



## a high level introduction

CSPP/IMAPP Users' Group Meeting  
SSEC, Madison, Wisconsin, June 27 2017

Adapted from original slides by Hilary J Oliver, [NIWA](#)

# what's cylc?

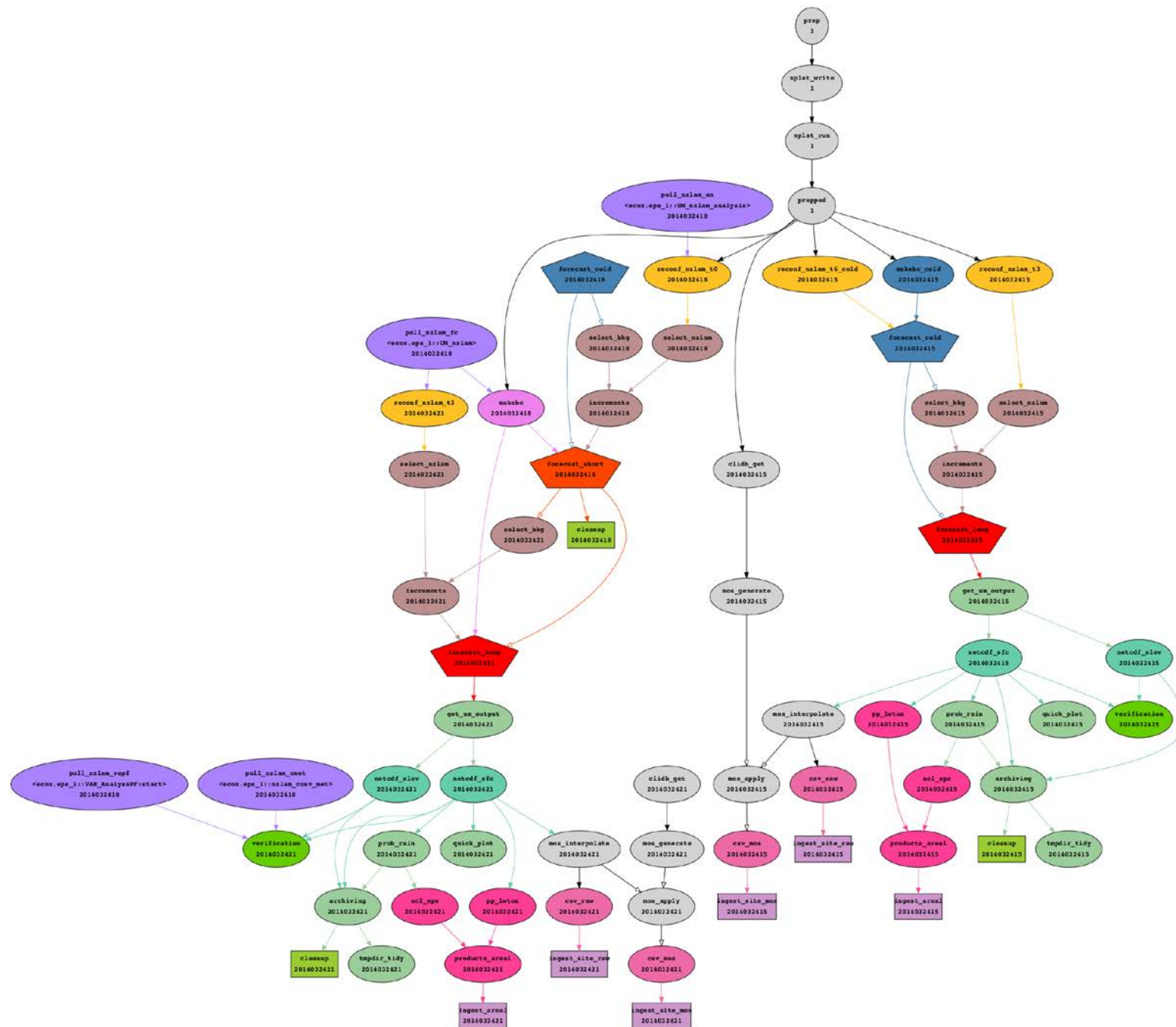
a workflow engine  
to construct **complex, cycling** workflows

<https://cylc.github.io/cylc>

# what's a workflow?

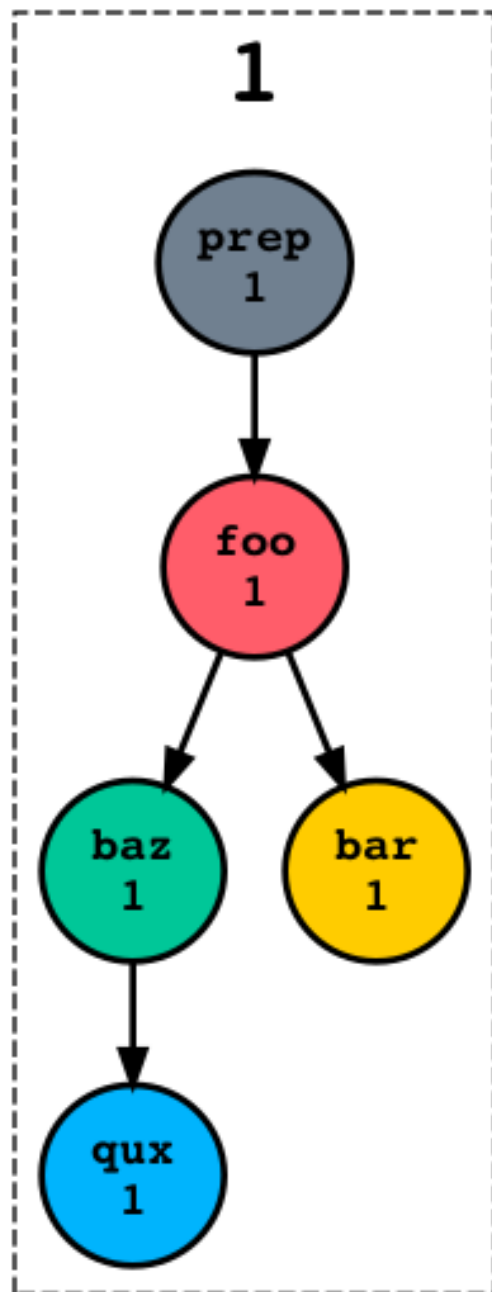
a group of tasks related by a dependency graph

cylc terminology: a suite



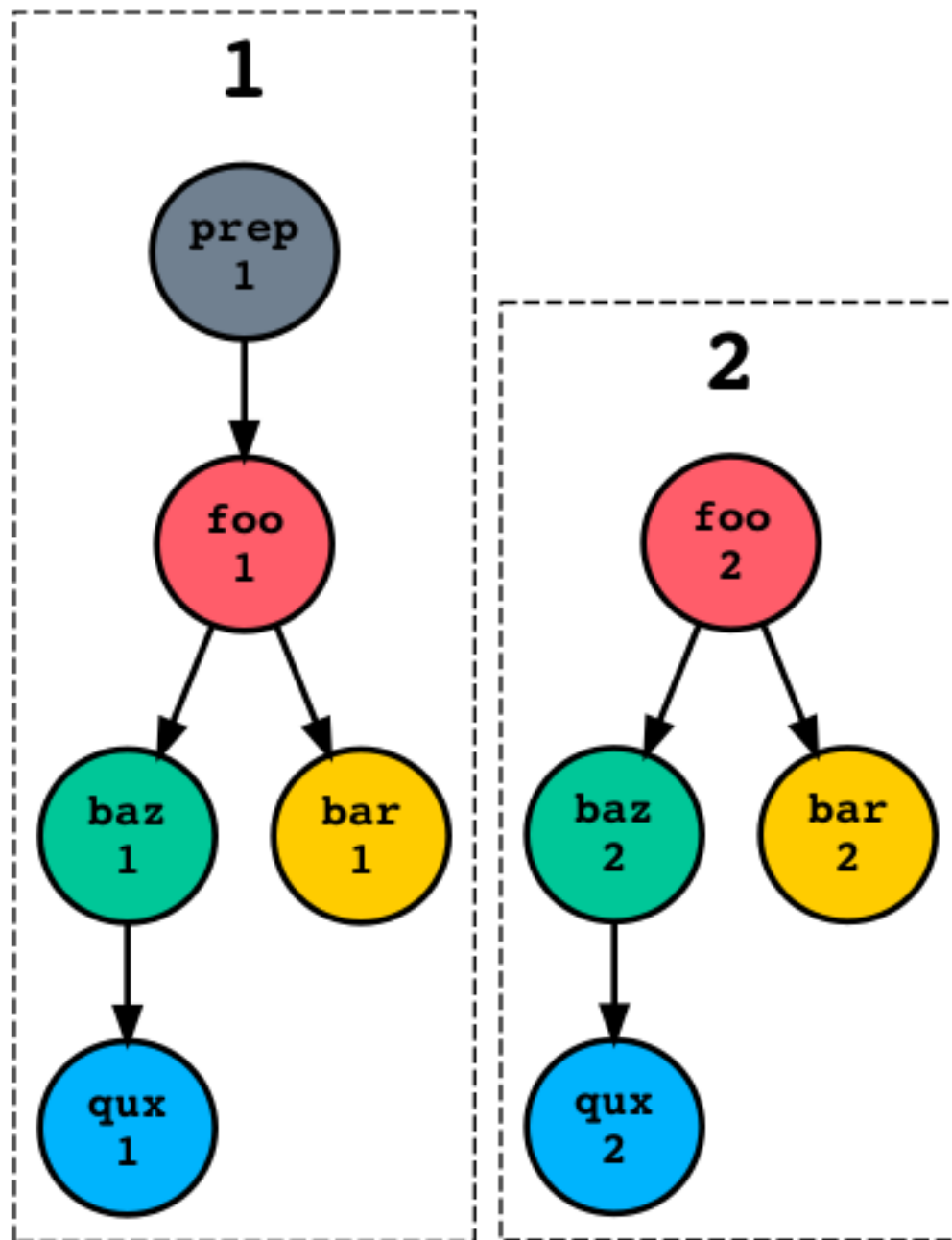
# what's a cycling workflow?

- **operational forecasting**: repeat (with variations) a workflow at intervals, when real-time data comes in
  - needs clock-triggers; and continue cycling indefinitely
- **forecasting research and testing**: run operational workflows (or variations thereof) over historical periods, off archived data
  - no clock-triggers, unless we catch up to the clock
- **to split a long model run into many short runs** (e.g. for long climate simulations) with associated processing for each chunk
  - no clock-triggers

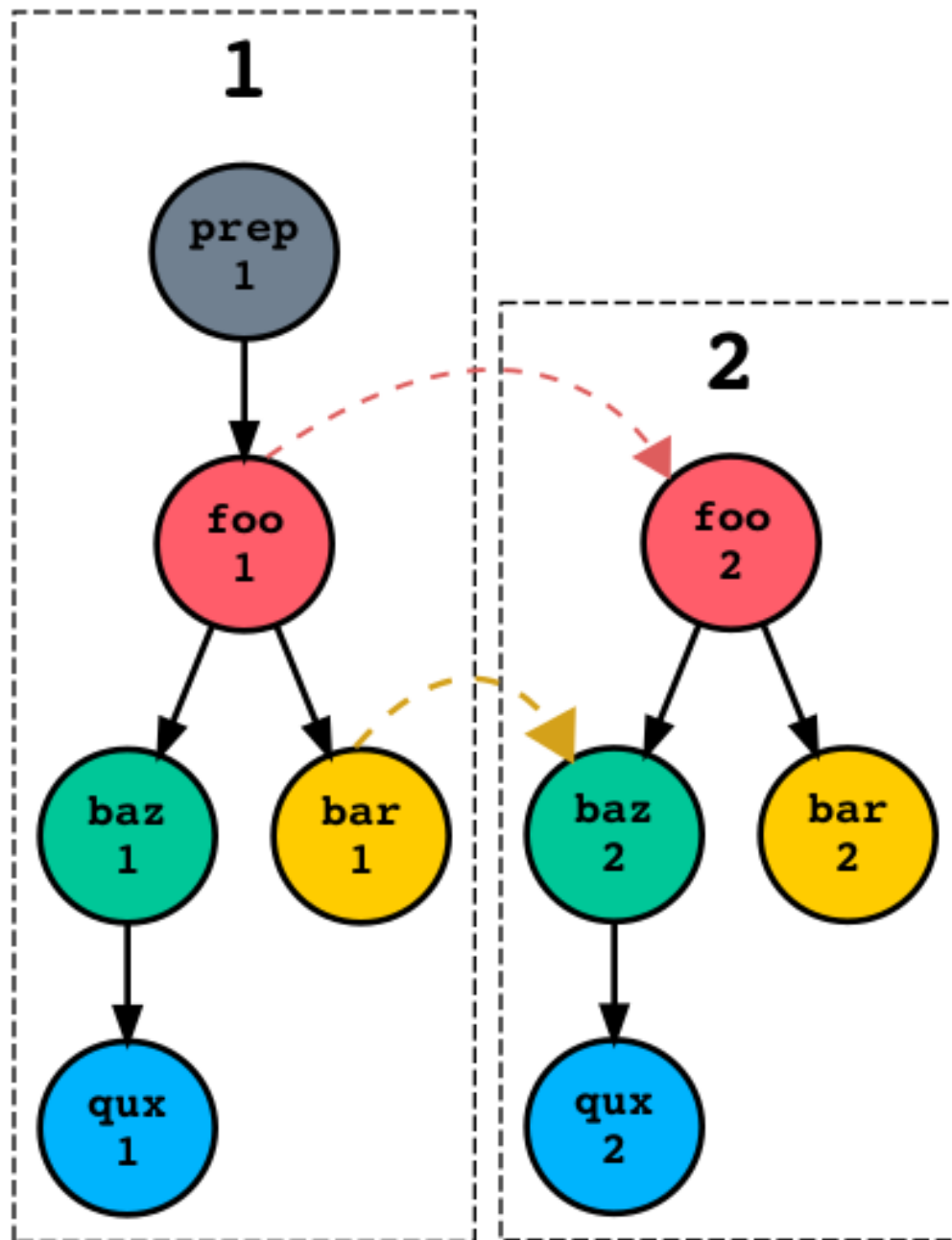


## a workflow **dependency graph**

- must be directed and acyclic (DAG)
- **nodes** represent **tasks** (which represent real jobs)
- **edges** represent **dependence** (typically input/output files)
- a **cycle point** is a particular point in sequence of date-time (or integer) points; this shows **cycle point 1**



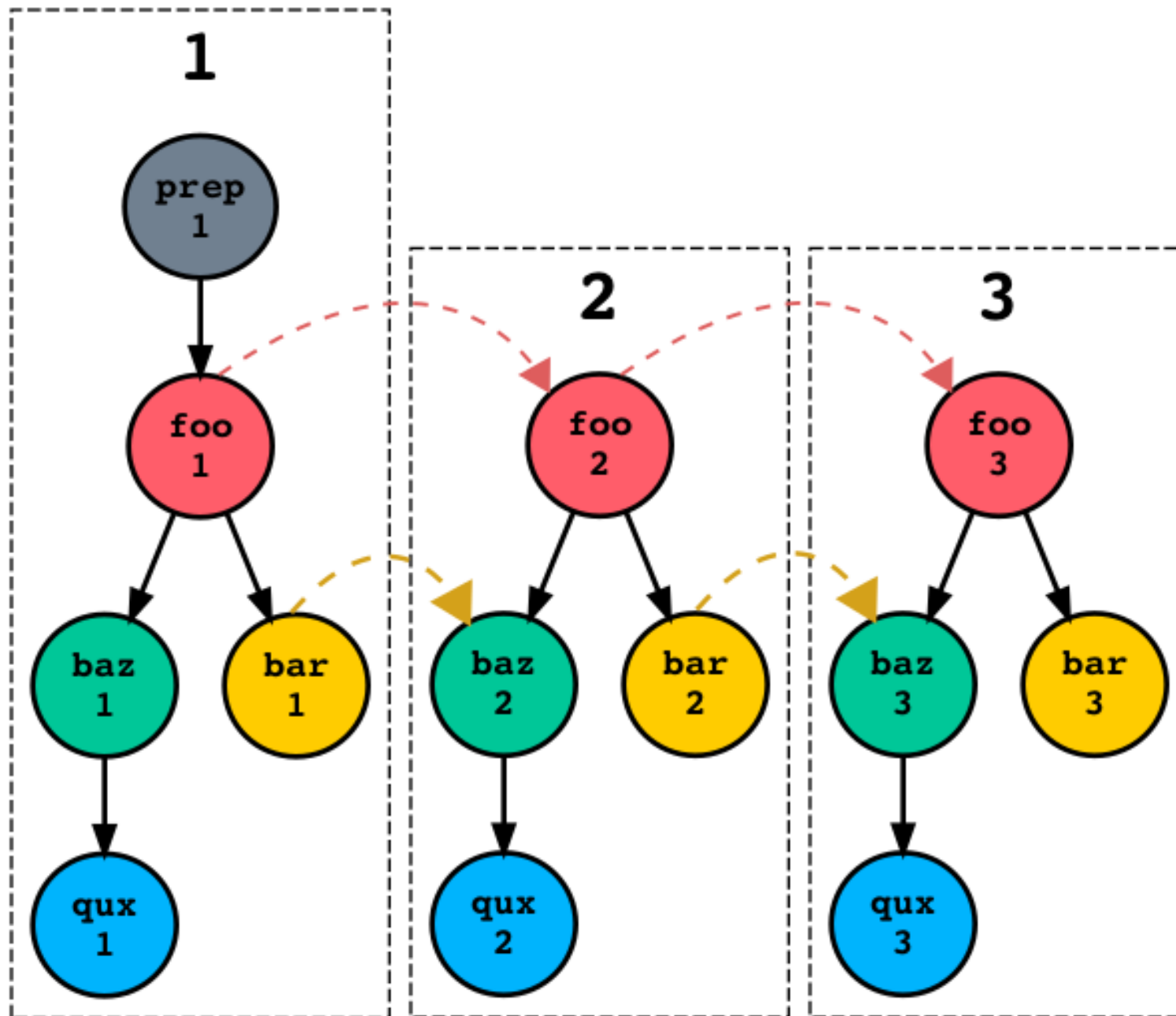
repeat, for the  
next forecast  
(cycle point 2)

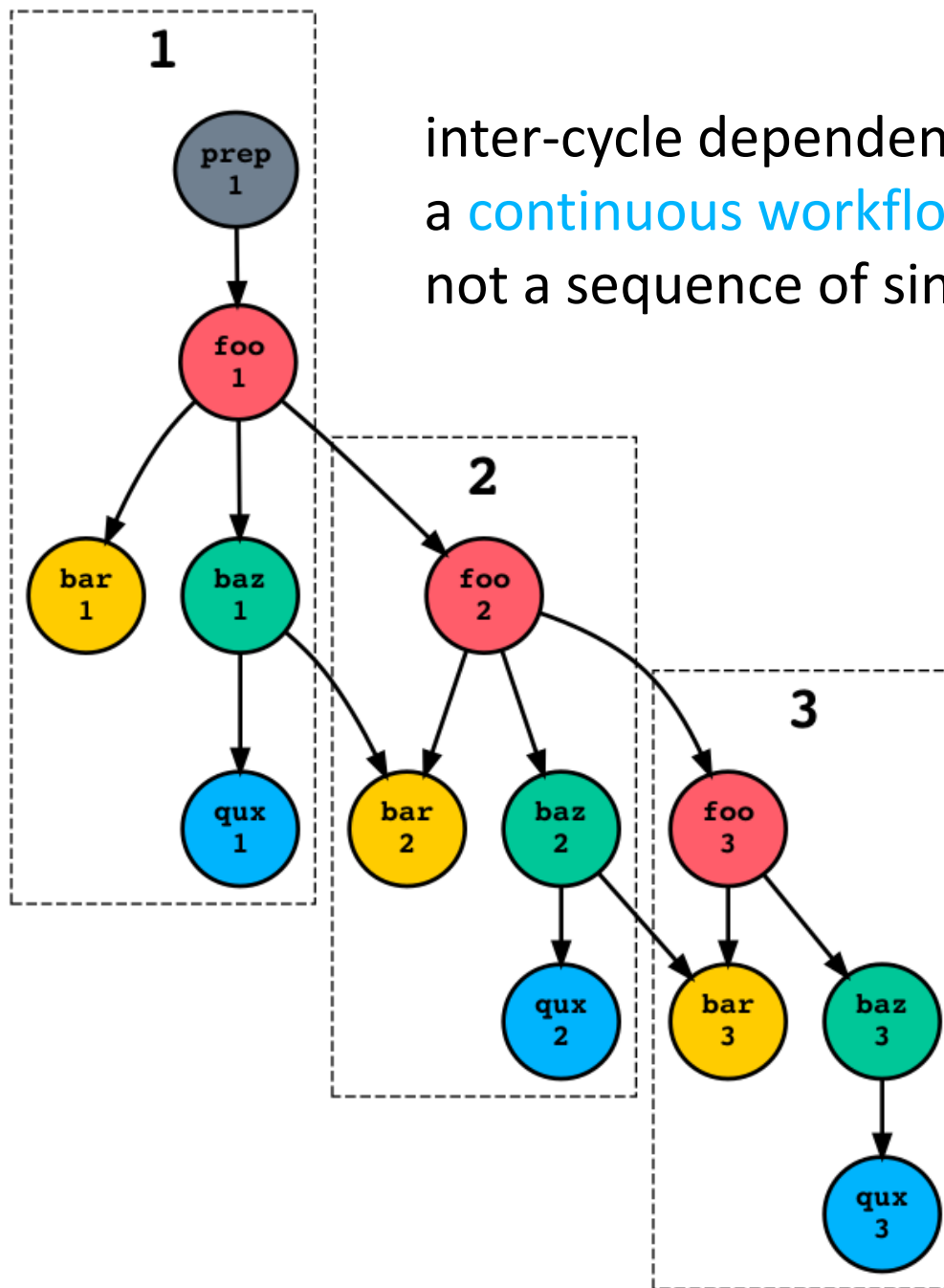


BUT NOTE there is **inter-cycle dependence** (e.g. model restart files)

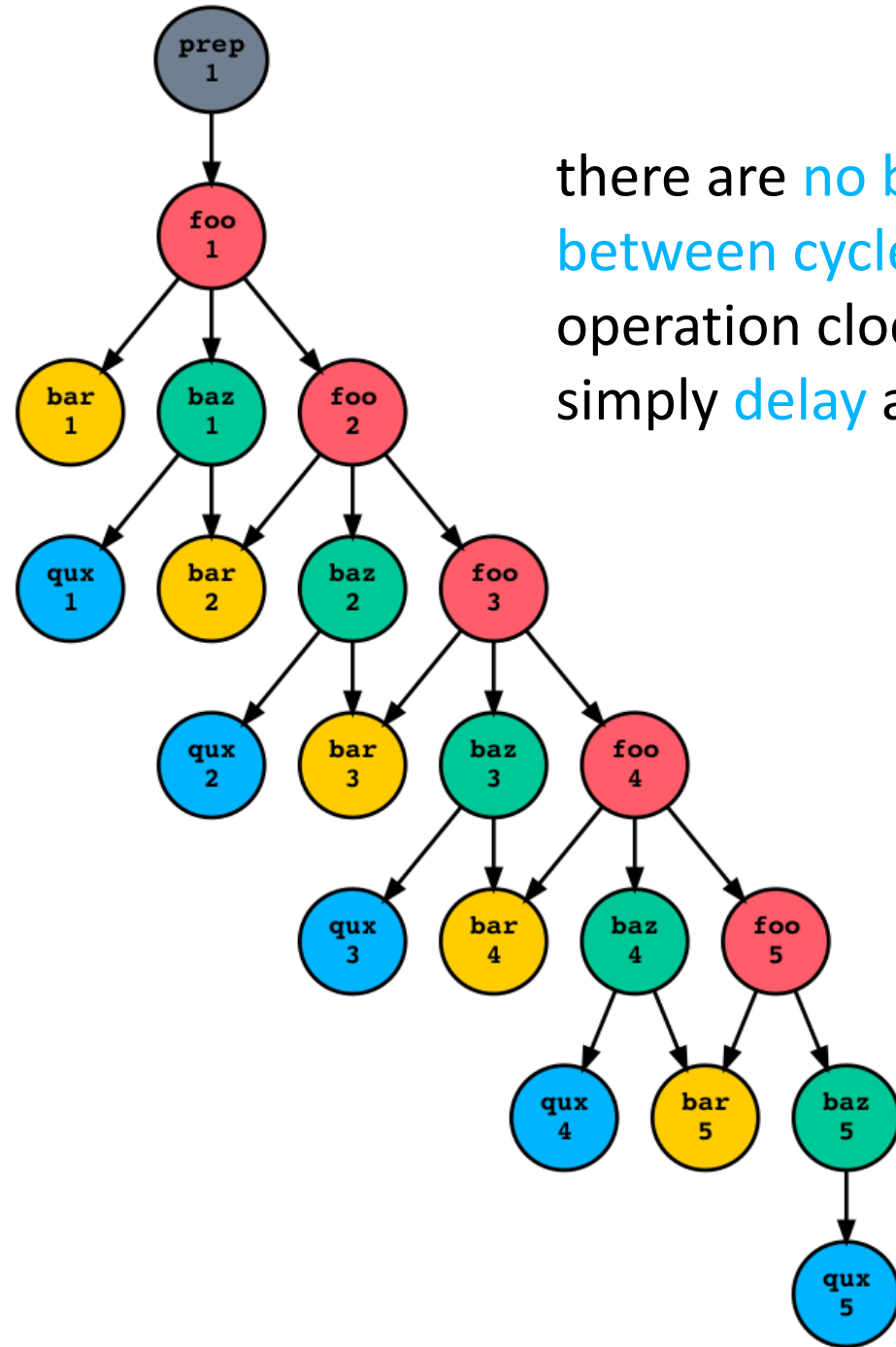
**foo[1] => foo[2]**  
**bar[1] => baz[2]**





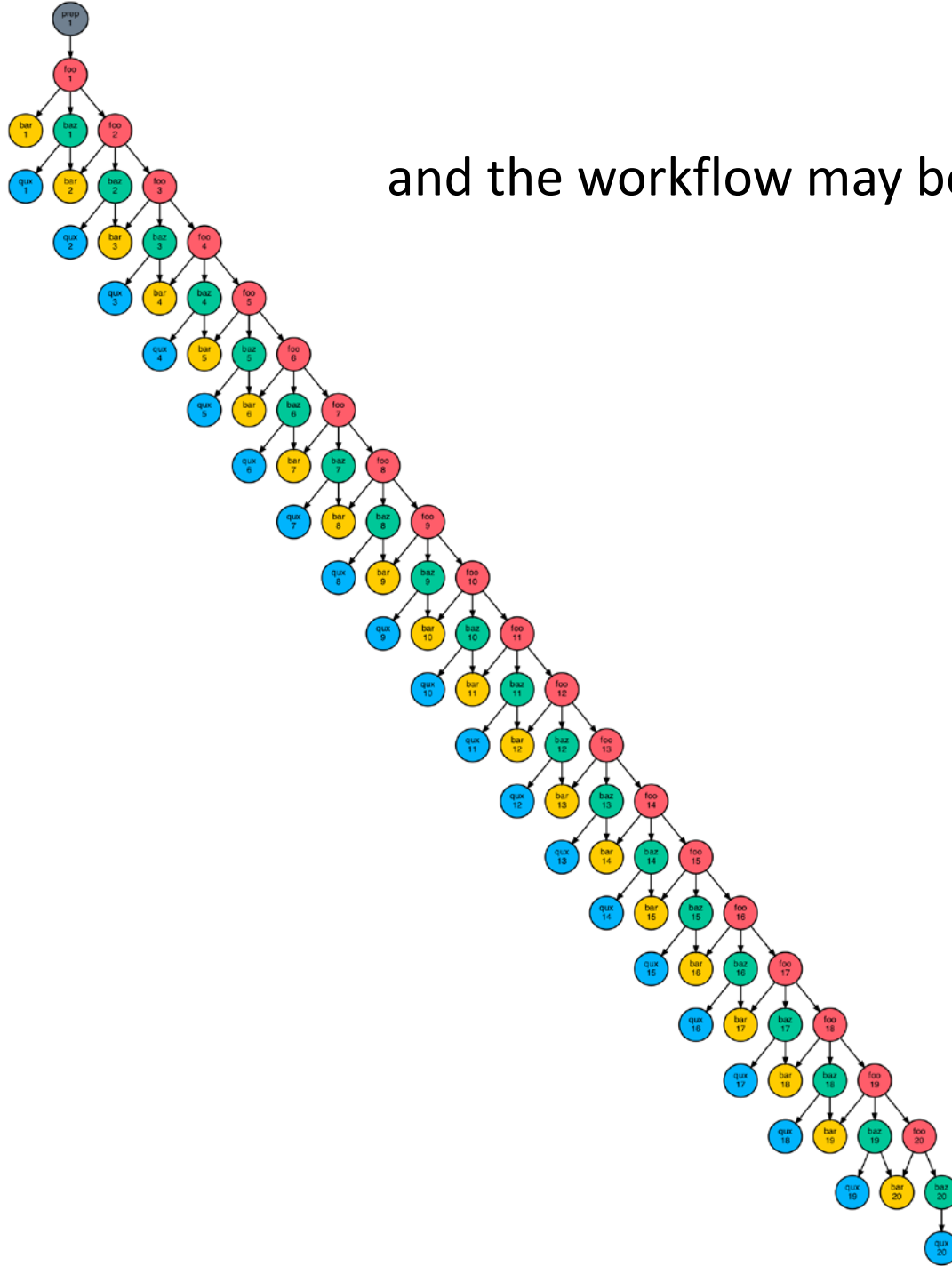


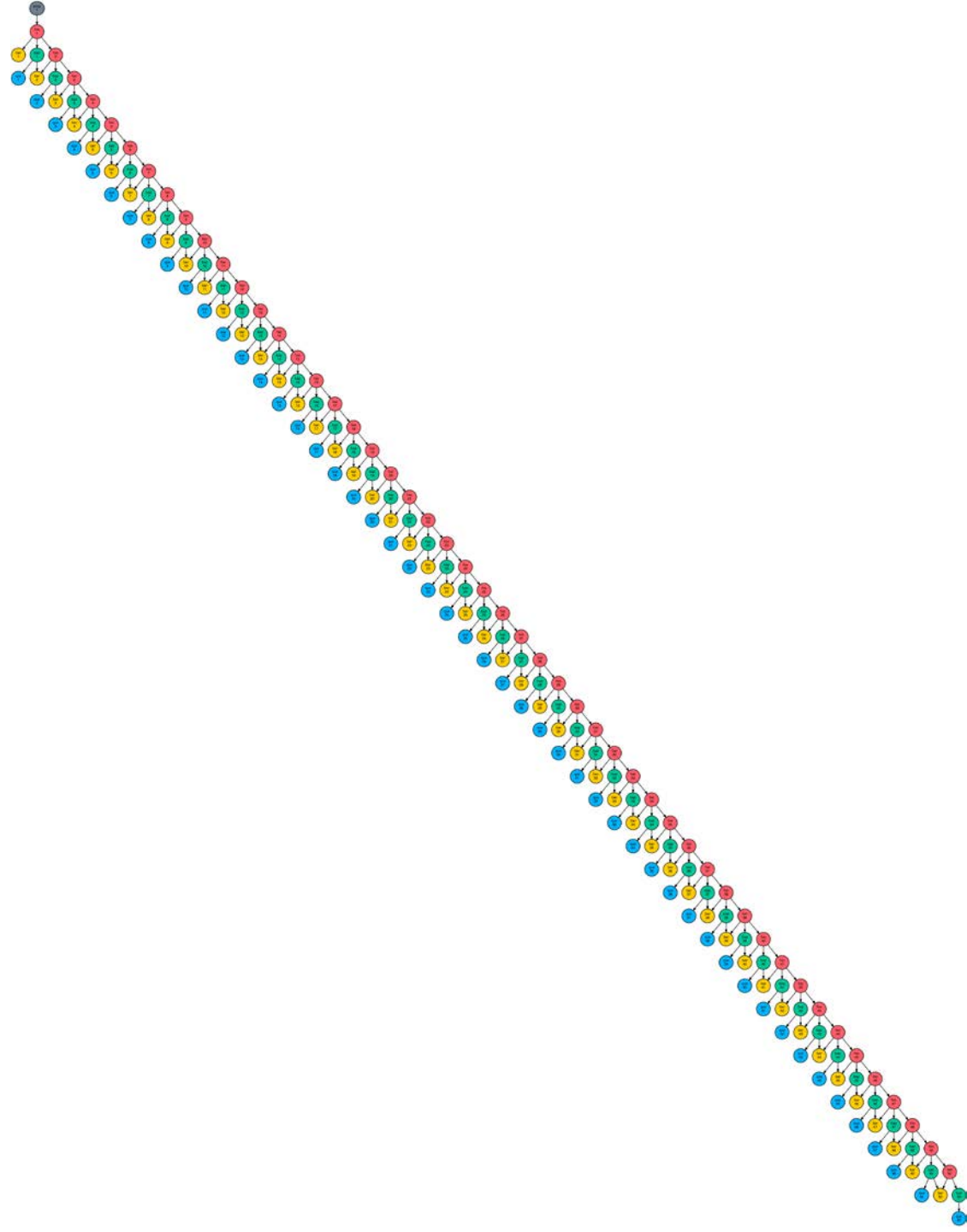
inter-cycle dependence makes this  
a **continuous workflow**  
not a sequence of single workflows

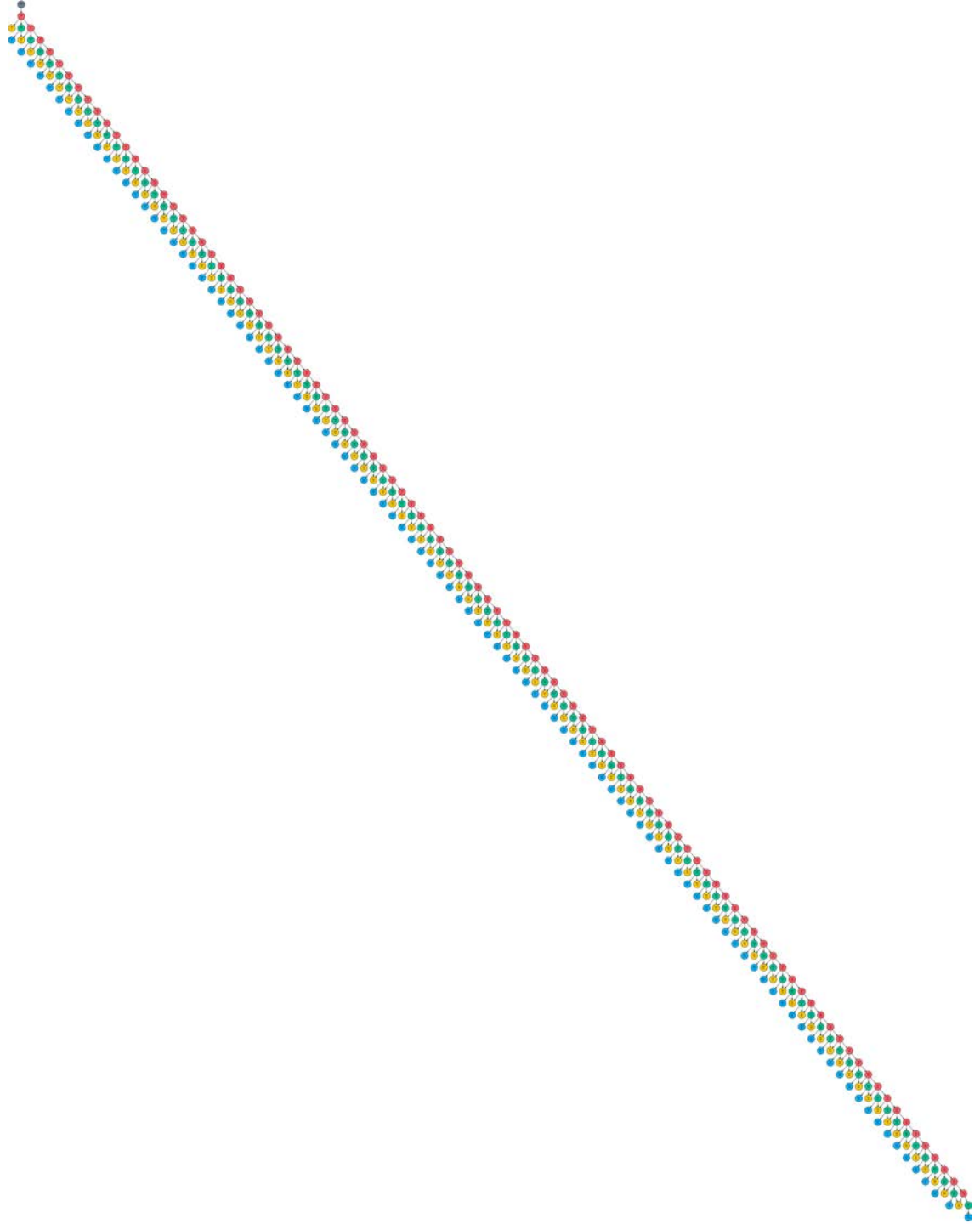


there are **no boundaries**  
**between cycles** (in real-time  
operation clock-triggers  
simply **delay** a few tasks)

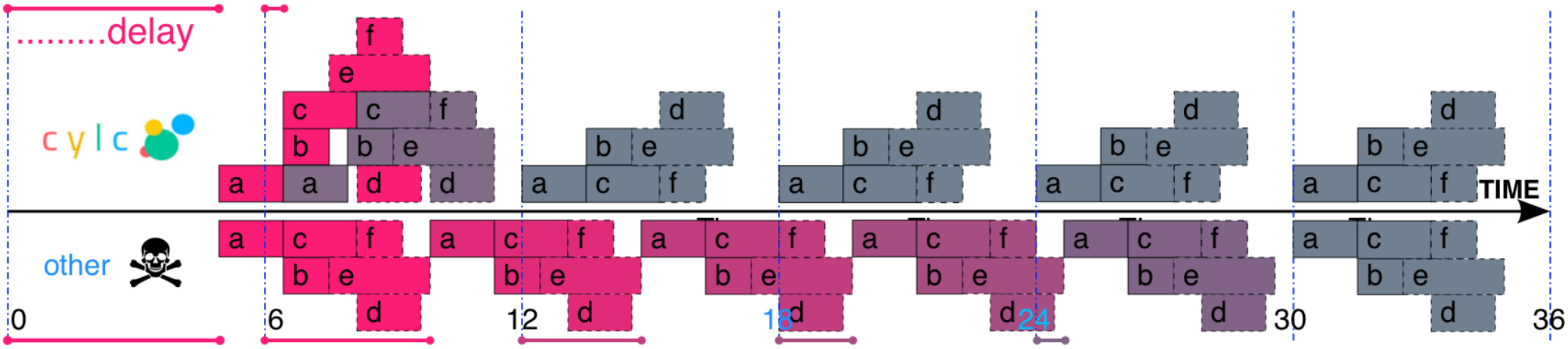
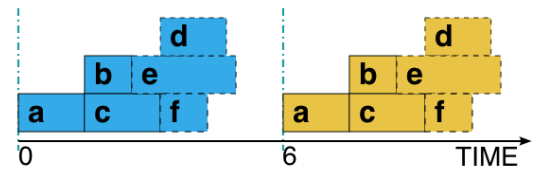
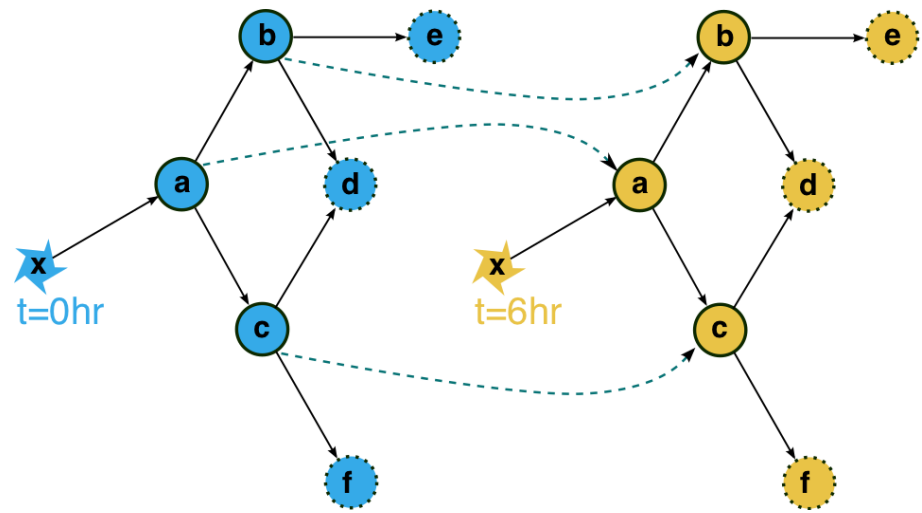
and the workflow may be unbounded...







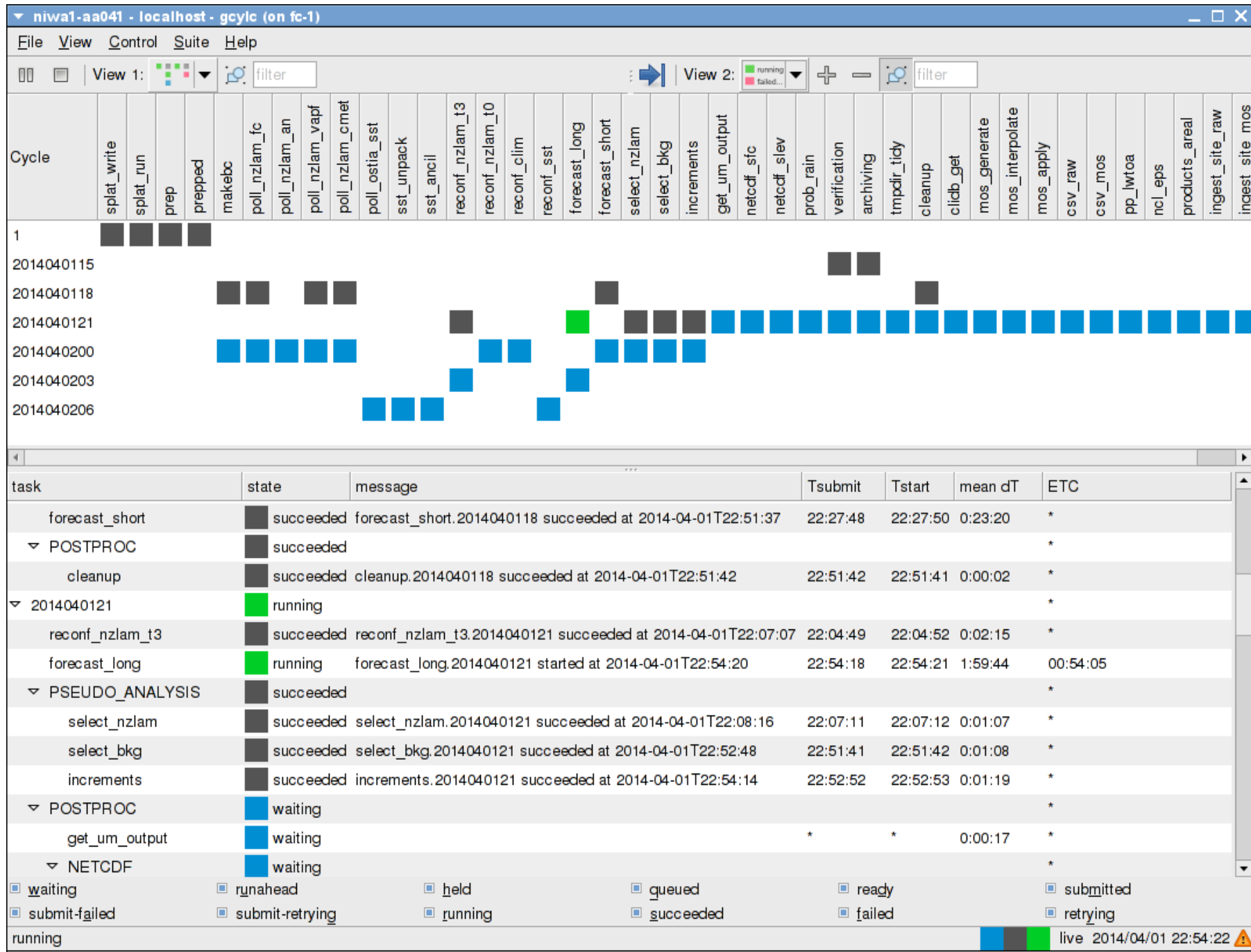
Note this cycle interleaving is particularly useful when real-time processing needs to catch up following a delay




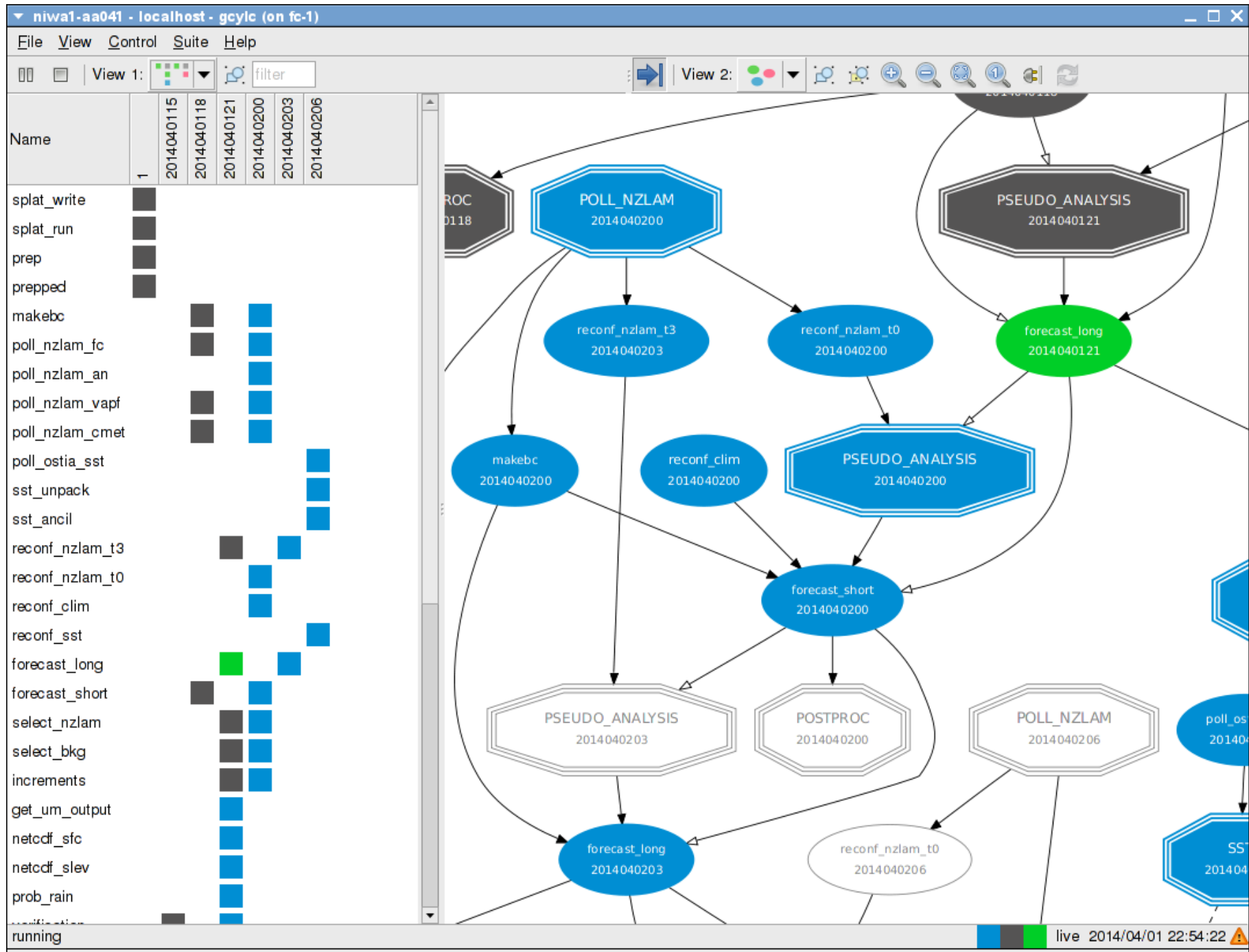
# what cylc does

- suite construction and visualization
- task meta-scheduling
- suite monitoring and control
- distributed suites
- adaptive scheduling
- cycling workflows
  - date-time and integer cycling
  - interleaves cycles for efficient scheduling
- (and a gazillion bells and whistles...)





live 2014/04/01 22:54:22 

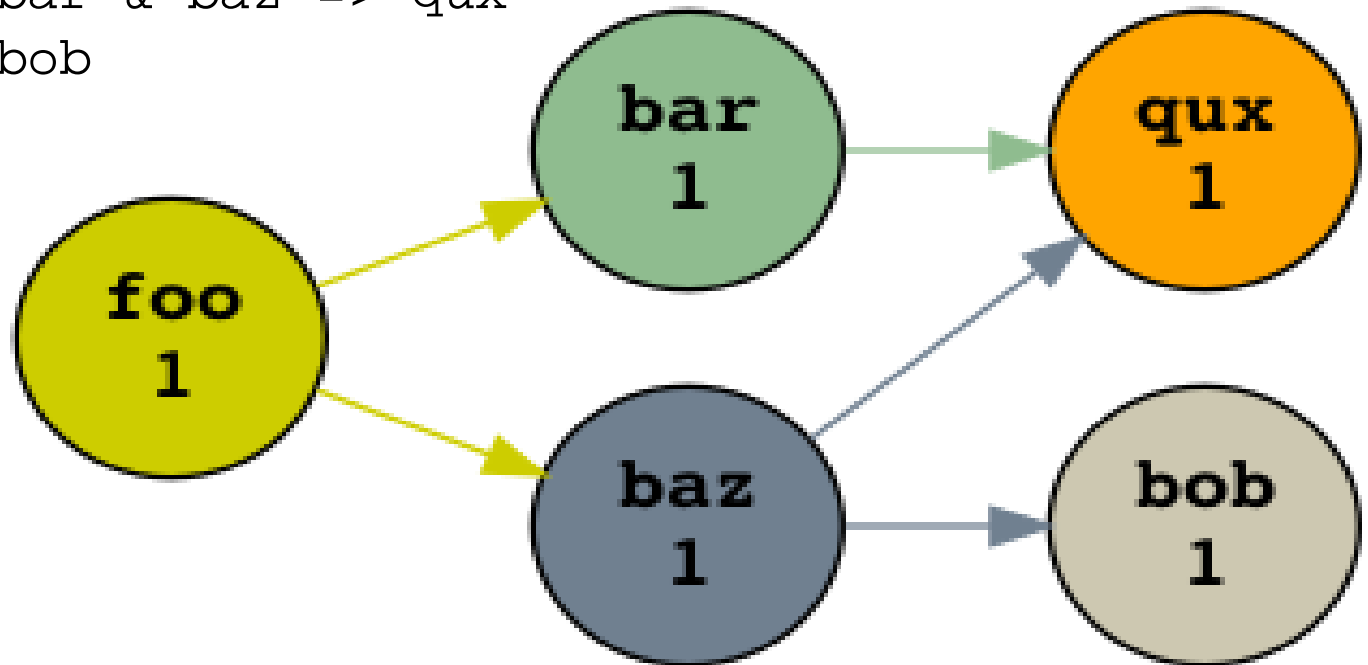


# suite.rc

```
# _____  
# FILE FORMAT: INI with [nested][[sections]].  
key = value  
# LEGAL CONTENT: see the cylc User Guide.  
#-----  
[cylc]  
    # Suite-level settings.  
[scheduling]  
    # Determines WHEN tasks can run.  
[runtime]  
    # Determines WHAT to run, WHERE, & HOW.  
[visualization]  
    # Styling the suite dependency graph.
```

# dependency graph notation

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



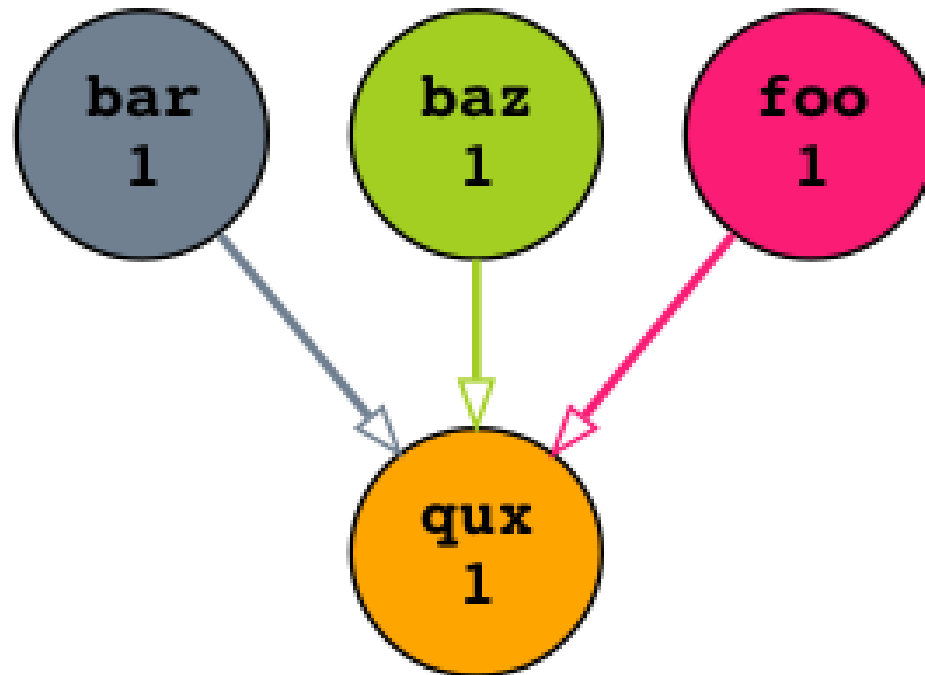
# conditional triggers

```
[scheduling]
```

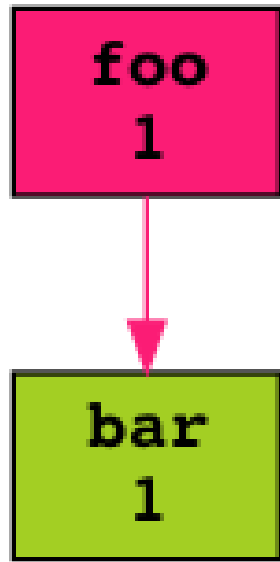
```
[[dependencies]]
```

```
graph = foo & (bar | baz) => qux
```

open arrow heads:



# task state triggering and suicide triggers

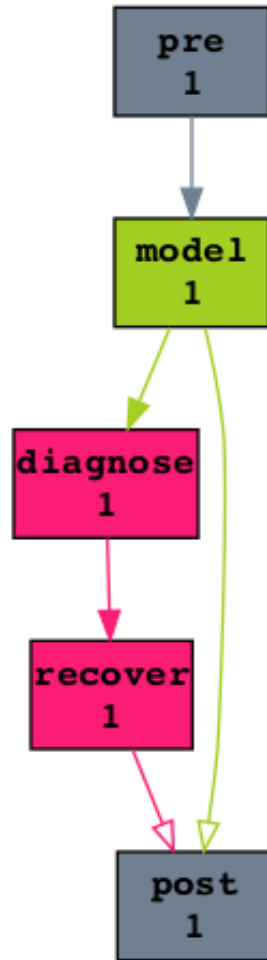


```
graph = foo:STATE => bar # trigger  
graph = foo:STATE => !bar # suicide
```

## STATES:

- foo # foo:succeed
- foo:submit
- foo:submit-fail
- foo:start
- foo:fail
- foo:finnish # :succeed OR :fail

# auto-recovery workflow



auto-recovery workflow:

```
[scheduling]
```

```
[[dependencies]]
```

```
graph = """
```

```
pre => model
```

```
model:fail => diagnose => recover
```

```
model => !diagnose & !recover
```

```
model:fail & post => !model
```

```
model | recover => post"""
```

e.g. diagnose - detect grid point storm failures;  
recover - run model with shorter timestep.

see also **automatic retry-on-failure**



# runtime: what to run

Hello World! In cylc:

```
#suite.rc
[scheduling]
    [[dependencies]]
        graph = greeter
[runtime]
    [[greeter]]
        script = "echo Hello World!"
```

The script can be any valid bash script; usually it would simply invoke an external script with appropriate parameters to perform the required task.

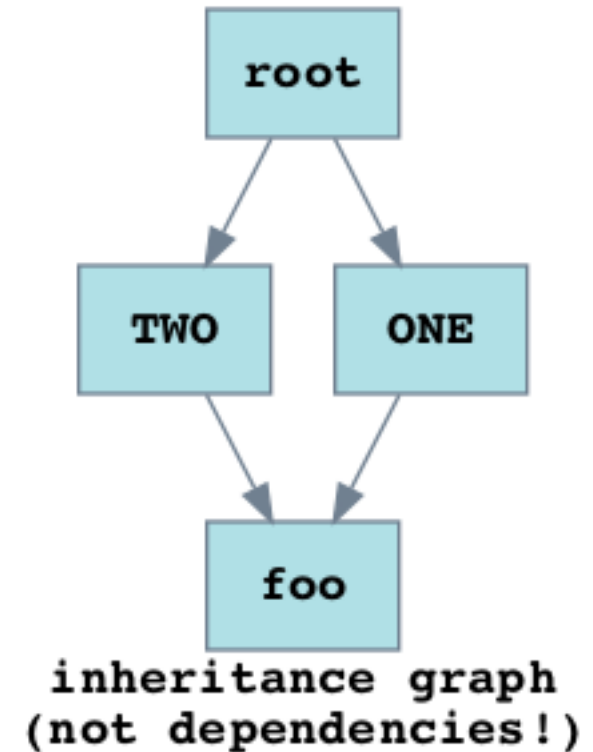
# runtime: where to run

## distributed suites:

```
[scheduling]
  [[dependencies]]
    graph = greeter_A => greeter_B
[runtime]
  [[root]]
    script = "echo Hello from ${HOSTNAME}!"
  [[greeter_A]]
    [[[remote]]]
      host = wrh-1.niwa.co.nz
  [[greeter_B]]
    [[[remote]]]
      host = wrh-2.niwa.co.nz
```

# runtime inheritance

```
[runtime]
  [[root]]      # family
    [[[environment]]]
      VAR0 = zero
  [[ONE]]       # family
    [[[environment]]]
      VAR1 = one
  [[TWO]]       # family
    [[[environment]]]
      VAR2 = two
  [[foo]]       # task
    inherit = ONE, TWO
```



# repeated sections

```
[scheduling]
  [[dependencies]]
    graph = ENSEMBLE
[runtime]
  [[ENSEMBLE]]
    title = "member xxx"
  [[m1,m2,m3,m4]]
    inherit = ENSEMBLE
  [[m2]]      # (extend or override)
title = "member two"
```

# runtime: when to run

- all date-times, durations and recurrences are specified using the ISO8601 Date-Time standard.
- Dates and times should be familiar to most:
  - hhmmss e.g. 061000 (ten past 6 in the morning).
  - YYYYMMDDThhmmss
  - YYYYMMDDThhmmssZ UTC
  - YYYYMMDDThhmmss+hhmm +ve time zone offset
  - YYYYMMDDThhmmss-hhmm -ve time zone offset
  - (same with +YYYYYY and -YYYYYY)

```
# cylc e.g.  
initial cycle point = 20140812T00Z
```

# ISO 8601 Durations

PnYnMnDTnHnMnS

- PT6H - 6 hours
- P1Y6M - 1 year and 6 months
- PT6M - 6 minutes
- P6M - 6 months
- P3W - 3 weeks

```
# cylc e.g.
```

```
[runtime]
```

```
[[long_forecast]]
```

```
# On failure, retry once after 1.5
```

```
# min, and then four times more at
```

```
# 10 min intervals:
```

```
retry delays = PT1.5M, 4*PT10M
```

# ISO 8601 Recurrences

- Rn/START\_TIME/PERIOD
- Rn/START\_TIME/END\_TIME
- Rn/PERIOD/END\_TIME
- Rn - repeat n times
- R - repeat indefinitely
- Can omit START, END or R[n] provided meaning unambiguous

```
# cylc e.g.  
# run 3 times with cycle times 20140812T00,  
# 20140812T06, 20140812T12  
[scheduling]  
  [[dependencies]]  
    [[[R3/20140812T00/PT6H]]]  
      graph = foo => bar
```

# date-time offsets

DATE\_TIME+PERIOD  
DATE\_TIME-PERIOD

- R/T06+P1D/P1D - repeat daily from a day after 0600 at or just beyond the initial cycle point
- R/+P3D/P2D - repeat two-daily starting three days after the initial cycle point

```
[scheduling]
  [[dependencies]]
    [[[P1Y]]]
      # an inter-cycle trigger offset:
      graph = foo[-P1Y] => foo
```

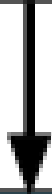


# Date-time cycling #0

run 6 hourly from initial date to final date, cycles run almost in parallel:

```
# suite.rc[cylc]
    cycle point format = CCYY-MM-DDThhZ
[scheduling]
    initial cycle point = 2014-08-01T00Z
    final cycle point = 2014-12-01T00Z
    [[dependencies]]
        [[[R1]]]
            graph = start => foo
        [[[R//PT6H]]]
            graph = foo => bar
```

start  
2014-08-01T00Z



foo  
2014-08-01T00Z



bar  
2014-08-01T00Z

foo  
2014-08-01T06Z



bar  
2014-08-01T06Z

foo  
2014-08-01T12Z



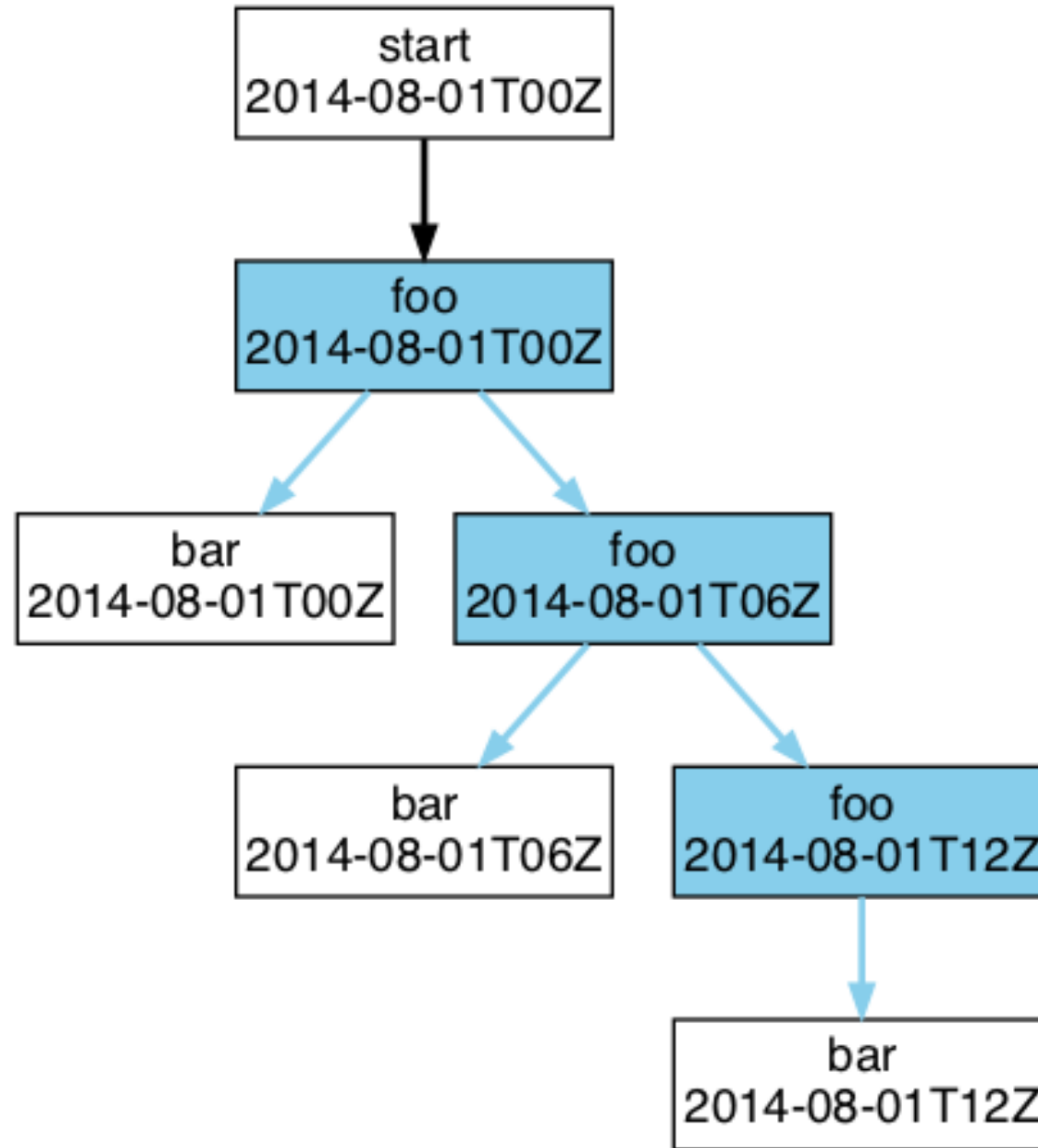
bar  
2014-08-01T12Z

cycling0

# Date-time cycling #1

Cycle N+1 cannot start until foo.N succeeds:

```
# suite.rc
[cylc]
    cycle point format = CCYY-MM-DDThhZ
[scheduling]
    initial cycle point = 2014-08-01T00Z
    final cycle point = 2014-12-01T00Z
    [[dependencies]]
        [[[R1]]]
            graph = start => foo
        [[[R//PT6H]]]
            graph = foo[-PT6H] => foo => bar
```

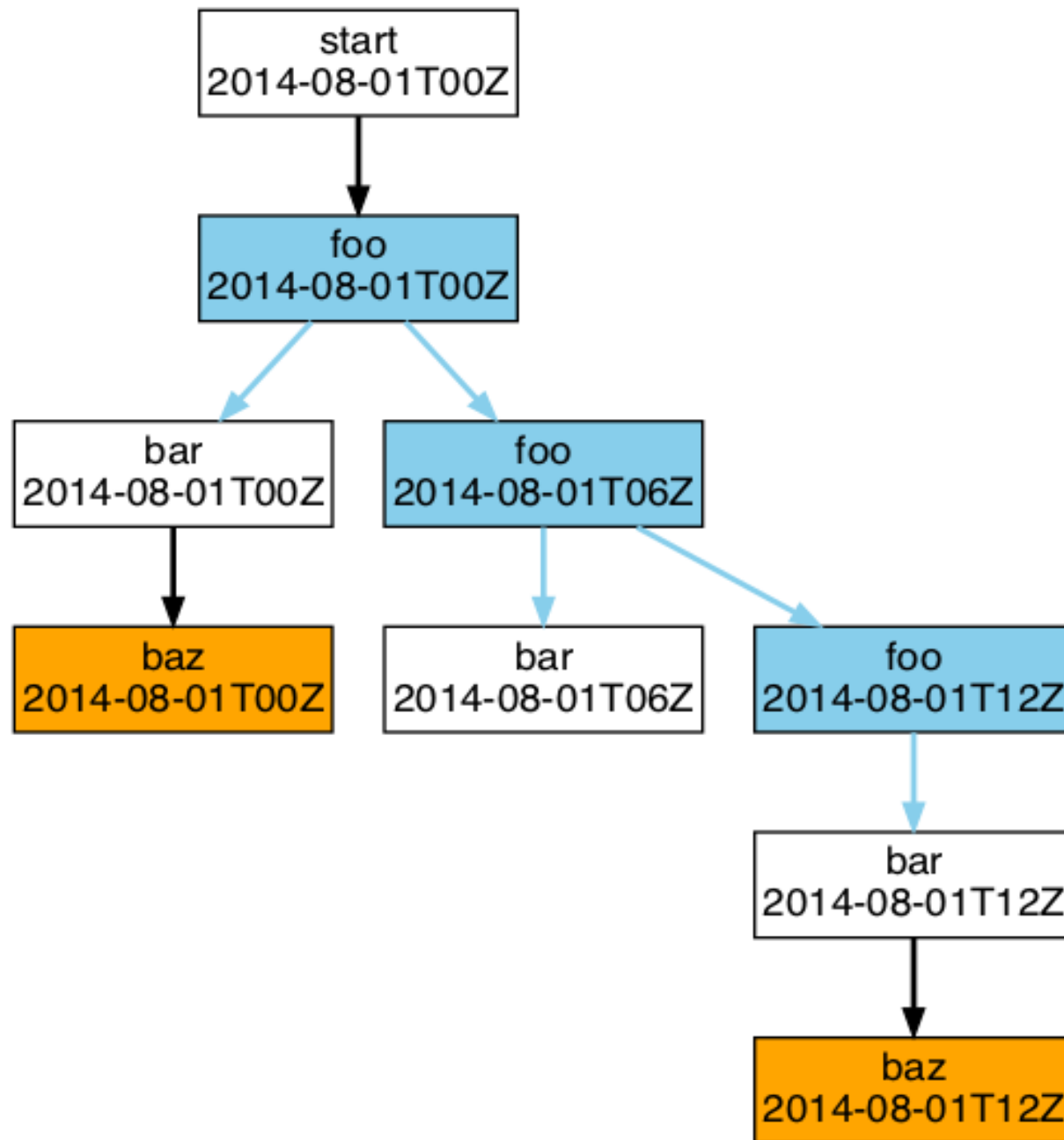


cycling1

# date-time cycling #2

- Add extra task 'baz' on every 2<sup>nd</sup> (12 hourly) cycle:

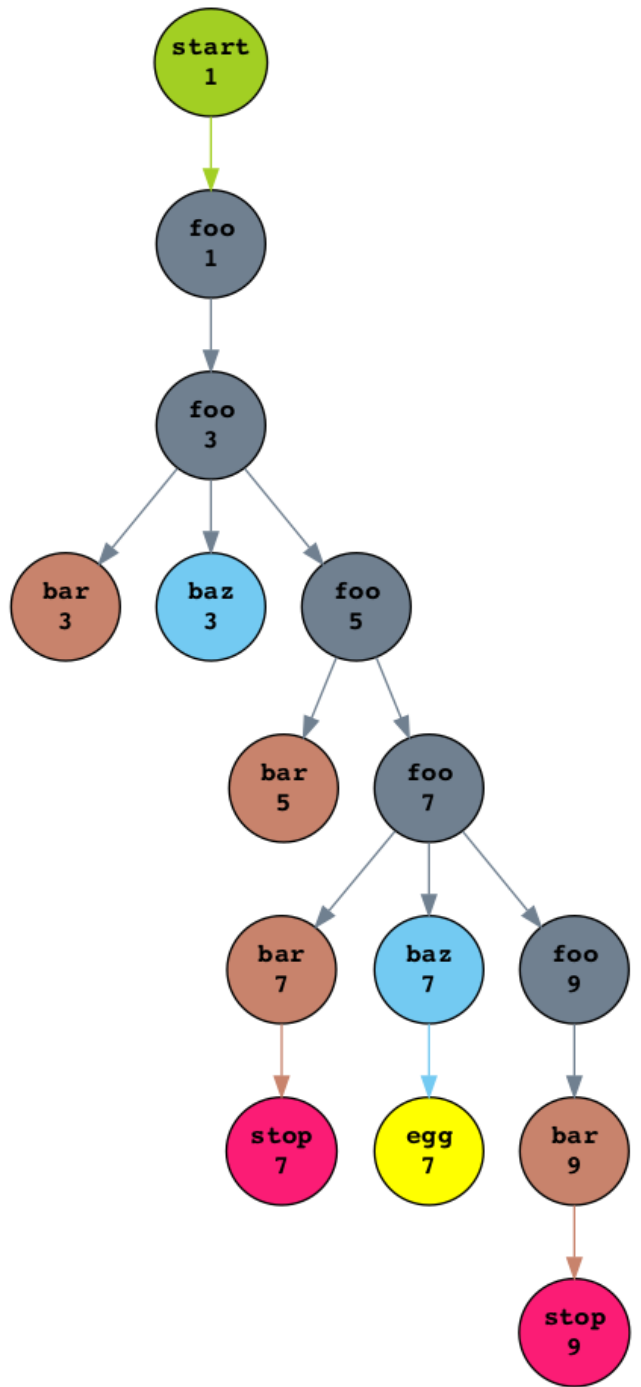
```
# suite.rc[cylc]
    cycle point format = CCYY-MM-DDThhZ
[scheduling]
    initial cycle point = 2014-08-01T00Z
    final cycle point = 2014-12-01T00Z
    [[dependencies]]
        [[[R1]]]
            graph = start => foo
        [[[R//PT6H]]]
            graph = foo[-PT6H] => foo => bar
        [[[R//PT12H]]]
            graph = bar => baz
```



cycling2

# integer cycling

- Almost identical except that instead of date-times, the cycle points are simply integer counters.
- Start and end times reduce to start and end count values
- Recurrence periods reduce to a cycle step value
- Offsets are just counted in cycles





# other features

- Run-ahead limiting
- Internal queues
- Broadcast messages to all tasks
- Nested subsuites
- Jinja2 scripting in the suite.rc file
- Introspection and self modification through e.g. issuing command line instructions from a running task to it's host suite

# who's using cylc?

- NIWA (NZ) \*
- Met Office (UK) \*
- Max-Planck-Institut für Meteorologie (DE)
- Deutsches Klimarechenzentrum (DE)
- Bureau of Meteorology (AU) \*
- NRL Marine Meteorology Division (US)
- 557th Weather Wing (US) \*
- Geophysical Fluid Dynamics Laboratory (US)
- Meteorological Service Singapore (SG) \*
- South African Weather Service (ZA) \*
- National Centre for Medium Range Weather Forecasting (IN) \*
- Korean Meteorological Administration (KR) \*
- National Center for Atmospheric Research - NCAR (US)

*\* used with Rose, a framework for managing meteorological suites.*