Workflow Automation for Cycling Systems

Oliver, Hilary; Shin, Matthew; Matthews, David; Sanders, Oliver; Bartholomew, Sadie; Clark, Andrew; Fitzpatrick, Ben; Haren, R. van; Hut, R.; Drost, Niels

DOI
10.1109/MCSE.2019.2906593

Publication date
2019

Document Version
Accepted author manuscript

Published in
Computing in Science & Engineering

Citation (APA)
Oliver, H., Shin, M., Matthews, D., Sanders, O., Bartholomew, S., Clark, A., Fitzpatrick, B., Haren, R. V., Hut, R., & Drost, N. (2019). Workflow Automation for Cycling Systems. *Computing in Science & Engineering*, *21*(4), 7-21. Article 8675433. https://doi.org/10.1109/MCSE.2019.2906593

THEME ARTICLE: INCORPORATING SCIENTIFIC WORKFLOWS IN COMPUTING RESEARCH PROCESSES.

Workflow Automation for Cycling Systems

The Cylc Workflow Engine

Hilary Oliver
NIWA, NZ

Matthew Shin
Met Office, UK

David Matthews
Met Office, UK

Oliver Sanders
Met Office, UK

Sadie Bartholomew
Met Office, UK

Andrew Clark
Met Office, UK

Ben Fitzpatrick
Met Office, UK

Ronald van Haren
Netherlands eScience Center

Rolf Hut
Delft University of Technology

Niels Drost
Netherlands eScience Center

Complex cycling workflows are fundamental to Numerical Weather Prediction (NWP) and related environmental forecasting systems. Large numbers of jobs are executed at regular intervals to process new data and generate new forecasts. Dependence between these *forecast cycles* creates a single never-ending workflow, but NWP workflow schedulers have traditionally ignored this—at the cost of efficiency when running "off the clock"—by enforcing a simpler non-overlapping sequence of single-cycle workflows. Cylc ("Silk") (1)(2)(3) is designed to manage infinite cycling workflows efficiently even after delays in real-time operation, or in historical runs, when cycles can typically interleave for much-increased throughput. Cylc is not actually specialized to environmental forecasting, however, and cycling workflows may also be useful in other contexts. In this article we describe the origins and major features of Cylc, future plans for the project, and our experience of Open Source development and community engagement.

Modern weather forecasting systems are based on large, complex cycling workflows of macroscopic applications, also known in the industry as *suites*. At regular intervals, ad infinitum, vast quantities of meteorological observation data are gathered and processed for assimilation by large atmospheric model ensembles (or increasingly, coupled ocean-atmosphere models), and screeds of output data are processed to generate forecast products. These workflows, or parts of them, typically run on large High Performance Computing (HPC/supercomputing) platforms to support the massive resource-hungry scientific models. Figure 1 shows a moderately-sized real weather forecasting workflow.

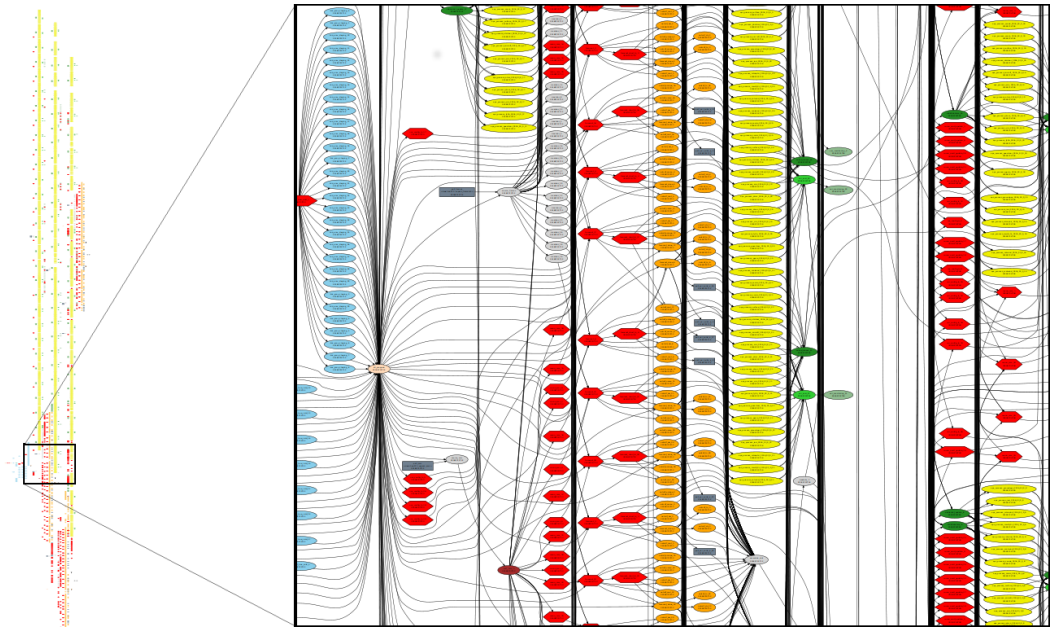

Figure 1. A small section of a 3-hourly cycling regional weather forecast ensemble implemented in Cylc. Each graph node represents an application (a script, program, or large scientific model) that executes on HPC nodes or other servers in the NIWA operations center. Other workflows, including global forecast ensembles and post-processing suites, can be much larger than this.

Cylc development began in 2008 in response to perceived deficiencies of existing NWP workflow schedulers. Interestingly, each run of a forecast model has to be partly initialized by a previous forecast because there is not enough information in the observations alone to determine the model state. This *warm cycling* of the model, and any other cross-cycle dependence, technically creates a single never-ending workflow rather than a never-ending sequence of single-cycle workflows. In normal clock-limited real-time operation this doesn't matter much because the workflows are designed to fit the available compute resource with room to spare, so that each cycle finishes well before the next batch of data is ready. But it can be a major impediment after delays that cause one cycle to run into another, or when processing historical data. Then, in principle, tasks from multiple cycles should be able to run concurrently (in practice it is never the case that the first task in one cycle depends on the last task in the previous cycle).

In the mid-late 2000s NIWA's new forecasting system ran into exactly this problem. Constrained to run whole cycles in sequence on an aging, over-subscribed supercomputer, it frequently took many hours to catch up from delays (see *NIWA Case Study* below). Sequential cycling was taken for granted by the leading production NWP workflow schedulers of the time such as SMS (Scheduler Monitoring System) from ECMWF (European Center for Medium-Range Weather Forecasts), to the extent that we could not find any published discussion of the problem. But expert users, and documentation, convinced us that this was a fundamental limitation. Further afield, it seemed that other systems could not help either, although it was difficult to be sure because the problem was simply not addressed in the literature. Some managed finite workflows with no cycles. Others such as Kepler (4) allowed loops over tasks or parts of a workflow. Expensive commercial systems did real-time scheduling of single workflows, but that is not the same as cycling either, at least not in the sense described here, and they were not used in HPC. The scientific tools also seemed more suited to research than production, by our requirements. So we elected to build a new workflow engine.

The primary design imperatives for Cylc were efficient scheduling of infinite cycling workflows without imposing an artificial barrier between cycles; compatibility with HPC platforms and workload managers like PBS; efficient workflow configuration for scientists and modelers rather than software developers, in a text-based format conducive to version control and collaborative development; to be light-weight and entirely application-agnostic in order to work with a vast and varied ecosystem of bespoke scientific software; and to be easy to use for researchers as well as NWP production centers. Cylc was written in Python and built around a new scheduling algorithm, described below, that can manage infinite workflows of cycling tasks without a sequential cycle loop. At any point during workflow execution it is only dependence between individual tasks that matters, regardless of their respective cycle points. As a result, Cylc can catch up from delays very quickly, and it can automatically and seamlessly transition between catch-up and clock-limited real-time operation as required. Off the clock, it can sustain interleaved cycling indefinitely. Figure 2 shows the dramatic effect this can have on job throughput.

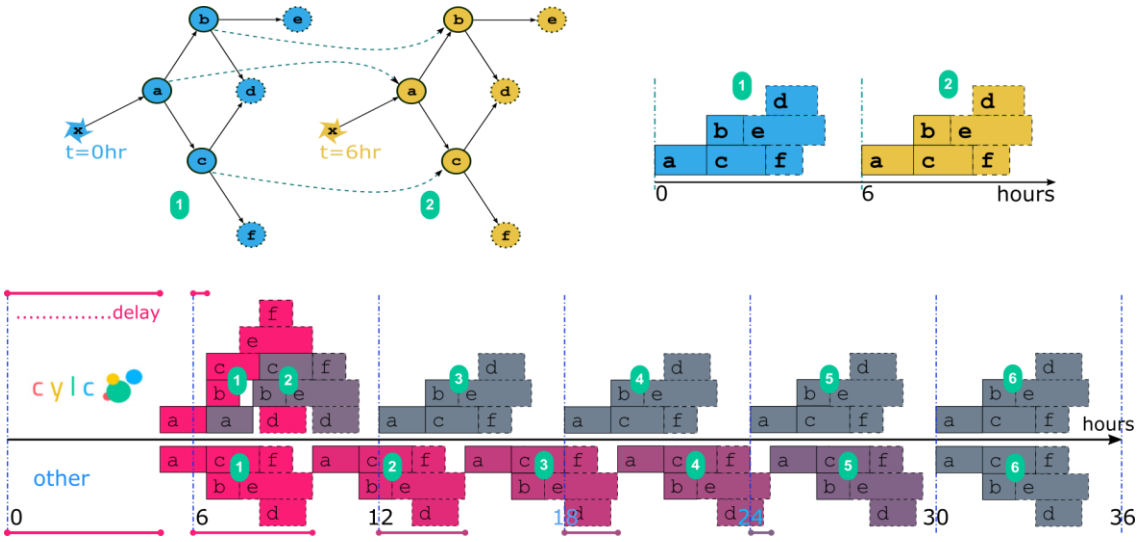

Figure 2: Efficient scheduling of a clock-limited real-time cycling workflow during catch-up from a delay. The upper left diagram shows two cycles of a highly simplified multi-model environmental forecasting workflow: sea-state (b) and storm-surge (c) models driven by a weather model (a), and product generation tasks (d, e, f). The star-shaped nodes are date-time clock triggers, and the dotted arrows show cross-cycle dependence (the models are initialized in part by their own previous forecasts). Vertical stacking in the upper right diagram shows concurrency during two consecutive cycles of the normal job schedule, for some assumed job run lengths in hours. The green badges label tasks with a common cycle point. In the lower diagram catch-up from a delay by sequential cycling is shown below the axis, and with optimal cycle-interleaving by Cylc above it. Gray coloring corresponds to normal "on the clock" operation, and redness to the size of the delay.

Ten years later, workflow management systems continue to proliferate; a prominent list on the internet records 238 of them (5). In the NWP arena SMS has been replaced by ecFlow (6), which handles cycling just like its predecessor. Whether or not others can do what Cylc does is something of a moot point now that Cylc is established at some major weather forecasting institutions (7) but we have not seen evidence that they can, or published solutions to the sequential cycling problem. The popular Apache Airflow, for example, can do repeat real-time scheduling of workflows (8). When deliberately put in "catchup" mode it disregards the schedule and runs them one after the other with no overlap—i.e. sequential cycling.

Since its release under an Open Source license (GNU GPL v3) Cylc has developed rapidly as a collaboration between NIWA, Met Office and the international Unified Model Partnership, ESiWACE (the Centre of Excellence in Simulation of Weather and Climate in Europe) (9), NRL (the US Naval Research Laboratory) (10), Altair Engineering (11), and others. Cylc has been widely adopted (7) for weather, climate, and environmental forecasting applications, although it is not technically specialized to these domains. Cycling workflows can also be useful elsewhere, e.g. for splitting long simulations into shorter chunks (common in climate and earth-system modeling); multi-run iterative statistical optimization of model parameters (12); processing many datasets with as much concurrency as possible; and even implementing classical pipelines (below).

HOW CYLC MANAGES CYCLING WORKFLOWS

A workflow can be represented as a Directed Acyclic Graph (DAG), with nodes as tasks and edges for dependence between them. If executed repeatedly with any cross-cycle dependence the result can be interpreted as a potentially infinite monolithic DAG composed of cycling tasks, as illustrated by Figure 3. Cylc decomposes a dependency graph like this at start-up to determine prerequisites and outputs for each task with respect to other tasks and relative to cycle point. For example, task A at a particular cycle point might depend on the automatic "*job succeeded*" output of task B and a custom "*file-1 completed*" output of task C, at the same cycle point. Proxy objects created to represent the first instance of each cycling task can submit their jobs to run when prerequisites are satisfied, and update their outputs in response to job status messages. The scheduling algorithm then matches unmet prerequisites with completed outputs to determine when tasks can run, advances the workflow by spawning new task proxies *individually* to subsequent cycle points, and removes spent tasks from the tail end of the workflow. If a task fails or is delayed by resource contention, others upstream of it can continue to advance as if nothing was wrong. Quick-running tasks can be held back by restricting the number of active cycle points if they are not otherwise constrained by clock triggers, external triggers, or dependence on other tasks. In this way Cylc manages an adaptive window that moves along the potentially infinite workflow graph.

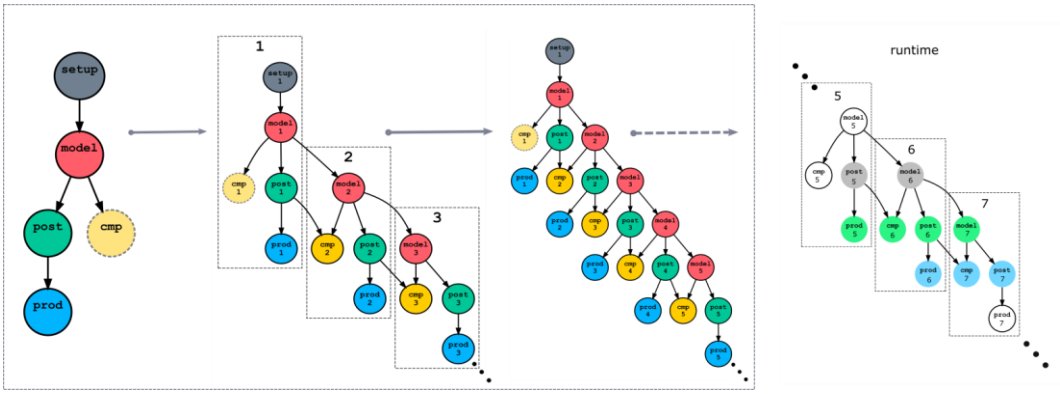

Figure 3. A toy infinite-cycling workflow. The **`model`** task represents a forecast model that depends on its own previous instance; **`post`** does model post-processing; **`prod`** computes forecast products; and **`cmp`** compares model output with some quantity derived from the previous forecast. On the left, more cycles are added, then the boxes drawn around them are removed to make it clear that this is really a never-ending monolithic graph. On the right, at runtime Cylc manages an adaptive window on the infinite workflow: solid green nodes represent running tasks, blue waiting (for their prerequisites to be satisfied), and gray succeeded. White nodes are not yet active (ahead) or are no longer needed (behind). Here, tasks from cycle points 5, 6, and 7 are running concurrently.

For completeness we note that *finite* workflows of this kind, if they are not too large, can be managed without dynamic cycling by parameterizing the entire run so that every job is in effect represented by a different logical task, rather than a new instance of a cycling task. Climate simulation experiments are handled in this way by Autosubmit (13), for example. Cylc can do this too, but the larger the workflow the more important dynamic cycling becomes. To run a workflow of 10 tasks per cycle for 100 cycles the workflow engine only has to be aware of 10 tasks at once in the dynamic case, or all 1000 of them if parameterized.

Workflow Configuration

Workflows can be defined by abstract dependencies (task B depends on task A); or by specifying the concrete inputs and outputs of each task. Recent authors have tended to favor the latter "data-modeling" approach (14) but each has its advantages. Abstract dependencies expose the structure of a workflow, they make it easy to trigger off of events and state changes, and artificial dependence can sometimes be useful; but dependency semantics are delegated to task configurations. Data dependencies allow workflows to dynamically self-assemble, and they make automatic data management possible; but other kinds of triggering may be harder to accommodate, and structure remains implicit in the workflow definition. Cylc workflows are internally self-assembling and early releases exposed this, but we now decompose dependency graphs to automatically define inputs and outputs for each task, as described above. Large ensemble post-processing suites have recently emerged, however, that need to radically reconfigure themselves according to the products selected, and these have led us to consider providing an alternative data-modeling interface in the future.

A workflow definition is primarily a configuration of the workflow engine, so Cylc uses a straightforward human-readable configuration file format for this purpose. The *suite definition* file can be validated against a specification, it encourages consistency of form, and it allows collaborative development of complex workflows with version control power tools like *git*. The base file format is augmented with task parameterization, inheritance of task runtime settings through a family hierarchy, Python-like Jinja2 or EmPy templating, and a cycling graph configuration language. For most users and most use cases this is easier than coding to an Application Programming Interface (API), but we do intend to provide a Python API as well for advanced use in the future.

In suite definition files, first a **`scheduling`** section determines *when* tasks run, with a dependency graph "drawn" in text form, clock triggers, external triggers, and internal queues. Then a **`runtime`** section dictates *what*, *where*, and *how* tasks run, including job command or scripting, environment, host, batch system and resource directives, automatic retry configuration, and event handling. And finally, graph styling can be configured in an optional **`visualization`** section (used to produce most of the images in this article).

In NWP suites like that of Figure 1 an ensemble of many atmospheric models is executed in each forecast cycle, each from different initial conditions consistent with current meteorological observations, in order to characterize uncertainty in the resulting forecasts. There is typically a large amount of common

configuration in such a system. For instance, closely-related ensemble members may have identical settings apart from a few member-specific details; tasks that share a job host will have technical settings in common; and there will be many shared file and IO workspace locations. To avoid ending up with a maintenance nightmare it is important to efficiently share rather than duplicate these common settings. Figure 4 illustrates one way to achieve economy of workflow definition in Cylc: a full multi-run multi-member ensemble suite is generated automatically by simply parameterizing a single-model base workflow over members and runs. Only the base tasks need to be explicitly configured in the suite definition, and the structure of the resulting workflow can still be understood at glance. Jinja2 or EmPy templating can also be used to define variables once for use throughout the workflow definition, and this can be pushed as far as programmatic generation of entire workflows using conditional expressions, loops, and so on. Finally, any runtime configuration can be shared, with no duplication at all, through a multiple inheritance hierarchy of *task families*. Templating and runtime inheritance can't be shown here due to space limitations, but they are heavily used in almost all Cylc suites.

```
[scheduling]
  [[dependencies]]
    graph = """init => fetch => model
      model => post => prod => clean"""
[runtime]
  [[init]]
    # ...
  [[fetch]]
    # ...
  [[model]]
    # ...
  [[post]]
    # ...
  # ...
```

```
[cylc]
  [[parameters]]
    r = a, b, c
    m = 1..3
[scheduling]
  [[dependencies]]
    graph = """
      init => fetch<r> => model<r,m> =>
      post<r,m> => prod<r> => clean"""
[runtime]
  [[init]]
    # ...
  [[fetch<r>]]
    # ...
  [[model<r,m>]]
    # ...
  [[post<r,m>]]
    # ...
  # ...
```

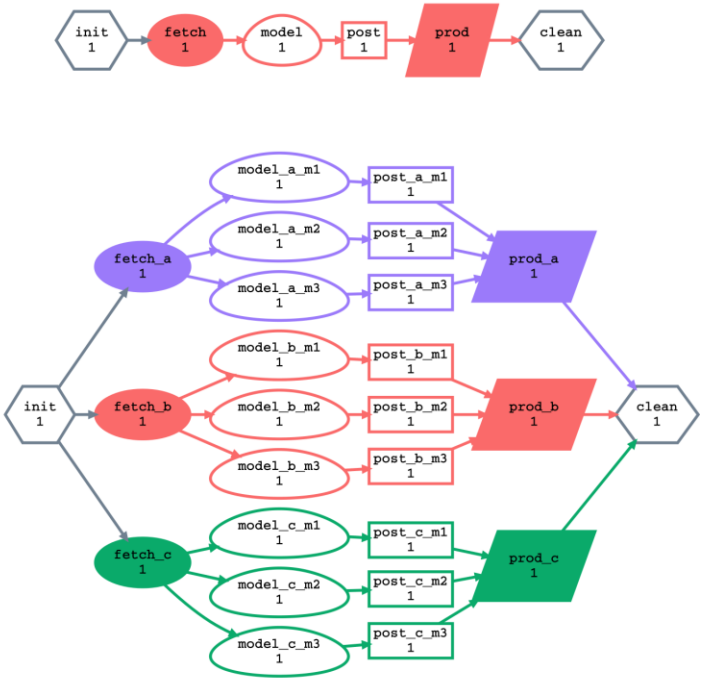

Figure 4. A non-cycling multi-run multi-member ensemble suite (bottom) generated by automatically replicating parts of a single-model workflow (top) over parameters **`r`** (runs) and **`m`** (members). The base tasks initialize a workspace (**`init`**), retrieve inputs (**`fetch`**), run a model (**`model`**), post-process outputs (**`post`**), generate products (**`prod`**), and tidy up (**`clean`**). Arrow symbols in the **`graph`** strings represent dependence between tasks: "**`init => fetch`**" means, by default triggering semantics, that **`fetch`** can trigger once **`init`** has successfully completed. Task configuration, represented here by placeholders under the **`runtime`** section, is described in the main text. Cylc passes parameter values to the jobs of parameterized tasks to allow appropriate specialization, e.g. for **`model_c_m2`** to identify its *run-c* and *member-2* specific inputs. Real ensemble suites typically have many more tasks and a cycling workflow.

For cycling systems, multiple graph segments associated with different bounded (or unbounded) cycling sequences combine, with an offset notation for dependence across cycles and between sequences, to make a pattern for generating concrete graphs over any given range of cycle points. Figure 5 illustrates this with a classical pipeline implemented by integer cycling, and Figure 6 shows a small date-time cycling workflow with special behaviour at several cycle points. Both examples rely on Cylc automatically ignoring dependence on tasks prior to the initial cycle point, for convenience, but the exact start-up dependencies can also be written down. Cycling sequence configuration is based on ISO 8601 standard date-time *recurrence expressions*, with some abbreviations and additions allowed in the Cylc context, and analogous simplified expressions for integer cycling. Of several recurrence forms, the most commonly used is ***`R[n]/<start-point>/P<interval>`*** where ***`n`*** is an optional limit on the number of cycles. So for example **`R/^/P1`** defines a never-ending sequence with an integer interval of one, beginning at the suite initial cycle point **`^`**. Similarly, ***`R5/^+PT6H/PT6H`*** defines a five-cycle bounded sequence with a six-hour interval, from six hours past the initial cycle point. Date-time arithmetic, with time zones and several special calendars for climate applications, is supported by our custom ISO 8601 date-time library (15).

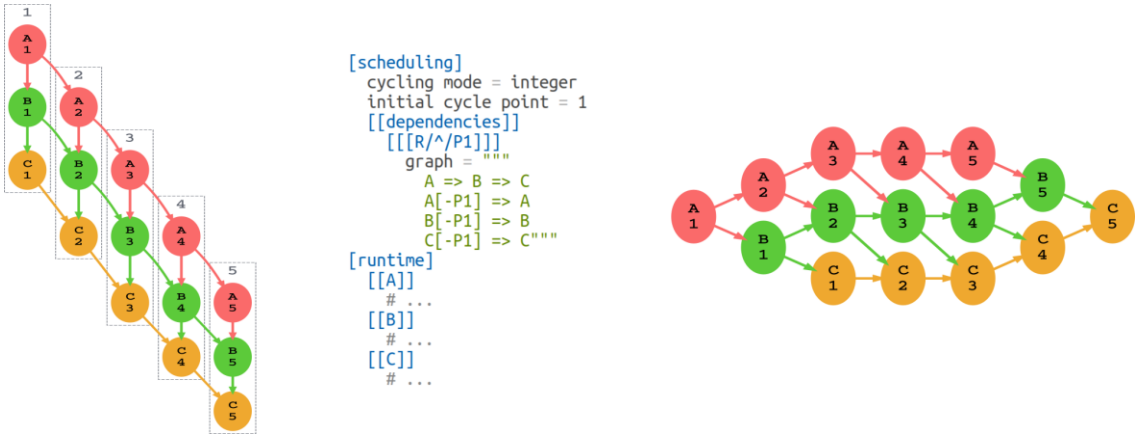

Figure 5. Cylc suite configuration for a linear pipeline **"A => B => C"** implemented with integer cycling. The workflow ensures that one instance each of A, B, and C runs concurrently and the pipeline is kept full: when **A.1** has finished processing the first dataset, **A.2** can start on the second one at the same time as **B.1** begins processing the output of **A.1**, and so on. The recurrence expressions that determine graph cycling sequences are described in the main text. Here **R/^/P1** defines an integer sequence with interval 1, starting at the suite initial cycle point. The artificial cross-cycle dependence **"A[-P1] => A"** ensures that only one instance of A can run at a time; and similarly B and C. The graphs show five iterations of the workflow, with cycle point boxes added for clarity on the left. If available compute resource supports more than three concurrent jobs just remove the cross-cycle dependence and Cylc will run many cycles at once. Task runtime configuration is omitted, but it would likely involve retrieving datasets by cycle point and processing them in cycle point-specific shared workspaces under the self-contained suite run directory.

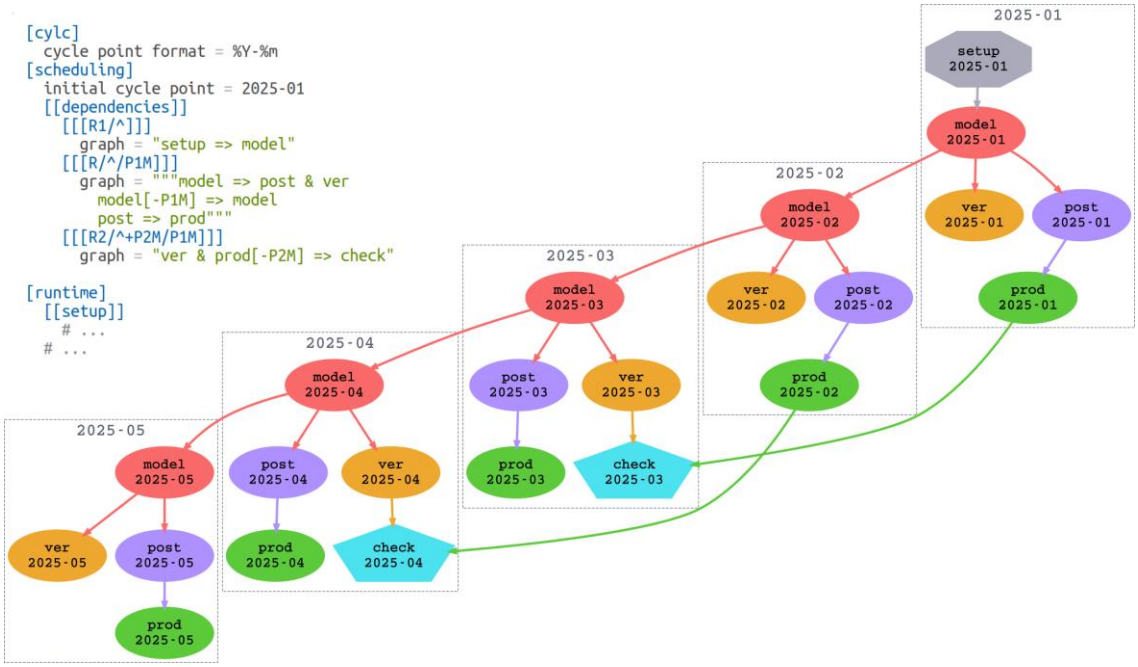

Figure 6. Cylc suite configuration for a toy monthly-cycling workflow: a warm-cycled atmospheric model (**model**) is followed by post-processing (**post**), forecast verification (**ver**), and product generation (**prod**) tasks; and to show a little more of what's possible, a task **check** compares some verification metric against products from two cycles earlier. The **"R1/^"** heading puts the first graph segment in once at the suite initial cycle point. For the second segment, **"R/^/P1M"** defines an ongoing monthly sequence starting at the initial cycle point. And finally, **"R2/^+P2M/P1M"** generates exactly two cycle points with a one month interval between them, two months after the initial point. The previous-instance dependence of each model run is determined by **"model[-P1M] => model"** in the main graph string. Task runtime configuration is omitted.

To properly distinguish date-time cycling from real-time scheduling it should be noted that cycle point values are just labels that distinguish task instances and anchor dependencies, and may be used by jobs to set the start date for model simulations, for example, or to identify the cycle-specific files being

processed. They have no connection to real time unless date-time clock triggers are attached to particular tasks.

Aspects of Cylc workflow configuration not covered in this article include conditional triggers; explicit task state triggers (e.g. to depend on failure of another task); message triggers; family triggers; clock triggers; inter-workflow triggers; external triggers (via arbitrary user-supplied plugin functions); configurable retry on failure; and comprehensive event handling (to send emails, or execute custom scripts in response to suite and task events). The *Cylc User Guide* (16) should be consulted for full documentation, and advice on clean and portable Cylc workflow design.

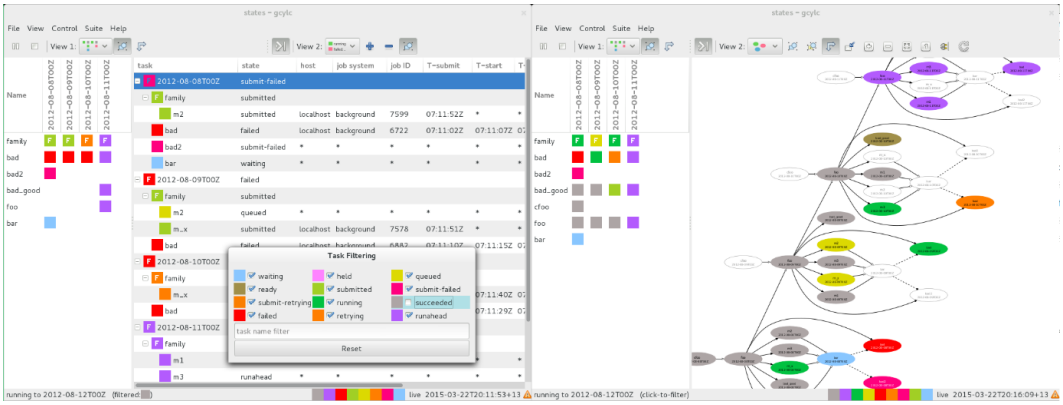

Figure 7. Two screenshots, to give an impression of what the current Cylc desktop GUI looks like. On the left, a detailed view of job ID, batch system, host, timing, etc.; on the right, a live dependency graph view. Different colors represent different task states: waiting, queued, submitted, running, succeeded, failed, etc. Views can be filtered by task state and name, and collapsed on families. Users can click on tasks to query, kill, and retrigger jobs, or view their log files, etc. Another desktop GUI displays summary states for many workflows, and there is a web interface for quick access to thousands of job logs. Work is under way to replace the desktop GUIs with a web interface.

SOFTWARE ARCHITECTURE

Unusually for a production NWP workflow scheduler, Cylc has no central server to manage all workflows for all users. Instead, a new lightweight ad hoc *suite server program* starts as the user to manage each workflow, and all job- and user-invoked clients are handled identically by servers. Any executable script or program can be utilized unmodified as a task job and executed locally or remotely (via ssh), in the background or via a workload manager such as PBS. Task configuration is encapsulated transparently in job scripts, just as the user would do it, but with some boilerplate code added to trap signals and errors and send status messages to the server REST API. Server-polling is also supported as a job-tracking mechanism, if return routing for task messages is not allowed. This simple architecture has low administrative overheads, a relatively small security footprint (everything runs as the user), it scales horizontally, and large production systems can be upgraded to new Cylc versions one workflow at a time.

Cylc is currently being re-architected to support a web UI and integration with site identity management systems, as shown in Figure 8. This is a significant change because the current UIs access the local system in ways that browsers cannot do. A major new system component is inspired by and may leverage JupyterHub (17): a privileged hub that acts as a single point of access for users, handles authentication, spawns workflow services (suites) as the user, and proxies network requests to them.

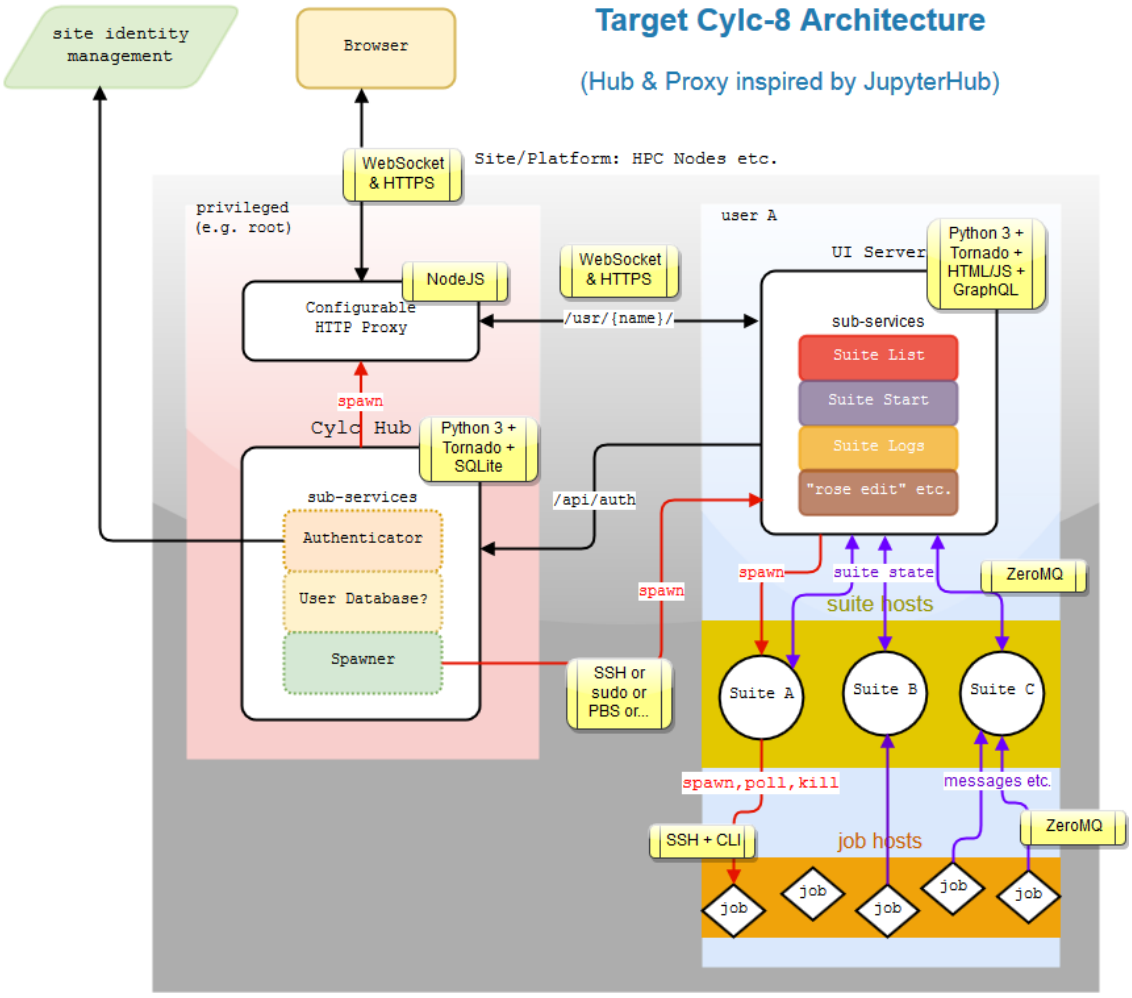

Figure 8. Target Cylc-8 architecture supporting a web UI and integration with site identity management. *This is work in progress at time of writing; some details may change during implementation.* The diagram depicts a shared multi-user multi-node (and potentially multi-cluster) platform. The privileged hub (left) is a single point of access for users, spawning Cylc workflow services (right) as user processes. A single user may have suites (and jobs) on multiple hosts. Yellow boxes show the various technologies and protocols involved. Current Cylc-7 client-server architecture is much like the "user A" box on the right, although the communications protocols are changing and the UI Server will replace current desktop GUIs.

RUNNING WORKFLOWS

Cylc's user interfaces handle everything from suite validation to run-time monitoring and control. The current GUI is shown in Figure 7. The command line interface makes workflows scriptable and provides advanced intervention capability. A single command can, for example, retrigger every failed task that matches some name and/or cycle point pattern, or dynamically *broadcast* settings and information (via environment variables) to selected groups of tasks.

Suite server programs can be very long-running, and sometimes their host servers fail or need downtime for maintenance. Cylc can select, based on load metrics, which of a pool of available hosts to start a new suite on, and running suites can self-migrate to another host if their current host is marked as condemned. State checkpoints are written to a database so that workflows can be restarted at any point, and at restart job hosts are queried to infer the fate of jobs that were orphaned when the server went down (are they still queued or running, or did they succeed or fail already?)

The performance of suite server programs depends primarily on the number of tasks per cycle in the workflow, potentially multiplied by the number of active cycles if running off the clock. GUI performance also depends on the number of tasks displayed, which can be reduced by restricting it to active tasks, but the graph view becomes impractical in very large suites as the Graphviz layout engine begins to struggle. We generally recommend that single workflows be kept to a manageable size, perhaps 1000 tasks per cycle, but with sufficient memory suite server programs scale well to tens of thousands, and

50,000 has been demonstrated. Single-suite NWP ensemble post-processing systems are rapidly approaching 100,000 jobs per cycle, however, and that currently requires splitting the problem into multiple suites or bunching multiple jobs into single tasks. Planned enhancements should allow Cylc to scale better to these levels in the future.

Cylc does not have built-in support for cloud platforms at this stage because usage to date has been largely confined to traditional HPC. But cloud instances can be used as remote job hosts, given ssh access, and even spun up by custom tasks in the workflow.

CASE STUDY: NIWA

Early incarnations of NIWA's 24/7 environmental forecasting operation in the mid-late 2000s consisted of a data-assimilating regional weather model driven by a global model feed from the Met Office, with downstream sea state, storm surge, tide, and river flow models, and several hundred associated processing tasks. These ran four times daily on an aging supercomputer with just an hour of downtime between cycles. If anything went wrong, which it frequently did, catching up from delays by sequential cycling could take 24 hours or more. Under Cylc from mid-2010 each task job could run as soon as its own prerequisites were satisfied, regardless of cycle point or the state of other tasks. This spectacularly reduced catch-up time from about 24 hours to about 30 minutes, just as in the toy example of Figure 2.

Today NIWA's forecasting system comprises 30 interdependent suites that run "out of the box" in distinct research, test, and production environments, so that the research-to-production transition is now a matter of straightforward working practice rather than (as is commonly the case) a fiendishly difficult porting exercise. Cylc is also used in other contexts at NIWA including satellite data processing, climate simulation, and earth system modeling.

CASE STUDY: MET OFFICE

In 2011 the Met Office began a project to update its workflow capability for HPC-based forecast application suites. The existing in-house systems, which were over 15 years old, allowed weather workflows to be programmed with some flexibility but they had become difficult to maintain, climate workflows were fixed, suites contained many site-specific assumptions, and there was a leakage of logic between the workflow manager and its applications.

Cylc was trialed against ECMWF's SMS, identified as the leading weather workflow scheduler at the time. The decision to migrate to Cylc in 2012 was driven by the openness of the project, ease of Python development, portability and generality (Cylc is entirely agnostic to the applications it manages, for instance), and a strategic desire to unify the handling of cycling workflows for weather and climate, as well as research and operations.

Following the migration, Cylc enabled users to develop workflows tailored for increasingly complex requirements. A more powerful cycling syntax based on ISO 8601 date-time recurrence expressions (and integer cycling too) was introduced for flexibility in weather applications, with alternate calendars for climate simulations. Robustness was enhanced to ease recovery from system outages. After migrating to a new HPC in 2014 with minimal effort, Cylc's job management subsystem was further developed to handle the large increase in capacity more efficiently.

Today, the entire research and production workload, more or less, of the 460,000 core Met Office HPC is controlled by Cylc, with a pool of 10 moderately-spec'd VMs routinely hosting 600–700 suite server programs for several hundred active users at any one time. Cylc's robustness has been put to the test by two major unscheduled power outages in the computer hall. In both cases the workflows were easily brought back up, with no problems attributed to Cylc.

CASE STUDY: EWATERCYCLE

The eWaterCycle project (18)(19) at the Netherlands eScience Center and Delft University of Technology aimed to demonstrate that a system predicting flood and drought events 10 days in advance, worldwide and at unprecedented high resolution, can be constructed from Open Source components. The resulting global hydrological forecasting framework uses Cylc to orchestrate an ensemble of data-assimilating hydrological models, forced by a weather forecast ensemble, to predict river discharge and quantify uncertainty in the forecasts. The eWaterCycle team found Cylc's cycling workflow configuration to be compact, readable, and powerful. Features such as clock triggers, event triggers, configurable retries, and event handlers allowed the team to design a system that waits for its input data to become available in each cycle, and can automatically retry failed tasks or choose alternate paths through the workflow. Cylc's manual intervention capabilities also proved helpful, in recovering from unusual or unexpected

problems. The current eWaterCycle II project aims to build a more portable, community multi-model environment for hydrological experiments and analyses. To inform this effort, in the small EOSCPilot-funded "FAIRifying eWaterCycle" project (FAIR: Findability, Accessibility, Interoperability, and Reusability) the eWaterCycle team created a reproducible version of their system that will be easier for other researchers to set up and use. To improve portability the team relied on Docker containers, and the Common Workflow Language (CWL) (20)—a specification for describing analysis workflows and tools in a way that is portable and scalable across a variety of software and hardware environments. CWL does not support cyclic workflows, however, so the team opted to only describe the workflow steps in CWL, and to continue orchestrating the cycling system with Cylc. Since CWL is not directly supported by Cylc, each CWL step is run with the CWL reference runner. Figure 9 shows the architecture of the FAIRified eWaterCycle system, and Cylc's place in it.

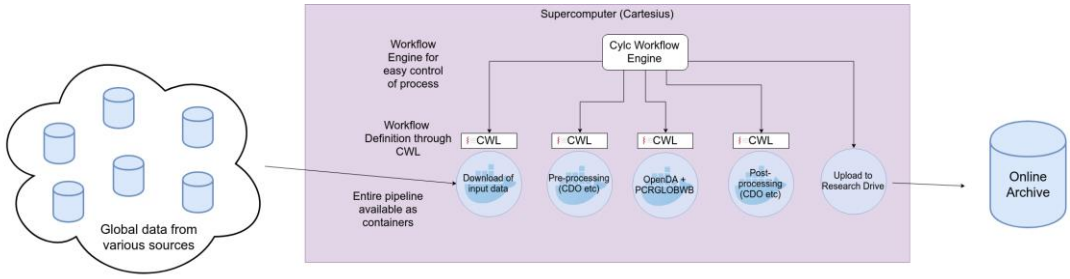

Figure 9. The FAIRified eWaterCycle Cylc suite combines Cylc with use of CWL for a fully reproducible cyclic workflow. Global data is downloaded each day from various sources, in several steps. The pipeline first preprocesses the data, followed by a forecast run in which a number of instances of the same model are combined into an ensemble. In post-processing, the output of the ensemble is combined into a single prediction, and finally uploaded to an online archive for subsequent visualization and analysis.

COMMUNITY ENGAGEMENT

Sites

Cylc's Open Source license has proved important for institutional uptake because it allows sites to protect their large investment in workflow automation by getting involved in the project and influencing its direction. On the other hand, the open development model can be challenging for sites with a strong security focus and business managers who rightly or wrongly see Open Source software as risky. To counter this we can promote the advantages of Open Source, such as leveraging the combined talent and support of the community, and point to the fact that security vulnerabilities are better exposed to scrutiny and fixed than hidden from sight. Sound development practices should of course be used, and be seen to be used, at all times.

Developers

Cylc has received contributions from about 30 developers to date, although most significantly from a core of fewer than 10. We manage the codebase with git, on GitHub, using the straightforward *GitHub Flow* model for collaborative development. Significant changes are discussed and agreed in GitHub Issues before implementation. Code contributions are developed on feature branches in developer forks and posted as Pull Requests for review by the maintainers. If accepted, they are merged to the repository master branch under a contributor license agreement. Every commit to the project triggers a large battery of tests with coverage reporting and comprehensive static code analysis.

Users

Early versions of Cylc were focused on the new scheduling algorithm for cycling systems, with a minimal user interface and scant documentation. The project's evolution since then can be characterized largely as a process of user- and usage-driven enhancement in the face of rapidly evolving workflow automation requirements.

Uptake has been strongest, unsurprisingly, where an institutional decision was made to use Cylc, and where the complexity of the work being done makes automation an obvious necessity. Elsewhere, we have been persuaded that we need to put more effort into showing how easily Cylc "scales down" as well as up, because it can be difficult to wean some people from their manual and ad hoc scripted

working practices. We have seen rapid increases in productivity and in the nature of the work that is possible, however, when users do take the plunge.

Feedback, bug reports, and feature requests tend to come directly to development team members at the main Cylc sites, or via GitHub Issues. We also have a traditional mail forum for release announcements and remote users, but we are looking at ways to replace that and centralise all discussions. At the largest Cylc site (Met Office) a community of practice known as the Suites Guild has arisen independently of the development team, for user-driven discussion of workflow design and other topics. A Suite Design working group has also been convened within the international Unified Model Partnership's Technical Infrastructure Programme, with quarterly meetings focused largely on collaborative development of site-portable weather and climate workflows.

FUTURE DIRECTIONS

Our immediate goals (in progress) are to port Cylc to Python 3, move to standard Python packaging, and replace the current simple client-server architecture and native desktop GUIs with a new web architecture and UI as described above.

We will then target performance for anticipated increases in workflow size and complexity into the exascale computing era. Plans include a Python API for advanced workflow configuration and better modularity, a data-modeling interface as an alternative to abstract dependencies, automatic job batching, and a lightweight event-driven scheduler kernel that can be used inside tasks to make hierarchical workflows more practical.

Finally, as noted in the eWaterCycle Case Study above, Cylc does not support the Common Workflow Language (20) because CWL does not understand cycling, and that is fundamental to our primary use cases. However, we may consider collaborating to extend CWL in the future if the community is interested.

ACKNOWLEDGEMENTS

We gratefully acknowledge all contributors to Cylc and the libraries it depends on; the many scientists who have discussed their cycling workflow needs with us; and the eWaterCycle team.

REFERENCES

1. *Cylc: A Workflow Engine for Cycling Systems.* Oliver, Hilary; Shin, Matthew; and Sanders, Oliver. 2018, Journal of Open Source Software, pp. 3(27), 737, https://doi.org/10.21105/joss.00737

2. *Cylc web site* [Online] 2019. https://cylc.github.io/cylc; https://cylc.org

3. *Cylc release DOI* [Online] 2019. https://doi.org/10.5281/zenodo.594577

4. *Scientific workflow management and the Kepler system.* Ludascher, Bertram; et al. 1039-1065, Concurrency and Computation: Practice and Experience., 2006, Vol. 18

5. *Existing Workflow Systems* [Online] Common Workflow Language repository, 2019. https://github.com/common-workflow-language/common-workflow-language/wiki/Existing-Workflow-systems

6. *ecFlow* [Online] ECMWF, 2018. https://confluence.ecmwf.int/display/ECFLOW

7. *Known Cylc Sites* [Online] 2019. https://cylc.github.io/cylc/users.html

8. *Airflow* [Online] Apache, 2019. https://airflow.apache.org/index.html

9. *ESiWACE Cylc Page* [Online] ESiWACE, 2019. https://www.esiwace.eu/services/sup_cylc

10. *Modernizing U.S. Navy NWP Operations* [Online] Conference presentation, 2018. https://www.ecmwf.int/sites/default/files/elibrary/2018/18606-modernizing-u-s-navy-nwp-operations-toward-distributed-hpc.pdf.

11. *PBS/Cylc Integration Project* [Online] Conference presentation, 2018. https://www.esiwace.eu/events/esiwace-workshop-on-workflows-1/presentations/david-block-overview-of-pbs-integration-with-cylc.

12. *Automated model optimisation using the Cylc Workflow Engine* (Cyclops v1.0). Gorman, Richard M and Oliver, Hilary J. 2018, Geosci. Model Dev., pp. 11(6); doi:10.5194/gmd-11-2153-2018.

13. *Seamless management of ensemble climate prediction experiments on HPC platforms*. Manubens-Gil, D, et al. Innsbruck: International Conference on High Performance Computing & Simulation (HPCS), 2016, pp. 895-900. doi: 10.1109/HPCSim.2016.7568429

14. *Scientific workflow design for mere mortals*. McPhillips, Timothy M; et. al. 2009, Future Generation Comp. Syst., pp. (25) 541-551

15. *ISO Date-Time Library* [Online] 2019. https://github.com/metomi/isodatetime.

16. *Cylc User Guide*. Oliver, Hilary; and Contributors. [Online] 2019. https://cylc.github.io/cylc/documentation.html

17. *JupyterHub* [Online] 2019. https://jupyter.org/hub

18. *eWaterCycle: a hyper-resolution global hydrological model for river discharge forecasts made from open source pre-existing components*. Hut, Rolf; et al. Geosci. Model Dev. Discuss, 2016/10/20. doi:10.5194/gmd-2016-225

19. *eWaterCycle* [Online] 2019. https://www.ewatercycle.org

20. *Common Workflow Language v1.0. Specification* [Online] Common Workflow Language working group, 2018. doi:10.6084/m9.figshare.3115156.v2; https://w3id.org/cwl/v1.0