SciPipe — Turning Scientific Workflows into Computer Programs

Samuel Lampa[1,*], Martin Dahlö[1], Jonathan Alvarsson[1], and Ola Spjuth[1]

[1]Department of Pharmaceutical Biosciences and Science for Life Laboratory, Uppsala University, Sweden

February 25, 2020

Scientific Workflows are becoming increasingly popular as a way to automate complex scientific computations consisting of multiple programs.

One of the main motivations behind this development is increased robustness and reproducibility of computational analyses. Chaining together multiple programs using plain scripts, as is often the first step in automating a pipeline, can easily become fragile and error-prone due to the manual management of file paths and program invocations. Also, plain scripts are not optimal if you for some reason have to cancel a run and try to restart it from any partially finished steps. It can be hard to know which output files are properly finished and which are truncated from the cancelled run. Last but not least, plain scripts do not by default save an execution trace of what was run, such that the full procedure used to create a specific output file can be clearly presented. These are all aspects that scientific workflows are designed to help with.

Despite many hundreds — if not thousands — of scientific workflow tools published over the years, there can still be significant challenges when trying to use many of them.

One reason for this is that many workflow tools have been designed with a very narrow use case in mind, often building in assumptions unique to the specific problem domain aimed at, which might make them less applicable for scientific pipeline needs in general.

Even among the many hundreds of general workflow tools, surprisingly many contain various constraints and assumptions limiting their generality. Often, this might not be obvious before trying to apply them to complex tasks.

At Dept. of Pharmaceutical Biosciences we have spent the last few years using workflow tools to automate machine learning pipelines for predictive toxicology among other things. We have reviewed the top dozen workflow tools popular in our field of bioinformatics. We even tried out Luigi, created by music company Spotify, which is popular in industry and far from a bioinformatics-aimed tool.

A recurring theme has been how often tools contain various limits that make them hard to use for complex use cases. Machine learning workflows in particular often lead to highly complex workflows because of their common inclusion of cross validation and parameter sweeps from hyperparameter optimization. Not only are these workflows complex but they also show some characteristics not always common in other domains: The need for dynamic scheduling. That is, they need to be able to parametrize and start tasks based on information obtained during the workflow run. Somewhat surprisingly, this is a problem in a majority of workflow tools because of how common it is that they have a strict separation between the scheduling and execution phases of the workflow run. That is, after a workflow has progressed into its execution phase it is commonly not possible to schedule and start new tasks with parameter values obtained in the current run. At least not without initiating completely new workflow runs.

Anyways, after a lot of evaluation, it seemed at the time that Luigi [1] was the most promising way forward for us. Later we learned that Luigi's functional programming inspired API design was

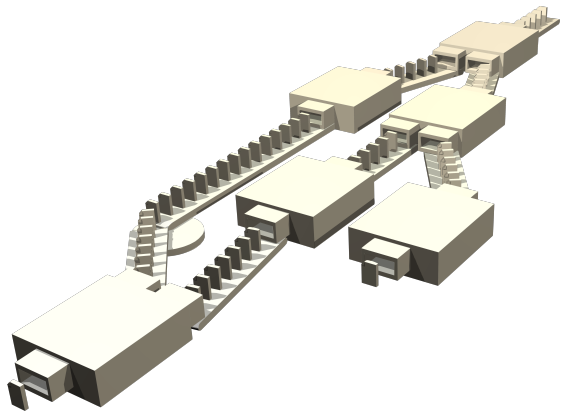

Figure 1: Flow-based programs can be likened to a factory with processing stations connected with conveyor belts, upon which data items "flow" through the network of stations and conveyor belts.

not quite fit for our needs of dynamic workflow rewiring, which is one of the reasons why we later chose to develop the SciPipe library (more on that later) but there is an interesting story to tell about our Luigi phase too.

Workflows as computer programs

It turned out that because Luigi was implemented as a programming library, it was flexible enough that we could build an alternative API on top of it, which resulted in the SciLuigi helper library [2]. Specifically, SciLuigi enabled us to keep the dependency network definition separate from task definitions – a core principle in flow-based programming, as we will explain shortly – which makes it much easier to reconnect workflows without changing internals of workflow components in complex ways.

This positive experience from a programming API-based workflow tool helped push a realization that has grown over a number of years of discussions and experimenting: Workflows are in the end just a glorified version of computer programs.

It turns out that although many use cases can be modeled as simple linear sequences of program invocations depending on each other, not all cases are that simple. There are many examples where the need for logic to control the workflow structure is so complex that any attempt at modeling it with a declarative workflow language ends up implementing what is already available in existing programming languages.

This tendency can also be seen in popular workflow engines building on domain specific languages (DSLs). Tools that become popular often either have a really flexible DSL from the start, *e.g.*, because the DSL is implemented in an existing scripting language (*e.g.*, Nextflow [3], building on the Groovy language), or their DSLs are step by step becoming increasingly complex, and in the end approaching computer programming languages in their capability (*e.g.* Cuneiform [4], implementing a powerful functional language [5]).

SciPipe

The above lesson is something we were taking into account when we set out to design the SciPipe workflow tool from scratch after finding out about several limitations with the Luigi/SciLuigi setup we were already using (primarily the need for dynamic scheduling and compile time warnings about errors in workflow connectivity).

SciPipe [7] is designed from the start as a programming library embedded in the implementation language (Google's Go, or "Golang"), rather than inventing new textual syntax or graphical tools. It thus leverages the full power and flexibility of the Go programming language for implementing workflow logic. So far we haven't encountered a workflow use case that we haven't been able to model with this approach. Even complex machine learning workflows with nested branching has been solvable, as exemplified in a recent paper by the authors [8]. Another nice side effect of the Go language in particular, is that it compiles to self-contained executable files, which makes deployment of most Go programs very straightforward. SciPipe is open source software (MIT licensed). A simple "Hello World" style workflow example is shown in Figure 2. For more information, source code and documentation, see [6] and [9].

Flow-based programming focuses on data flow

Now, there *are* some differences between most workflow programs and most normal programs.

```go
package main

import (
    // Import the SciPipe library
    "github.com/scipipe/scipipe"
)

func main() {
    // Initialize a workflow with max 4 concurrent tasks
    wf := scipipe.NewWorkflow("hello_world", 4)

    // Initialize processes, and file extensions
    hello := wf.NewProc("hello", "echo 'Hello ' > {o:out|.txt}")
    world := wf.NewProc("world", "echo $(cat {i:in}) World > {o:out|.txt}")

    // Define the data flow from out-port to in-port
    // on the two processes in the workflow
    world.In("in").From(hello.Out("out"))

    // Run the workflow
    wf.Run()
}
```

Figure 2: A simple example workflow implemented with SciPipe. The workflow consists of two processes: "hello" and "world", where "hello" writes the word "Hello" to a file, which the "world" process uses as input and appends the word "World", before writing the combined output to a new file. Processes are defined from shell commands, where placeholders on the form of {`i:inport|file-extension`} and {`o:outport|file-extension`} are used to define in-ports and out-ports. These placeholders act as templates that will be replaced with specific file names during the workflow execution, based on the files that are sent on the processes' in-ports. Although this is a very simple example, Go programmers will recognize that this workflow definition is also a simple Go program. More detailed examples are provided on the main SciPipe website [6].

The main difference can be seen in the strong focus in workflow programs on *data flow*. When defining how multiple programs depend on each other we are in effect defining how the data will flow through these programs. There is actually a looming risk for workflow tool developers to miss this detail and model the workflow dependency graph as just dependencies between programs and not their inputs and outputs. This can quickly lead to problems because one program typically don't depend on just one other program but rather specific outputs of possibly multiple upstream programs. That is, data needs to be a first class citizen when defining workflows, or we risk missing important details that will otherwise be buried in less thought-out *ad hoc* code.

One paradigm that takes note of this fact is Flow-based programming (FBP) [10]. Invented at IBM in the late 1960s and used on large mainframe computers at banks and other large institutions, the flow-based programming paradigm has seen some resurge in popularity in recent years, possibly driven by the recent trends towards multi-core CPUs, distributed computing and message oriented architectures.

Flow-based programming ordains a number of design principles. The most important one in the context of dependency definition though is that it models dependencies between processes in an appropriate level of detail; in terms of data inputs and outputs. The data itself is modeled through so called "information packets" and inputs and outputs as "ports" - A kind of pluggable component between which yet another concept can be connected; "channels". Channels have bounded buffers and act as a kind of conveyor belt between processes, letting processes work on information packets from its in-ports asynchronously and sending them on their out-ports (mostly) independently from the processing rate of other processes. Fig-

ure 1 tries to depict this in an artistic way.

The core idea of flow-based programming is how it draws all of these things together into a declarative data flow definition *separate* from the process implementations. This allows easy rewiring of the data flow without changing a single line of process implementations. It thus allows to create libraries of reusable components which can be plugged in at any place in the program network, as long as its in- and out-ports are compatible with the in- and out-ports they connect to.

Note that while FBP is often associated with visual programming, that is far from a requirement. In our experience, skipping the visual part and focusing on a simple programming API has more than fulfilled our needs, while letting us avoid depending on the complexity of a visual programming framework. The fact that the Go language provides the most important pieces for this to work (independently running go-routines and channels with bounded buffers) certainly helps.

Reproducibility in workflow programs

We have presented some rationale for writing workflows as computer programs, but what about the other aspects important for workflows, such as reproducibility? Aren't there areas where the traditional ways of designing workflows still have its merits?

In terms of reproducibility we argue that maximizing the declarative nature of the workflow definition is not necessarily the key requirement, although that has been a prioritized focus area in many workflow frameworks. Rather, we think what is most important is to be able to produce an unambiguous record of how each data output from a workflow was produced.

In SciPipe we think we have found a simple yet powerful way to capture provenance, leveraging its low complexity and flow-based programming based design.

The fact that programs are defined, even programmatically, as a graph of processes with dataflow connections on which data items "flow", made it straight-forward to implement a provenance mechanism around the information packets themselves; Letting them aggregate information about the processing nodes in the graph as they pass through them. Since one data item might be the merger of two initially separate dataflow paths this is done in a hierarchical way. This way of tracing the execution also means that the provenance will be captured at the granularity that matters; at the data level. In concrete terms it means SciPipe can — and does — save a provenance record for every output file of the workflow.

Saving a provenance record only for the full workflow run risks introducing a distance between the data and the workflow run that over time will only grow bigger. By instead saving the provenance information in an accompanying file associated with every output file of the workflow, as is done in SciPipe, means it is far easier to ensure this information stays associated.

Conclusion

By providing increased robustness to failure and reproducibility of results, workflows are an established workhorse of computational science as well as an increasingly important part also in data pipelines in industry.

Dominant approaches to workflow authoring, such as specialized domain specific languages or graphical workflow authoring tools, do not always cater to the full spectrum of workflow needs, such as machine learning workflows consisting of parameter sweeps nested with cross validation and dynamic scheduling.

For such complex and dynamic needs we argue for the benefits of workflows implemented as computer programs, using workflow tools implemented as programming libraries. Furthermore, by using flow-based programming principles, a declarative while programmatic API can be presented, while still hiding a lot of low-level details under the hood.

Being able to provide workflows in the form of compiled executable files has benefits both for the ease of deployment and future reproducibility. Workflow tool implementations are at risk for bit rot or incompatible changes well before the mainstream programming languages — or processor architectures — are.

For more information on SciPipe, check out the main website at scipipe.org.

References

[1] Erik Bernhardsson, Elias Freider, and Arash Rouhani. spotify/luigi - GitHub. [Online; Accessed 23-March-2019], https://github.com/spotify/luigi.

[2] Samuel Lampa, Jonathan Alvarsson, and Ola Spjuth. Towards agile large-scale predictive modelling in drug discovery with flow-based programming design principles. *Journal of Cheminformatics*, 8(1):67, 2016.

[3] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature Biotech*, 35(4):316–319, 2017.

[4] Jörgen Brandt, Marc Bux, and Ulf Leser. Cuneiform: a functional language for large scale scientific data analysis. In *EDBT/ICDT Workshops*, pages 7–16, 2015.

[5] Jörgen Brandt, Wolfgang Reisig, and Ulf Leser. Computation semantics of the functional scientific workflow language cuneiform. *Journal of Functional Programming*, 27:e22, 2017.

[6] Samuel Lampa. SciPipe website. [Online; Accessed 23-March-2019], http://scipipe.org.

[7] Samuel Lampa, Martin Dahlö, Jonathan Alvarsson, and Ola Spjuth. SciPipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines. *GigaScience*, 8(5), April 2019.

[8] Samuel Lampa, Jonathan Alvarsson, Staffan Arvidsson Mc Shane, Arvid Berg, Ernst Ahlberg, and Ola Spjuth. Predicting off-target binding profiles with confidence using conformal prediction. *Frontiers in Pharmacology*, 9:1256, 2018.

[9] Samuel Lampa, Martin Czygan, and Jonathan Alvarsson. SciPipe source code repository at GitHub. [Online; Accessed 23-March-2019], https://github.com/scipipe/scipipe.

[10] J Paul Morrison. *Flow-Based Programming: A new approach to application development*. Self-published via CreateSpace, Charleston, 2nd edition, May 2010.