exeter 60
+     script = post-process $CYLC_TASK_PARAM_site 60
```

3.3.4 The `cylc get-config` Command

The `cylc get-config` command reads in then prints out the `suite.rc` file to the terminal.

Throughout this section we will be introducing methods for consolidating the `suite.rc` file, the `cylc get-config` command can be used to "expand" the `suite.rc` file back to its full form.

Note: The main use of `cylc get-config` is inspecting the `[runtime]` section of a suite. The `cylc get-config` command does not expand parameterisations and families in the suite's graph. To inspect the graphing use the `cylc graph` command.

Call `cylc get-config` with the path of the suite (`.` if you are already in the suite directory) and the `--sparse` option which hides default values.

```
cylc get-config <path> --sparse
```

To view the configuration of a particular section or setting refer to it by name using the `-i` option (see *The suite.rc File Format* (page 5) for details), e.g:

```
# Print the contents of the [scheduling] section.
cylc get-config <path> --sparse -i '[scheduling]'
# Print the contents of the get_observations_heathrow task.
cylc get-config <path> --sparse -i '[runtime][get_observations_heathrow]'
# Print the value of the script setting in the get_observations_heathrow task
cylc get-config <path> --sparse -i '[runtime][get_observations_heathrow]script'
```

3.3.5 The Three Approaches

The next three sections cover the three consolidation approaches and how we could use them to simplify the suite from the previous tutorial. *Work through them in order!*

- *families* (page 45)
- *jinja2* (page 49)
- *parameters* (page 52)

3.3.6 Which Approach To Use

Each approach has its uses. Cylc permits mixing approaches, allowing us to use what works best for us. As a rule of thumb:

- Families work best consolidating runtime configuration by collecting tasks into broad groups, e.g. groups of tasks which run on a particular machine or groups of tasks belonging to a particular system.
- Jinja2 (http://jinja.pocoo.org/) is good at configuring settings which apply to the entire suite rather than just a single task, as we can define variables then use them throughout the suite.
- Parameterisation works best for describing tasks which are very similar but which have subtly different configurations (e.g. different arguments or environment variables).

CHAPTER

FOUR

FURTHER TOPICS

This section looks at further topics in cylc.

4.1 Clock Triggered Tasks

In a datetime cycling suite the time represented by the cycle points bear no relation to the real-world time. Using clock-triggers we can make tasks wait until their cycle point time before running.

Clock-triggering effectively enables us to tether the "cycle time" to the "real world time" which we refer to as the wall-clock time.

4.1.1 Clock Triggering

When clock-triggering tasks we can use different *offsets* (page 22) to the cycle time as follows:

```
clock-trigger = taskname(CYCLE_OFFSET)
```

Note: Regardless of the offset used, the task still belongs to the cycle from which the offset has been applied.

4.1.2 Example

Our example suite will simulate a clock chiming on the hour.

Within your `~/cylc-run` directory create a new directory called `clock-trigger`:

```
mkdir ~/cylc-run/clock-trigger
cd ~/cylc-run/clock-trigger
```

Paste the following code into a `suite.rc` file:

```
[cylc]
    UTC mode = True # Ignore DST

[scheduling]
    initial cycle point = TODO
    final cycle point = +P1D # Run for one day
    [[dependencies]]
        [[[PT1H]]]
            graph = bell

[runtime]
    [[root]]
        [[[events]]]
```

(continues on next page)

(continued from previous page)

```
            mail events = failed
    [[bell]]
        env-script = eval $(rose task-env)
        script = printf 'bong%.0s\n' $(seq 1 $(cylc cyclepoint --print-hour))
```

Change the initial cycle point to 00:00 this morning (e.g. if it was the first of January 2000 we would write `2000-01-01T00Z`).

We now have a simple suite with a single task that prints "bong" a number of times equal to the (cycle point) hour.

Run your suite using:

```
cylc run clock-trigger
```

Stop the suite after a few cycles using the *stop* button in the `cylc gui`. Notice how the tasks run as soon as possible rather than waiting for the actual time to be equal to the cycle point.

4.1.3 Clock-Triggering Tasks

We want our clock to only ring in real-time rather than the simulated cycle time.

To do this, add the following lines to the `[scheduling]` section of your `suite.rc`:

```
[[special tasks]]
    clock-trigger = bell(PT0M)
```

This tells the suite to clock trigger the `bell` task with a cycle offset of `0` hours.

Save your changes and run your suite.

Your suite should now be running the `bell` task in real-time. Any cycle times that have already passed (such as the one defined by `initial cycle time`) will be run as soon as possible, while those in the future will wait for that time to pass.

At this point you may want to leave your suite running until the next hour has passed in order to confirm the clock triggering is working correctly. Once you are satisfied, stop your suite.

By making the `bell` task a clock triggered task we have made it run in real-time. Thus, when the wall-clock time caught up with the cycle time, the `bell` task triggered.

4.1.4 Adding More Clock-Triggered Tasks

We will now modify our suite to run tasks at quarter-past, half-past and quarter-to the hour.

Open your `suite.rc` and modify the `[runtime]` section by adding the following:

```
[[quarter_past, half_past, quarter_to]]
    script = echo 'chimes'
```

Edit the `[[scheduling]]` section to read:

```
[[special tasks]]
    clock-trigger = bell(PT0M), quarter_past(PT15M), half_past(PT30M), quarter_
↪to(PT45M)
[[dependencies]]
    [[[PT1H]]]
        graph = """
            bell
            quarter_past
            half_past
```

(continues on next page)

(continued from previous page)

```
            quarter_to
        """
```

Note the different values used for the cycle offsets of the clock-trigger tasks.

Save your changes and run your suite using:

```
cylc run clock-trigger now
```

Note: The `now` argument will run your suite using the current time for the initial cycle point.

Again, notice how the tasks trigger until the current time is reached.

Leave your suite running for a while to confirm it is working as expected and then shut it down using the *stop* button in the `cylc gui`.

4.1.5 Summary

- Clock triggers are a type of dependency which cause tasks to wait for the wall-clock time to reach the cycle point time.
- A clock trigger applies only to a single task.
- Clock triggers can only be used in datetime cycling suites.

For more information see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

4.2 Broadcast

This tutorial walks you through using `cylc broadcast` which can be used to change *task runtime configuration* (page 31) in a running suite, on-the-fly.

4.2.1 Purpose

`cylc broadcast` can be used to change any `[runtime]` setting whilst the suite is running.

The standard use of `cylc broadcast` is to update the suite to an unexpected change in configuration, for example modifying the host a task runs on.

4.2.2 Standalone Example

Create a new suite in the `cylc-run` directory called `tutorial-broadcast`:

```
mkdir ~/cylc-run/tutorial-broadcast
cd ~/cylc-run/tutorial-broadcast
```

Copy the following configuration into a `suite.rc` file:

```
[scheduling]
    initial cycle point = 1012
    [[dependencies]]
        [[[R1]]]
            graph = wipe_log => announce
        [[[PT1H]]]
```

(continues on next page)

(continued from previous page)

```
            graph = announce[-PT1H] => announce

[runtime]
    [[wipe_log]]
        # Delete any files in the suite's "share" directory.
        script = rm "${CYLC_SUITE_SHARE_DIR}/knights" || true

    [[announce]]
        script = echo "${CYLC_TASK_CYCLE_POINT} - ${MESSAGE}" >> "${FILE}"
        [[[environment]]]
            WORD = ni
            MESSAGE = We are the knights who say \"${WORD}\"!
            FILE = "${CYLC_SUITE_SHARE_DIR}/knights"
```

We now have a suite with an `announce` task which runs every hour, writing a message to a log file (`share/knights`) when it does so. For the first cycle the log entry will look like this:

```
10120101T0000Z - We are the knights who say "ni"!
```

The `cylc broadcast` command enables us to change runtime configuration whilst the suite is running. For instance we could change the value of the `WORD` environment variable using the command:

```
cylc broadcast tutorial-broadcast -n announce -s "[environment]WORD=it"
```

- `tutorial-broadcast` is the name of the suite.
- `-n announce` tells Cylc we want to change the runtime configuration of the `announce` task.
- `-s "[environment]WORD=it"` changes the value of the `WORD` environment variable to `it`.

Run the suite then try using the `cylc broadcast` command to change the message:

```
cylc run tutorial-broadcast
cylc broadcast tutorial-broadcast -n announce -s "[environment]WORD=it"
```

Inspect the `share/knights` file, you should see the message change at certain points.

Stop the suite:

```
cylc stop tutorial-broadcast
```

4.2.3 In-Situ Example

We can call `cylc broadcast` from within a task's script. This effectively provides the ability for tasks to communicate between themselves.

It is almost always better for tasks to communicate using files but there are some niche situations where communicating via `cylc broadcast` is justified. This tutorial walks you through using `cylc broadcast` to communicate between tasks.

Add the following recurrence to the `dependencies` section:

```
[[[PT3H]]]
    graph = announce[-PT1H] => change_word => announce
```

The `change_word` task runs the `cylc broadcast` command to randomly change the `WORD` environment variable used by the `announce` task.

Add the following runtime configuration to the `runtime` section:

```
[[change_word]]
    script = """
        # Select random word.
        IFS=',' read -r -a WORDS <<< $WORDS
        WORD=${WORDS[$(date +%s) % ${#WORDS[@]}]}

        # Broadcast random word to the announce task.
        cylc broadcast $CYLC_SUITE_NAME -n announce -s "[environment]WORD=${WORD}"
    """
    [[[environment]]]
        WORDS = ni, it, ekke ekke ptang zoo boing
```

Run the suite and inspect the log. You should see the message change randomly after every third entry (because the `change_word` task runs every 3 hours) e.g:

```
10120101T0000Z - We are the knights who say "ni"!
10120101T0100Z - We are the knights who say "ni"!
10120101T0200Z - We are the knights who say "ni"!
10120101T0300Z - We are the knights who say "ekke ekke ptang zoo boing!"
```

Stop the suite:

```
cylc stop tutorial-broadcast
```

4.3 Family Triggers

To reduce duplication in the graph is is possible to write dependencies using collections of tasks called families).

This tutorial walks you through writing such dependencies using family triggers.

4.3.1 Explanation

Dependencies between tasks can be written using a qualifier to describe the task state that the dependency refers to (e.g. `succeed` `fail`, etc). If a dependency does not use a qualifier then it is assumed that the dependency refers to the `succeed` state e.g:

```
bake_bread => sell_bread          # sell_bread is dependent on bake_bread␣
↪succeeding.
bake_bread:succeed => sell_bread  # sell_bread is dependent on bake_bread␣
↪succeeding.
sell_bread:fail => through_away   # through_away is dependent on sell_bread␣
↪failing.
```

The left-hand side of a dependency (e.g. `sell_bread:fail`) is referred to as the trigger.

When we write a trigger involving a family, special qualifiers are required to specify whether the dependency is concerned with *all* or *any* of the tasks in that family reaching the desired state e.g:

- `succeed-all`
- `succeed-any`
- `fail-all`

Such triggers are referred to as family triggers

Foo `cylc gui` bar

4.3.2 Example

Create a new suite called `tutorial-family-triggers`:

```
mkdir ~/cylc-run/tutorial-family-triggers
cd ~/cylc-run/tutorial-family-triggers
```

Paste the following configuration into the `suite.rc` file:

```
[cylc]
    UTC mode = True # Ignore DST
[scheduling]
    [[dependencies]]
        graph = visit_mine => MINERS
[runtime]
    [[visit_mine]]
        script = sleep 5; echo 'off to work we go'

    [[MINERS]]
        script = """
sleep 5;
if (($RANDOM % 2)); then
    echo 'Diamonds!'; true;
else
    echo 'Nothing...'; false;
fi
"""
    [[doc, grumpy, sleepy, happy, bashful, sneezy, dopey]]
        inherit = MINERS
```

You have now created a suite that:

- Has a `visit_mine` task that sleeps for 5 seconds then outputs a message.
- Contains a `MINERS` family with a command in it that randomly succeeds or fails.
- Has 7 tasks that inherit from the `MINERS` family.

Open the `cylc gui` then run the suite by pressing the "play" button (top left hand corner) then clicking *Start*:

```
cylc gui tutorial-family-triggers &
```

You should see the `visit_mine` task run, then trigger the members of the `MINERS` family. Note that some of the `MINERS` tasks may fail so you will need to stop your suite using the "stop" button in the `cylc gui` in order to allow it to shutdown.

4.3.3 Family Triggering: Success

As you will have noticed by watching the suite run, some of the tasks in the `MINERS` family succeed and some fail.

We would like to add a task to sell any diamonds we find, but wait for all the miners to report back first so we only make the one trip.

We can address this by using *family triggers*. In particular, we are going to use the `finish-all` trigger to check for all members of the `MINERS` family finishing, and the `succeed-any` trigger to check for any of the tasks in the `MINERS` family succeeding.

Open your `suite.rc` file and change the `[[dependencies]]` to look like this:

```
[[dependencies]]
    graph = """visit_mine => MINERS
               MINERS:finish-all & MINERS:succeed-any => sell_diamonds"""
```

Then, add the following task to the [runtime] section:

```
[[sell_diamonds]]
   script = sleep 5
```

These changes add a sell_diamonds task to the suite which is run once all the MINERS tasks have finished and if any of them have succeeded.

Save your changes and run your suite. You should see the new sell_diamonds task being run once all the miners have finished and at least one of them has succeeded. As before, stop your suite using the "stop" button in the cylc gui.

4.3.4 Family Triggering: Failure

Cylc also allows us to trigger off failure of tasks in a particular family.

We would like to add another task to close down unproductive mineshafts once all the miners have reported back and had time to discuss their findings.

To do this we will make use of family triggers in a similar manner to before.

Open your suite.rc file and change the [[dependencies]] to look like this:

```
[[dependencies]]
    graph = """visit_mine => MINERS
               MINERS:finish-all & MINERS:succeed-any => sell_diamonds
               MINERS:finish-all & MINERS:fail-any => close_shafts
               close_shafts => !MINERS
               """
```

Alter the [[sell_diamonds]] section to look like this:

```
[[close_shafts, sell_diamonds]]
    script = sleep 5
```

These changes add a close_shafts task which is run once all the MINERS tasks have finished and any of them have failed. On completion it applies a *suicide trigger* to the MINERS family in order to allow the suite to shutdown.

Save your changes and run your suite. You should see the new close_shafts run should any of the MINERS tasks be in the failed state once they have all finished.

Tip: See the *Suicide Triggers* (page 77) tutorial for handling task failures.

4.3.5 Different Triggers

Other family qualifiers beyond those covered in the example are also available.

The following types of "all" qualifier are available:

- :start-all - all the tasks in the family have started
- :succeed-all - all the tasks in the family have succeeded
- :fail-all - all the tasks in the family have failed
- :finish-all - all the tasks in the family have finished

The following types of "any" qualifier are available:

- :start-any - at least one task in the family has started
- :succeed-any - at least one task in the family has succeeded

- `:fail-any` - at least one task in the family has failed
- `:finish-any` - at least one task in the family has finished

4.3.6 Summary

- Family triggers allow you to write dependencies for collections of tasks.
- Like task triggers, family triggers can be based on success, failure, starting and finishing of tasks in a family.
- Family triggers can trigger off either *all* or *any* of the tasks in a family.

4.4 Inheritance

We have seen in the *runtime tutorial* (page 45) how tasks can be grouped into families.

In this tutorial we will look at nested families, inheritance order and multiple inheritance.

4.4.1 Inheritance Hierarchy

Create a new suite by running the command:

```
rose tutorial inheritance-tutorial
cd ~/cylc-run/inheritance-tutorial
```

You will now have a `suite.rc` file that defines two tasks each representing a different aircraft, the Airbus A380 jumbo jet and the Robson R44 helicopter:

```
[scheduling]
    [[dependencies]]
        graph = a380 & r44
```

(continues on next page)

(continued from previous page)

```
[runtime]
    [[VEHICLE]]
        init-script = echo 'Boarding'
        pre-script = echo 'Departing'
        post-script = echo 'Arriving'

    [[AIR_VEHICLE]]
        inherit = VEHICLE
        [[[meta]]]
            description = A vehicle which can fly.
    [[AIRPLANE]]
        inherit = AIR_VEHICLE
        [[[meta]]]
            description = An air vehicle with fixed wings.
        [[[environment]]]
            CAN_TAKE_OFF_VERTICALLY = false
    [[HELICOPTER]]
        inherit = AIR_VEHICLE
        [[[meta]]]
            description = An air vehicle with rotors.
        [[[environment]]]
            CAN_TAKE_OFF_VERTICALLY = true

    [[a380]]
        inherit = AIRPLANE
        [[[meta]]]
            title = Airbus A380 Jumbo-Jet.
    [[r44]]
        inherit = HELICOPTER
        [[[meta]]]
            title = Robson R44 Helicopter.
```

Note: The `[meta]` section is a freeform section where we can define metadata to be associated with a task, family or the suite itself.

This metadata should not be mistaken with Rose conf-meta.

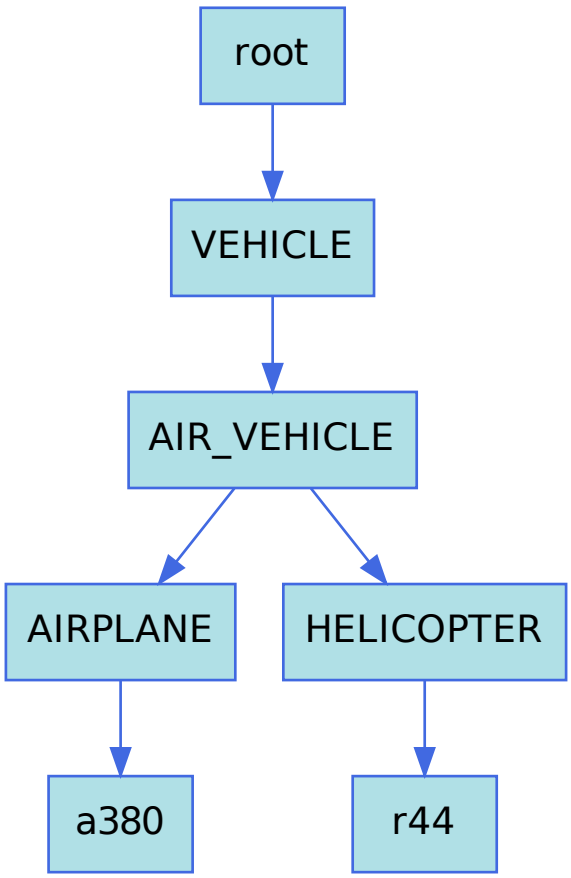
Reminder

By convention we write family names in upper case (with the exception of the special `root` family) and task names in lower case.

These two tasks sit at the bottom of an inheritance tree. The `cylc graph` command has an option (`-n`) for drawing such inheritance hierarchies:

```
cylc graph -n . &
```

Running this command will generate the following output:



Note: The `root` family sits at the top of the inheritance tree as all tasks/families automatically inherit it:

Cylc handles inheritance by starting with the root family and working down the inheritance tree applying each section in turn.

To see the resulting configuration for the `a380` task use the `cylc get-config` command:

```
cylc get-config . --sparse -i "[runtime][a380]"
```

You should see some settings which have been inherited from the `VEHICLE` and `AIRPLANE` families as well as a couple defined in the `a380` task.

```
init-script = echo 'Boarding'                          # Inherited from VEHICLE
pre-script = echo 'Departing'                          # Inherited from VEHICLE
post-script = echo 'Arriving'                          # Inherited from VEHICLE
inherit = AIRPLANE                                     # Defined in a380
[[[meta]]]
    description = An air vehicle with fixed wings.     # Inherited from AIR_VEHICLE -␣
↪overwritten by AIRPLANE
    title = Airbus A380 Jumbo-Jet.                     # Defined in a380
[[[environment]]]
    CAN_TAKE_OFF_VERTICALLY = false                    # Inherited from AIRPLANE
```

Note that the `description` setting is defined in the `AIR_VEHICLE` family but is overwritten by the value

specified in the AIRPLANE family.

4.4.2 Multiple Inheritance

Next we want to add a vehicle called the V-22 Osprey to the suite. The V-22 is a cross between a plane and a helicopter - it has wings but can take-off and land vertically.

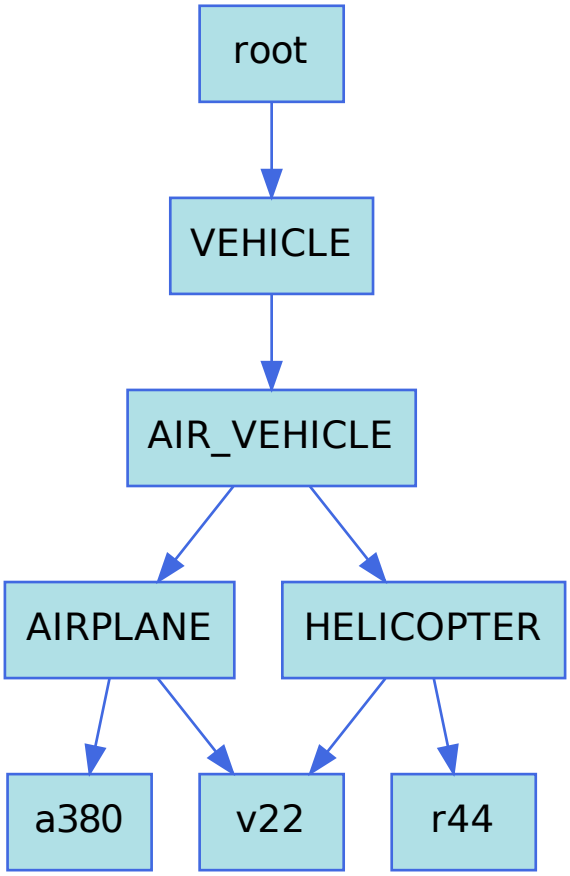
As the V-22 can be thought of as both a plane and a helicopter we want it to inherit from both the AIRPLANE and HELICOPTER families. In Cylc we can inherit from multiple families by separating their names with commas:

Add the following task to your `suite.rc` file.

```
[[v22]]
    inherit = AIRPLANE, HELICOPTER
    [[[meta]]]
        title = V-22 Osprey Military Aircraft.
```

Refresh your `cylc graph` window or re-run the `cylc graph` command.

The inheritance hierarchy should now look like this:



Inspect the configuration of the `v22` task using the `cylc get-config` command.

Hint

```
cylc get-config . --sparse -i "[runtime][v22]"
```

You should see that the `CAN_TASK_OFF_VERTICALLY` environment variable has been set to `false` which isn't right. This is because of the order in which inheritance is applied.

Cylc handles multiple-inheritance by applying each family from right to left. For the `v22` task we specified `inherit = AIRPLANE, HELICOPTER` so the `HELICOPTER` family will be applied first and the `AIRPLANE` family after.

The inheritance order would be as follows:

```
root
VEHICLE
AIR_VEHICLE
HELICOPTER    # sets "CAN_TAKE_OFF_VERTICALLY to "true"
AIRPLANE      # sets "CAN_TAKE_OFF_VERTICALLY to "false"
v22
```

We could fix this problem by changing the order of inheritance:

```
inherit = HELICOPTER, AIRPLANE
```

Now the `HELICOPTER` family is applied second so its values will override any in the `AIRPLANE` family.

```
root
VEHICLE
AIR_VEHICLE
AIRPLANE      # sets "CAN_TAKE_OFF_VERTICALLY to "false"
HELICOPTER    # sets "CAN_TAKE_OFF_VERTICALLY to "true"
v22
```

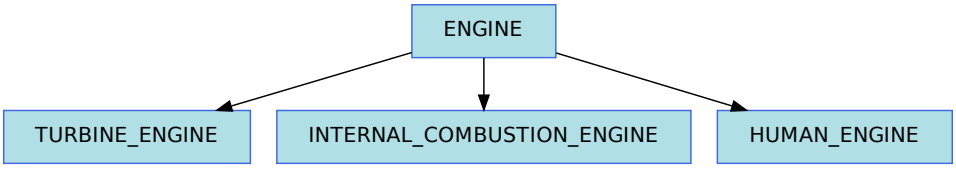
Inspect the configuration of the `v22` task using `cylc get-config` to confirm this.

4.4.3 More Inheritance

We will now add some more families and tasks to the suite.

Engine Type

Next we will define four families to represent three different types of engine.



Each engine type should set an environment variable called `FUEL` which we will assign to the following values:

- Turbine - kerosene
- Internal Combustion - petrol
- Human - pizza

Add lines to the `runtime` section to represent these four families.

Solution

```
[[ENGINE]]
[[TURBINE_ENGINE]]
    inherit = ENGINE
    [[[environment]]]
        FUEL = kerosene
[[INTERNAL_COMBUSTION_ENGINE]]
    inherit = ENGINE
    [[[environment]]]
        FUEL = petrol
[[HUMAN_ENGINE]]
    inherit = ENGINE
    [[[environment]]]
        FUEL = pizza
```

We now need to make the three aircraft inherit from one of the three engines. The aircraft use the following types of engine:

- A380 - turbine
- R44 - internal combustion
- V22 - turbine

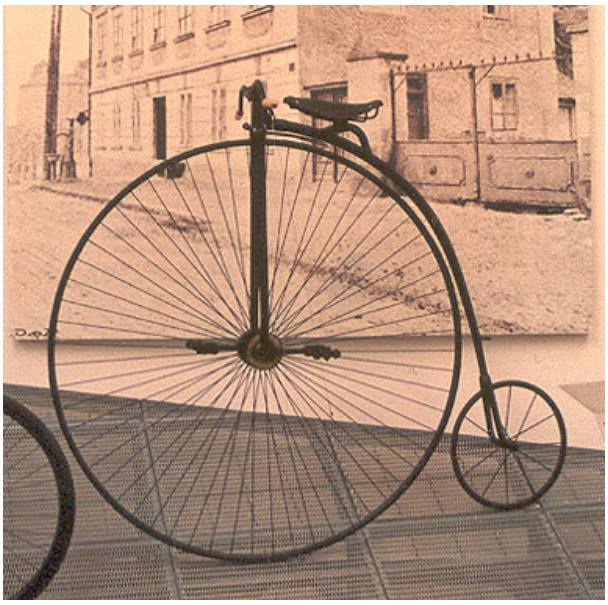
Modify the three tasks so that they inherit from the relevant engine families.

Solution

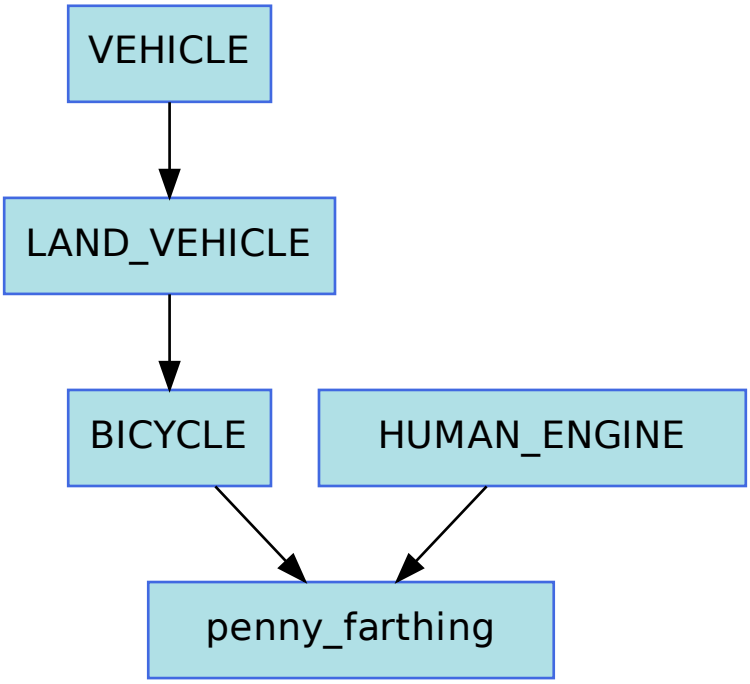
```
[[a380]]
    inherit = AIRPLANE, TURBINE_ENGINE
    [[[meta]]]
        title = Airbus A380 Jumbo-Jet.
[[r44]]
    inherit = HELICOPTER, INTERNAL_COMBUSTION_ENGINE
    [[[meta]]]
        title = Robson R44 Helicopter.
[[v22]]
    inherit = AIRPLANE, HELICOPTER, TURBINE_ENGINE
    [[[meta]]]
        title = V-22 Ofsprey Military Aircraft.
```

Penny Farthing

Next we want to add a new type of vehicle, an old-fashioned bicycle called a penny farthing.



To do this we will need to add two new families, `LAND_VEICHLE` and `BICYCLE` as well as a new task, `penny_farthing` related in the following manner:



Add lines to the `runtime` section to represent the two new families and one task outlined above.

Add a description (`[meta]description`) to the `LAND_VEHICLE` and `BICYCLE` families and a title (`[meta]title`) to the `penny_farthing` task.

Solution

```
[[LAND_VEHICLE]]
    inherit = VEHICLE
    [[[meta]]]
        description = A vehicle which can travel over the ground.

[[BICYCLE]]
    inherit = LAND_VEHICLE
    [[[meta]]]
        description = A small two-wheeled vehicle.

[[penny_farthing]]
    inherit = BICYCLE, HUMAN_ENGINE
    [[[meta]]]
        title = An old-fashioned bicycle.
```

Using `cylc get-config` to inspect the configuration of the `penny_farthing` task we can see that it inherits its settings from the `VEHICLE`, `BICYCLE` and `HUMAN_ENGINE` families.

```
inherit = BICYCLE, HUMAN_ENGINE
init-script = echo 'Boarding'  # Inherited from VEHICLE
pre-script = echo 'Departing'  # Inherited from VEHICLE
post-script = echo 'Arriving'  # Inherited from VEHICLE
[[[environment]]]
```

(continues on next page)

(continued from previous page)

```
    FUEL = pizza                # Inherited from HUMAN_ENGINE
[[[meta]]]
    description = A small two-wheeled vehicle.  # Inherited from LAND_VEHICLE -␣
↪overwritten by BICYCLE
    title = An old-fashioned bicycle.           # Defined in penny_farthing
```

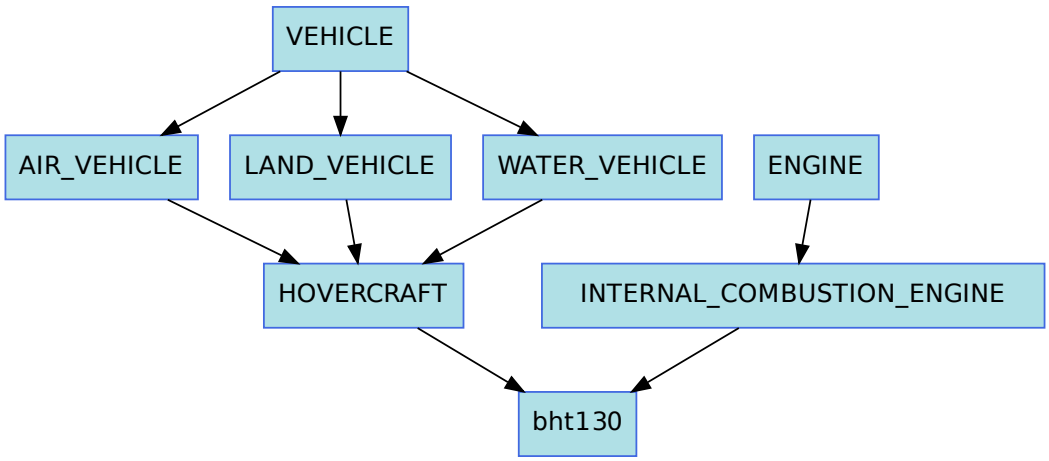
Hint

```
cylc get-config . --sparse -i "[runtime]penny_farthing"
```

Hovercraft

We will now add a hovercraft called the Hoverwork BHT130, better known to some as the Isle Of Wight Ferry.

Hovercraft can move over both land and water and in some respects can be thought of as flying vehicles.



Write new families and one new task to represent the above structure.

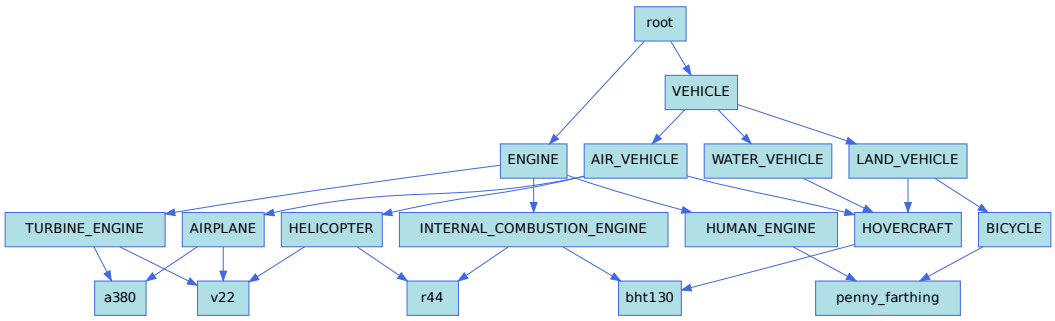
Add a description (`[meta]description`) to the `WATER_VEHICLE` and `HOVERCRAFT` families and a title (`[meta]title`) to the `bht130` task.

Solution

```
[[WATER_VEHICLE]]
    inherit = VEHICLE
    [[[meta]]]
        description = A vehicle which can travel over water.

[[HOVERCRAFT]]
    inherit = LAND_VEHICLE, AIR_VEHICLE, WATER_VEHICLE
    [[[meta]]]
        description = A vehicle which can travel over ground, water and ice.

[[bht130]]
    inherit = HOVERCRAFT, INTERNAL_COMBUSTION_ENGINE
    [[[meta]]]
        title = Griffon Hoverwork BHT130 (Isle Of Whight Ferry).
```

4.4.4 Finished Suite

You should now have a suite with an inheritance hierarchy which looks like this:



4.5 Queues

Queues are used to put a limit on the number of tasks that will be active at any one time, even if their dependencies are satisfied. This avoids swamping systems with too many tasks at once.

4.5.1 Example

In this example, our suite manages a particularly understaffed restaurant.

Create a new suite called `queues-tutorial`:

```
rose tutorial queues-tutorial
cd ~/cylc-run/queues-tutorial
```

You will now have a `suite.rc` file that looks like this:

```
[scheduling]
    [[dependencies]]
        graph = """
            open_restaurant => steak1 & steak2 & pasta1 & pasta2 & pasta3 & \
```

(continues on next page)

(continued from previous page)

```
                                pizza1 & pizza2 & pizza3 & pizza4
            steak1 => ice_cream1
            steak2 => cheesecake1
            pasta1 => ice_cream2
            pasta2 => sticky_toffee1
            pasta3 => cheesecake2
            pizza1 => ice_cream3
            pizza2 => ice_cream4
            pizza3 => sticky_toffee2
            pizza4 => ice_cream5
        """

[runtime]
    [[open_restaurant]]
    [[MAINS]]
    [[DESSERT]]
    [[steak1,steak2,pasta1,pasta2,pasta3,pizza1,pizza2,pizza3,pizza4]]
        inherit = MAINS
    [[ice_cream1,ice_cream2,ice_cream3,ice_cream4,ice_cream5]]
        inherit = DESSERT
    [[cheesecake1,cheesecake2,sticky_toffee1,sticky_toffee2]]
        inherit = DESSERT
```

Note: In graph sections backslash (\) is a line continuation character i.e. the following two examples are equivalent:

```
foo => bar & \
       baz
```

```
foo => bar & baz
```

Open the `cylc gui` then run the suite:

```
cylc gui queues-tutorial &
cylc run queues-tutorial
```

You will see that all the `steak`, `pasta`, and `pizza` tasks are run at once, swiftly followed by all the `ice_cream`, `cheesecake`, `sticky_toffee` tasks as the customers order from the dessert menu.

This will overwhelm our restaurant staff! The chef responsible for `MAINS` can only handle 3 tasks at any given time, and the `DESSERT` chef can only handle 2.

We need to add some queues. Add a `[queues]` section to the `[scheduling]` section like so:

```
[scheduling]
    [[queues]]
        [[[mains_chef_queue]]]
            limit = 3  # Only 3 mains dishes at one time.
            members = MAINS
        [[[dessert_chef_queue]]]
            limit = 2  # Only 2 dessert dishes at one time.
            members = DESSERT
```

Re-open the `cylc gui` if you have closed it and re-run the suite.

You should see that there are now never more than 3 active `MAINS` tasks running and never more than 2 active `DESSERT` tasks running.

The customers will obviously have to wait!

4.5.2 Further Reading

For more information, see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

4.6 Retries

Retries allow us to automatically re-submit tasks which have failed due to failure in submission or execution.

4.6.1 Purpose

Retries can be useful for tasks that may occasionally fail due to external events, and are routinely fixable when they do - an example would be a task that is dependent on a system that experiences temporary outages.

If a task fails, the Cylc retry mechanism can resubmit it after a pre-determined delay. An environment variable, `$CYLC_TASK_TRY_NUMBER` is incremented and passed into the task - this means you can write your task script so that it changes behaviour accordingly.

4.6.2 Example

Create a new suite by running the following commands:

```
rose tutorial retries-tutorial
cd retries-tutorial
```

You will now have a suite with a `roll_doubles` task which simulates trying to roll doubles using two dice:

```
[cylc]
    UTC mode = True # Ignore DST

[scheduling]
    [[dependencies]]
        graph = start => roll_doubles => win

[runtime]
    [[start]]
    [[win]]
    [[roll_doubles]]
        script = """
sleep 10
RANDOM=$$  # Seed $RANDOM
DIE_1=$((RANDOM%6 + 1))
DIE_2=$((RANDOM%6 + 1))
echo "Rolled $DIE_1 and $DIE_2..."
if (($DIE_1 == $DIE_2)); then
   echo "doubles!"
```

(continues on next page)

(continued from previous page)

```
else
   exit 1
fi
"""
```

4.6.3 Running Without Retries

Let's see what happens when we run the suite as it is. Open the `cylc gui`:

```
cylc gui retries-tutorial &
```

Then run the suite:

```
cylc run retries-tutorial
```

Unless you're lucky, the suite should fail at the roll_doubles task.

Stop the suite:

```
cylc stop retries-tutorial
```

4.6.4 Configuring Retries

We need to tell Cylc to retry it a few times - replace the line `[[roll_doubles]]` in the `suite.rc` file with:

```
[[roll_doubles]]
    [[[job]]]
        execution retry delays = 5*PT6S
```

This means that if the `roll_doubles` task fails, Cylc expects to retry running it 5 times before finally failing. Each retry will have a delay of 6 seconds.

We can apply multiple retry periods with the `execution retry delays` setting by separating them with commas, for example the following line would tell Cylc to retry a task four times, once after 15 seconds, then once after 10 minutes, then once after one hour then once after three hours.

```
execution retry delays = PT15S, PT10M, PT1H, PT3H
```

4.6.5 Running With Retries

If you closed it, re-open the `cylc gui`:

```
cylc gui retries-tutorial &
```

Re-run the suite:

```
cylc run retries-tutorial
```

What you should see is Cylc retrying the `roll_doubles` task. Hopefully, it will succeed (there is only about a about a 1 in 3 chance of every task failing) and the suite will continue.

4.6.6 Altering Behaviour

We can alter the behaviour of the task based on the number of retries, using `$CYLC_TASK_TRY_NUMBER`.

Change the `script` setting for the `roll_doubles` task to this:

```
sleep 10
RANDOM=$$  # Seed $RANDOM
DIE_1=$((RANDOM%6 + 1))
DIE_2=$((RANDOM%6 + 1))
echo "Rolled $DIE_1 and $DIE_2..."
if (($DIE_1 == $DIE_2)); then
    echo "doubles!"
elif (($CYLC_TASK_TRY_NUMBER >= 2)); then
    echo "look over there! ..."
    echo "doubles!"  # Cheat!
else
    exit 1
fi
```

If your suite is still running, stop it, then run it again.

This time, the task should definitely succeed before the third retry.

4.6.7 Further Reading

For more information see the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

4.7 Suicide Triggers

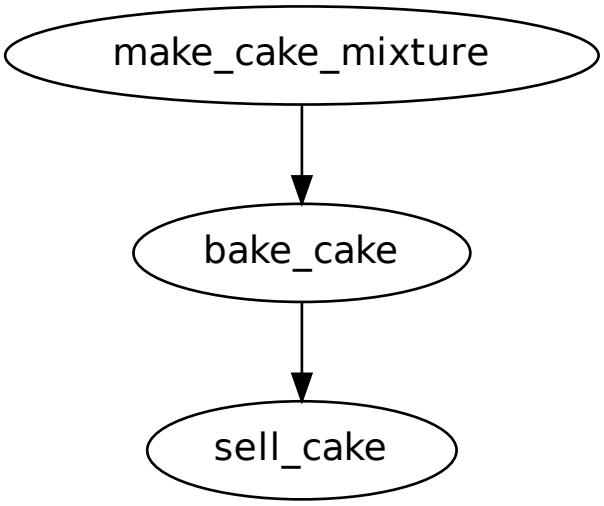
Suicide triggers allow us to remove a task from the suite's graph whilst the suite is running.

The main use of suicide triggers is for handling failures in the workflow.

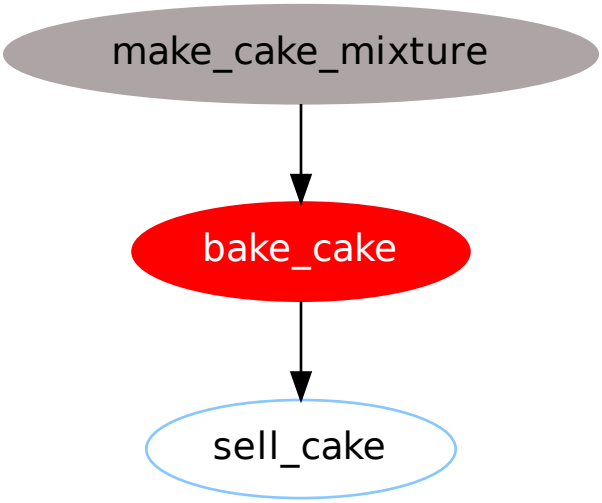
4.7.1 Stalled Suites

Imagine a bakery which has a workflow that involves making cake.

```
make_cake_mixture => bake_cake => sell_cake
```



There is a 50% chance that the cake will turn out fine, and a 50% chance that it will get burnt. In the case that we burn the cake the workflow gets stuck.



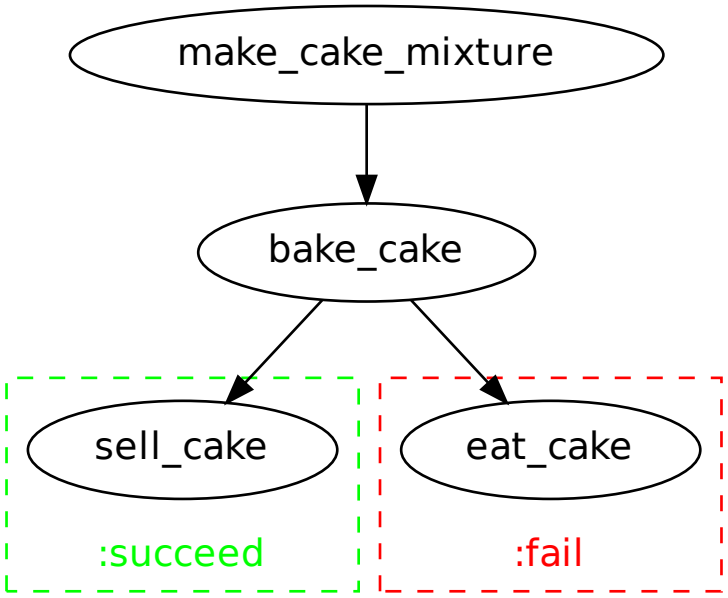
In this event the `sell_cake` task will be unable to run as it depends on `bake_cake`. We would say that this suite has stalled. When Cylc detects that a suite has stalled it sends you an email to let you know that the suite has got stuck and requires human intervention to proceed.

4.7.2 Handling Failures

In order to prevent the suite from entering a stalled state we need to handle the failure of the `bake_cake` task.

At the bakery if they burn a cake they eat it and make another.

The following diagram outlines this workflow with its two possible pathways, `:succeed` in the event that `bake_cake` is successful and `:fail` otherwise.



We can add this logic to our workflow using the `fail` qualifier.

```
bake_cake => sell_cake
bake_cake:fail => eat_cake
```

Reminder

If you don't specify a qualifier Cylc assumes you mean `:succeed` so the following two lines are equivalent:

```
foo => bar
foo:succeed => bar
```

4.7.3 Why Do We Need To Remove Tasks From The Graph?

Create a new suite called `suicide-triggers`:

```
mkdir -p ~/cylc-run/suicide-triggers
cd ~/cylc-run/suicide-triggers
```

Paste the following code into the `suite.rc` file:

```
[scheduling]
   cycling mode = integer
   initial cycle point = 1
   [[dependencies]]
       [[[P1]]]
           graph = """
```

(continues on next page)

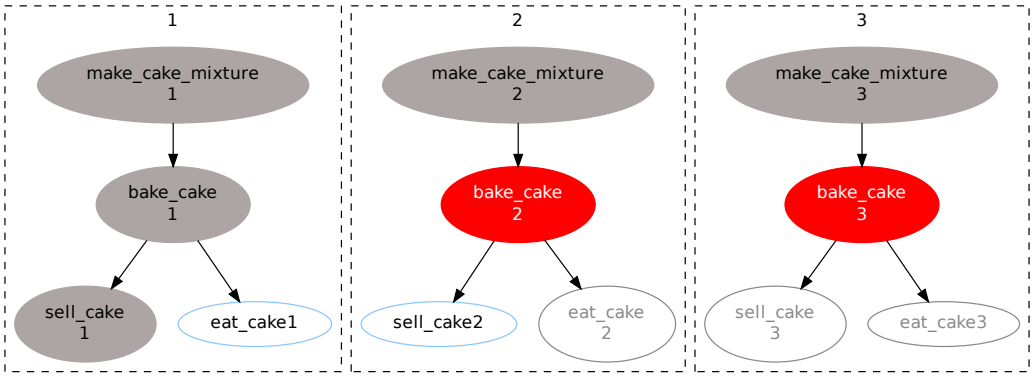
(continued from previous page)

```
            make_cake_mixture => bake_cake => sell_cake
            bake_cake:fail => eat_cake
        """
[runtime]
    [[root]]
        script = sleep 2
    [[bake_cake]]
        # Random outcome 50% chance of success 50% chance of failure.
        script = sleep 2; if (( $RANDOM % 2 )); then true; else false; fi
```

Open the `cylc gui` and run the suite:

```
cylc gui suicide-triggers &
cylc run suicide-triggers
```

The suite will run for three cycles then get stuck. You should see something similar to the diagram below. As the `bake_cake` task fails randomly what you see might differ slightly. You may receive a "suite stalled" email.



The reason the suite stalls is that, by default, Cylc will run a maximum of three cycles concurrently. As each cycle has at least one task which hasn't either succeeded or failed Cylc cannot move onto the next cycle.

Tip: For more information search `max active cycle points` in the Cylc User Guide (http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide).

You will also notice that some of the tasks (e.g. `eat_cake` in cycle `2` in the above example) are drawn in a faded gray. This is because these tasks have not yet been run in earlier cycles and as such cannot run.

4.7.4 Removing Tasks From The Graph

In order to get around these problems and prevent the suite from stalling we must remove the tasks that are no longer needed. We do this using suicide triggers.

A suicide trigger is written like a normal dependency but with an exclamation mark in-front of the task on the right-hand-side of the dependency meaning *"remove the following task from the graph at the current cycle point."*

For example the following graph string would remove the task `bar` from the graph if the task `foo` were to succeed.

```
foo => ! bar
```

There are three cases where we would need to remove a task in the cake-making example:

1. If the `bake_cake` task succeeds we don't need the `eat_cake` task so should remove it.

```
bake_cake => ! eat_cake
```

2. If the `bake_cake` task fails we don't need the `sell_cake` task so should remove it.

```
bake_cake:fail => ! sell_cake
```

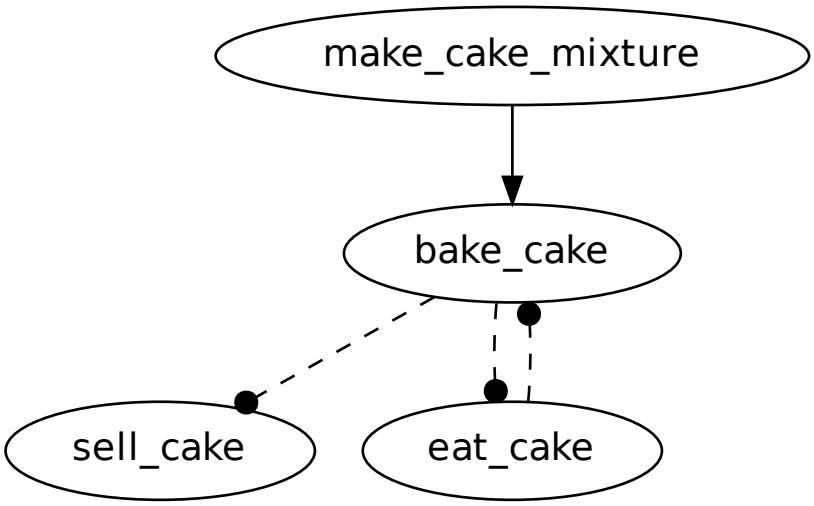
3. If the `bake_cake` task fails then we will need to remove it else the suite will stall. We can do this after the `eat_cake` task has succeeded.

```
eat_cake => ! bake_cake
```

Add the following three lines to the suite's graph:

```
bake_cake => ! eat_cake
bake_cake:fail => ! sell_cake
eat_cake => ! bake_cake
```

We can view suicide triggers in `cylc graph` by un-selecting the *Ignore Suicide Triggers* button in the toolbar. Suicide triggers will then appear as dashed lines with circular endings. You should see something like this:



4.7.5 Downstream Dependencies

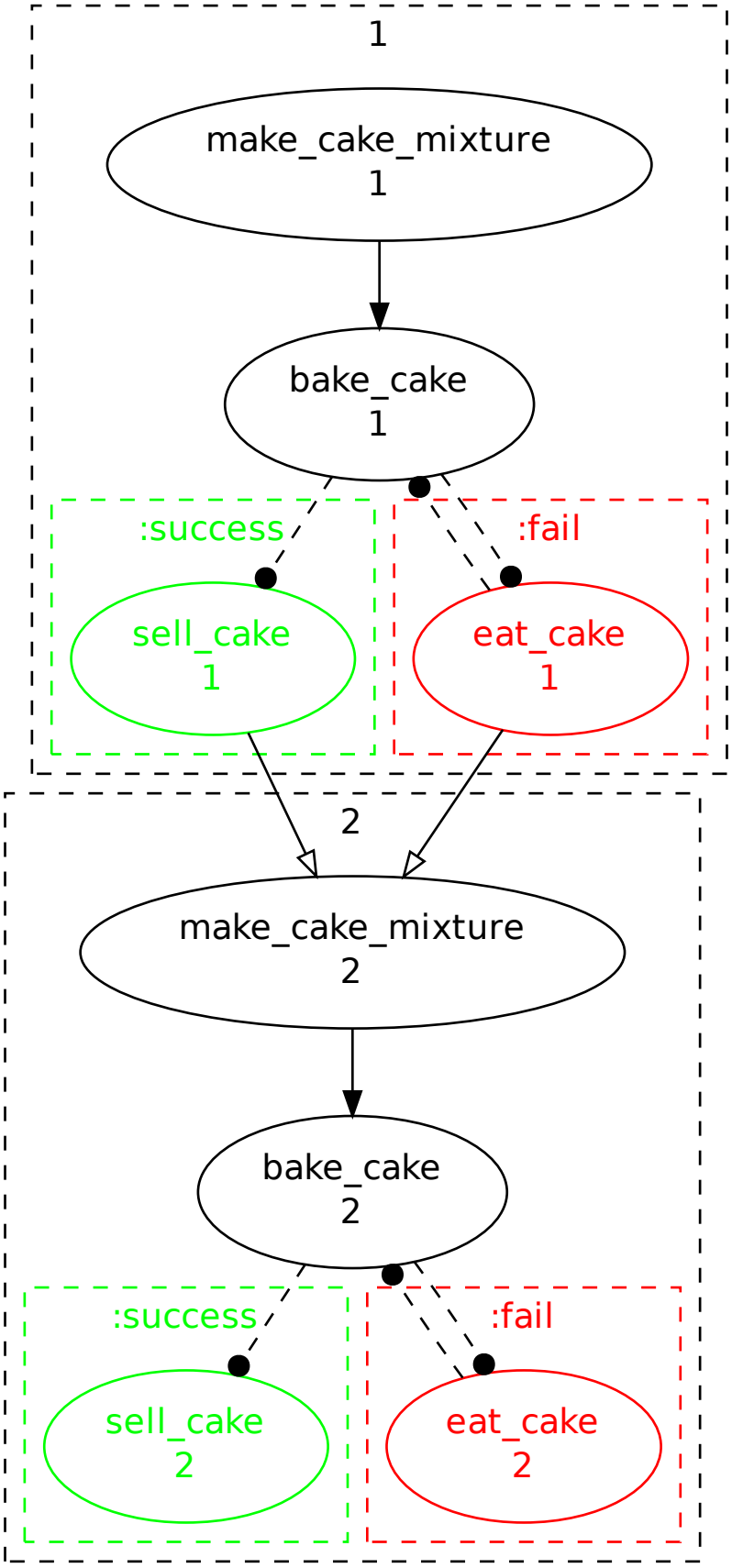
If we wanted to make the cycles run in order we might write an inter-cycle dependency like this:

```
sell_cake[-P1] => make_cake_mixture
```

In order to handle the event that the `sell_cake` task has been removed from the graph by a suicide trigger we can write our dependency with an or symbol `|` like so:

```
eat_cake[-P1] | sell_cake[-P1] => make_cake_mixture
```

Now the `make_cake_mixture` task from the next cycle will run after whichever of the `sell_cake` or `eat_cake` tasks is run.


1
make_cake_mixture
1
bake_cake
1
:success
:fail
sell_cake
1
eat_cake
1
2
make_cake_mixture
2
bake_cake
2
:success
:fail
sell_cake
2
eat_cake
2

Add the following graph string to your suite.

```
eat_cake[-P1] | sell_cake[-P1] => make_cake_mixture
```

Open the `cylc gui` and run the suite. You should see that if the `bake_cake` task fails both it and the `sell_cake` task disappear and are replaced by the `eat_cake` task.

4.7.6 Comparing "Regular" and "Suicide" Triggers

In Cylc "regular" and "suicide" triggers both work in the same way. For example the following graph lines implicitly combine using an `&` operator:

```
foo => pub
bar => pub
```

```
foo & bar => pub
```

Suicide triggers combine in the same way:

```
foo => !pub
bar => !pub
```

```
foo & bar => !pub
```

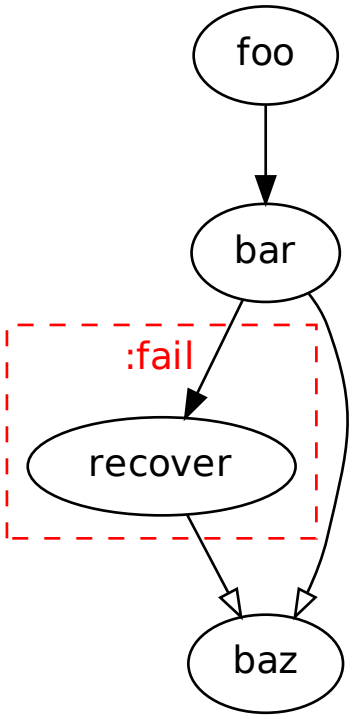
This means that suicide triggers are treated as "invisible tasks" rather than as "events". Suicide triggers can have pre-requisites just like a normal task.

4.7.7 Variations

The following sections outline examples of how to use suicide triggers.

Recovery Task

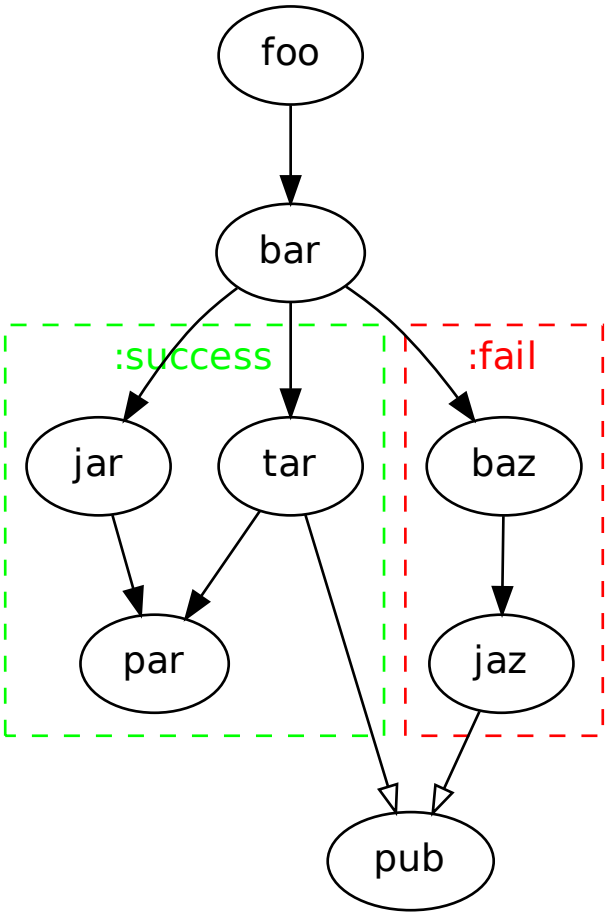
A common use case where a `recover` task is used to handle a task failure.




```
[scheduling]
    [[dependencies]]
        graph = """
            # Regular graph.
            foo => bar

            # The fail case.
            bar:fail => recover

            # Remove the "recover" task in the success case.
            bar => ! recover

            # Remove the "bar" task in the fail case.
            recover => ! bar

            # Downstream dependencies.
            bar | recover => baz
        """
[runtime]
    [[root]]
        script = sleep 1
    [[bar]]
        script = false
```

Branched Workflow

A workflow where sub-graphs of tasks are to be run in the success and or fail cases.




```
[scheduling]
    [[dependencies]]
        graph = """
            # Regular graph.
            foo => bar

            # Success case.
            bar => tar & jar

            # Fail case.
            bar:fail => baz => jaz

            # Remove tasks from the fail branch in the success case.
            bar => ! baz & ! jaz

            # Remove tasks from the success branch in the fail case.
            bar:fail => ! tar & ! jar & ! par

            # Remove the bar task in the fail case.
            baz => ! bar

            # Downstream dependencies.
            tar | jaz => pub
```

(continues on next page)

(continued from previous page)

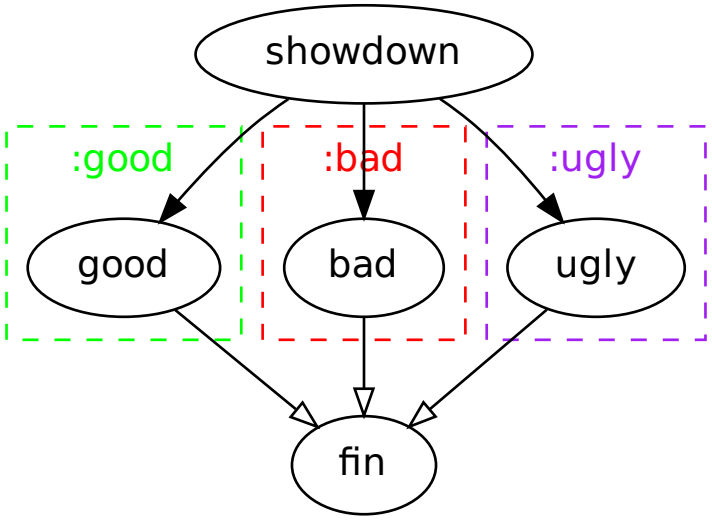
```
        """
[runtime]
    [[root]]
        script = sleep 1
    [[bar]]
        script = true
```

Triggering Based On Other States

In these examples we have been using suicide triggers to handle task failure. The suicide trigger mechanism works with other qualifiers as well for example:

```
foo:start => ! bar
```

Suicide triggers can also be used with custom outputs. In the following example the task `showdown` produces one of three possible custom outputs, `good`, `bad` or `ugly`.




```
[scheduling]
    [[dependencies]]
        graph = """
            # The "regular" dependencies
            showdown:good => good
            showdown:bad => bad
            showdown:ugly => ugly
            good | bad | ugly => fin

            # The "suicide" dependencies for each case
            showdown:good | showdown:bad => ! ugly
            showdown:bad | showdown:ugly => ! good
            showdown:ugly | showdown:good => ! bad
        """
[runtime]
    [[root]]
        script = sleep 1
```

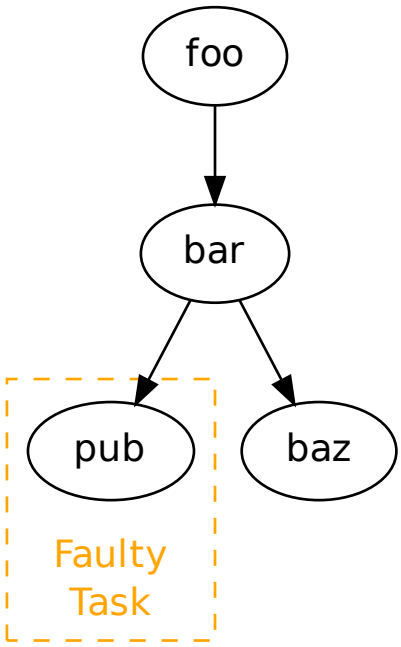
(continues on next page)

(continued from previous page)

```
    [[showdown]]
        # Randomly return one of the three custom outputs.
        script = """
            SEED=$RANDOM
            if ! (( $SEED % 3 )); then
                cylc message 'The-Good'
            elif ! (( ( $SEED + 1 ) % 3 )); then
                cylc message 'The-Bad'
            else
                cylc message 'The-Ugly'
            fi
        """
        [[[outputs]]]
            # Register the three custom outputs with cylc.
            good = 'The-Good'
            bad = 'The-Bad'
            ugly = 'The-Ugly'
```

Self-Suiciding Task

An example of a workflow where there are no tasks which are dependent on the task to suicide trigger.



It is possible for a task to suicide trigger itself e.g:

```
foo:fail => ! foo
```

Warning: This is usually not recommended but in the case where there are no tasks dependent on the one to remove it is an acceptable approach.

```
[scheduling]
    [[dependencies]]
        graph = """
            foo => bar => baz
            bar => pub

            # Remove the "pub" task in the event of failure.
            pub:fail => ! pub
        """
[runtime]
    [[root]]
        script = sleep 1
    [[pub]]
        script = false
```

/(str:prefix)

```
GET (str:prefix)/get_known_keys, ??
GET (str:prefix)/get_optional_keys, ??
GET (str:prefix)/get_query_operators,
      ??
GET (str:prefix)/query, ??
GET (str:prefix)/search, ??
```