Building and Documenting Workflows with Python-Based Snakemake

Johannes Köster[1,2] and Sven Rahmann[1]

1 **Genome Informatics, Institute of Human Genetics, Faculty of Medicine, University of Duisburg-Essen, Hufelandstr. 55, 45122 Essen, Germany,** {johannes.koester,sven.rahmann}@uni-due.de
2 **Paediatric Oncology, University Childrens Hospital Essen, Hufelandstr. 55, 45122 Essen, Germany**

Abstract

Snakemake is a novel workflow engine with a simple Python-derived workflow definition language and an optimizing execution environment. It is the first system that supports multiple named wildcards (or variables) in input and output filenames of each rule definition. It also allows to write human-readable workflows that document themselves. We have found Snakemake especially useful for building high-throughput sequencing data analysis pipelines and present examples from this area. Snakemake exemplifies a generic way to implement a domain specific language in python, without writing a full parser or introducing syntactical overhead by overloading language features.

1998 ACM Subject Classification D.3.2 Language Classifications, D.3.4 Processors, J.3 Life and Medical Sciences

Keywords and phrases workflow engine, dependency graph, knapsack problem, Python, high-throughput sequencing, next-generation sequencing

Digital Object Identifier 10.4230/OASIcs.GCB.2012.49

1 Introduction

Workflow systems manage the ever-increasing complexity of computational pipelines in large-scale bioinformatics applications, e.g. for image processing or next-generation sequencing data analysis. Existing solutions range from graphical systems, e.g. Biopipe [5], Taverna [11], Galaxy [2] and GeneProf [4], to frameworks supporting only text based workflow definitions, e.g. Ruffus [3], Pwrake [14], GXP Make [15] and Bpipe [12]. While graphical solutions are easy to learn, text-based representations can be advantageous: Workflows can be edited faster and directly on servers without graphical environments and developers can collaborate on them via source code management tools.

A basic way to define a workflow is provided by the build system GNU Make [13] that was originally intended to compile source code but can be used to execute arbitrary pipelines of command line applications. In contrast to many of the above systems, here the actual workflow (dependencies, parallelization) is inferred from a given set of rules with input and output files, rather than relying on an explicit specification. This idea was adapted by Pwrake and GXP Make, and is also the foundation of *Snakemake*.

In the spirit of the Python language, Snakemake complements these prior works with a syntax close to pseudocode. The resulting workflows are human-readable and serve as both documentation and formal definition. Further, Snakemake provides scalability because it optimizes the number of parallel processes with respect to provided CPU cores and needed

German Conference on Bioinformatics 2012 (GCB'12).
Editors: S. Böcker, F. Hufsky, K. Scheubert, J. Schleicher, S. Schuster; pp. 49–56

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

threads and can make use of single machines as well as cluster engines without the need to modify the workflow. In contrast to Pwrake and GXP Make, Snakemake does not rely on any password-less SSH setup or custom server processes runing on the cluster nodes. Finally, Snakemake is the first system to support multiple named wildcards and output files in rules, which we show to be useful in a common bioinformatics workflow.

This paper also presents a uniform way to implement a domain specific language (DSL) that avoids to write a full parser. The approach avoids unnecessary syntactic overhead to be exposed to the user (e.g., Python decorators or unintuitively overloaded operators).

The remainder of this paper is structured as follows: In Section 2.1 we describe the DSL for defining and documenting Snakemake workflows, which is illustrated by an exemplary pipeline for calling single nucleotide polymorphisms (SNPs) in Section 2.2. Section 2.3 focuses on the technical implementation of the Snakemake language as a Python-based DSL. The engine that executes the workflow is described in Section 3, and a final conclusion is drawn in Section 4.

2 The Snakemake Workflow Language

2.1 Language Definition

A workflow is defined in a *Snakefile* via a domain specific language close to Python syntax. It consists of rules that define how to create output files from input files. The workflow is implied by dependencies between the rules that arise from one rule needing an output file of another as an input file.

A rule definition in a Snakefile specifies (1) a name, (2) any number of input and output files and (3) either a shell command or Python code that creates the output from the input. Input and output files may contain multiple named wildcards and are specified as Python strings. In extended Backus-Naur Form (EBNF), the Snakemake syntax can be defined as follows.

$$
\begin{aligned}
\text{snakemake} &= \textit{statement} \mid \text{rule} \\
\text{ni} &= \textit{NEWLINE INDENT} \\
\text{rule} &= \text{"rule"}\ (\textit{identifier} \mid \text{""})\ \text{":"}\ \text{ruleparams} \\
\text{ruleparams} &= [\text{ni input}]\ [\text{ni output}]\ [\text{ni threads}]\ [\text{ni (run} \mid \text{shell)}]\ \textit{NEWLINE}\ \text{snakemake} \\
\text{input} &= \text{"input"}\ \text{":"}\ \textit{parameter\_list} \\
\text{output} &= \text{"output"}\ \text{":"}\ \textit{parameter\_list} \\
\text{threads} &= \text{"threads"}\ \text{":"}\ \textit{integer} \\
\text{run} &= \text{"run"}\ \text{":"}\ \text{ni}\ \textit{statement} \\
\text{shell} &= \text{"shell"}\ \text{":"}\ \textit{stringliteral}
\end{aligned}
$$

The terminal symbols *NEWLINE* and *INDENT* and the italic non-terminals refer to plain Python syntax, e.g. *statement* allows any Python statement, *stringliteral* denotes a Python string (i.e. `"..."` or `"""..."""`), and *parameter_list* enables the same syntax as for the parameters of Python functions.

2.2 Example: A SNP-Calling Pipeline

Listing 1 shows a single nucleotide polymorphism (SNP) calling pipeline in the context of next generation sequencing, implemented as a Snakemake workflow. SNP calling is a typical

Listing 1 Example Snakefile for read mapping and SNP calling.

```
SAMPLES = "100 101 102 103".split()
REF = "hg19.fasta"
DBSNP = "dbsnp.vcf"

rule all:
    input:  "calls/{sample}.vcf".format(sample = sample)
            for sample in SAMPLES

rule map_reads:
    input:  ref = REF, reads = "{sample}.{group}.fastq"
    output: temp("{sample}.{group}.sai")
    threads: 8
    shell:  "bwa aln -t{threads} {input.ref} {input.reads} > {output}"

rule sai_to_bam:
    input:  REF, "{sample}.1.sai", "{sample}.2.sai",
                "{sample}.1.fastq", "{sample}.2.fastq"
    output: protected("{sample}.bam")
    shell:  "bwa sampe \
            -r'@RG\\tID:{wildcards.sample}\\tSM:{wildcards.sample}'\
            {input} | samtools view -Sbh - > {output}"

rule sort:
    input: "{name}.bam"
    output: "{name}.sorted.bam"
    shell: "samtools sort {input} {wildcards.name}.sorted"

rule index:
    input:  "{name}.sorted.bam"
    output: "{name}.sorted.bam.bai"
    shell:  "samtools index {input}"

rule realign_targets:
    input:  ref = REF, dbsnp = DBSNP
    output: "known.intervals"
    shell:  "gatk -T RealignerTargetCreator -R {input.ref}\
            -known {input.dbsnp} -o {output}"

rule realign:
    input:  bam = "{sample}.sorted.bam",
            bai = "{sample}.sorted.bam.bai",
            ref = REF, intervals = "known.intervals"
    output: "realigned/{sample}.sorted.bam"
    shell:  "gatk -T IndelRealigner -I {input.bam} -R {input.ref}\
            --targetIntervals {input.intervals} -o {output}"

rule call_snps:
    input:  bam = "realigned/{sample}.sorted.bam",
        bai = "realigned/{sample}.sorted.bam.bai",
        dbsnp = DBSNP, ref = REF
    output: "calls/{sample}.vcf"
    shell:  "gatk -T UnifiedGenotyper --dbsnp {input.dbsnp}\
            -I {input.bam} -R {input.ref} -o {output}"
```

task e.g. in cancer genomics [10]. In this example, paired-end sequence reads are given as `.fastq` files for four samples with IDs from 100 to 103, and shall be mapped to the human reference genome. Afterwards, reads are realigned to avoid artifacts due to the mapping heuristic, and finally SNPs are called. The Snakefile consists of seven rules that each start with the keyword `rule` followed by their name and the indented definitions of input and output files and executable shell commands.

In lines 1–3, a list of sample IDs is created, the desired genome reference and the database of known SNPs is specified in plain Python. As Snakemake rules can intermix with any Python statement, accessing configuration files or environment variables, even waiting for user input or opening a graphical user interface is possible. In line 9, the rule `map_reads` aligns the sequence reads to the reference genome with the BWA software [7], which is assumed to be installed on the system. Specified in braces inside the input and output files, the rule uses two *named wildcards*, since it shall be applied to the first and the second read of each read pair (wildcard `{group}`) from each sample (wildcard `{sample}`). BWA creates suffix array intervals (`.sai`-files) which are converted to the common format for aligned reads in the rule `sai_to_bam` (line 15). These two rules illustrate major patterns for accessing input and output files inside a rule. In the rule `map_reads`, input files are named and can be accessed as attributes of the `input`-object (e.g. `{input.ref}` in line 13). In rule `sai_to_bam`, all input files are accessed at once (separated by a space) via `{input}` (line 21).

BWA's internal `.sai`-files can be deleted once the `.bam`-files are created, which are in turn worth to be write-protected to avoid accidental deletion. Snakemake supports this by marking files as `temp` (line 11) or `protected` (line 18), respectively.

Next, the rule `realign` (line 39) uses the GATK [9] to ensure that the read alignments are free of artifacts from the mapping heuristic. It depends on the rules `sort` (line 23) and `index` (line 28) being applied to the alignments from BWA to create an index for the `.bam` file using samtools [8].

The actual SNP calling with GATK takes place in the rule `call_snps` (line 47); the result is saved to `.vcf`-files. Here, the alignments created by the rule `realign` are used. Further, another `.bam`-index has to be created, hence the rule `index` is used a second time.

When Snakemake is invoked without a specific target, the first rule (here called `all`) in line 5 is executed, which ensures that the `.vcf`-files and hence all needed intermediate files are created for each sample.

In place of shell commands, Snakemake allows to write pure Python code using a `run` block instead of a `shell` block to create the output files of a rule. For example, we may want to plot the coverage histogram as a quality control.

```
from matplotlib.pyplot import hist, savefig
rule plot_coverage_histogram:
    input: "{sample}.bam"
    output: hist = "{sample}.coverage.pdf"
    run:
        hist(list(map(int,
            shell("samtools mpileup {input} | cut -f4",
            iterable = True))))
        savefig(output.hist)
```

Here, we use the `hist` function of Python's matplotlib module [6] and parse the output of Snakemakes `shell` function, that invokes samtools to calculate a per-nucleotide coverage. This demonstrates how to apply Python code to the result of shell commands.

The example workflow illustrates the readability of the language and how the workflow

rules document themselves. The advantage of conceptually separating a rule's name from the output file(s) it produces becomes evident, as does the benefit of mutiple named wildcards for input and output files.

2.3 Implementation

In general, two extreme approaches for implementing a domain specific language are thinkable: a full parser can be generated, or in-language facilities can be overloaded. The first allows a concise syntax, the second (that can be approached e.g. by Python decorators or operator overloading) can sometimes lead to syntactical overhead and unintuitive constructs. Here we present a third way, using a finite automaton that translates tokens from the Snakemake language to a sequence of Python tokens, that are interpreted by the ordinary Python interpreter. Thereby, only a small subset of tokens needs to be altered. Formally, the procedure can be specified by an automaton that recognizes the Snakemake grammar (Section 2.1) and an emission function that emits Python language terminal symbols based on the current state and token.

▶ **Definition 1** (Snakemake Language Automaton)**.** Given the Snakemake grammar, let NT be the set of non-terminals and T be the set of terminal symbols. Over the alphabet of tokens $\Sigma = T$, the Snakemake language automaton is denoted as $(Q, q_0, F, \Sigma, \delta)$ with states $Q = NT$, start state $snakemake \in NT$, and a single final state $F = \{statement\}$. The transition function $\delta : \Sigma \times Q \to Q$ is given by the EBNF grammar of Section 2.1.

The above finite automaton definition is possible because in the Snakemake grammar each non-terminal is uniquely determined by a prefix of terminal symbols. We extend the automaton with an emission function $e : Q \times \Sigma \to \{(t_0, t_1, \dots) \mid t_0, t_1, \dots \in T'\}$, where T' is the set of terminal symbols in the Python language, such that rule definitions are converted to Python API calls. For example, the rule `map_reads` from Listing 1 is translated to the following Python code using the (less user-readable) function decorator syntax (function decorators are higher-order functions that operate on Python functions). This internal representation is evaluated by the Python interpreter.

```
@rule("map_reads")
@input(ref = REF, reads = "{sample}.{group}.fastq")
@output(temp("{sample}.{group}.sai"))
@threads(8)
@run
def __map_reads(input, output, wildcards, threads):
    shell("bwa aln -t {threads} {input.ref} {input.reads} > {output}")
```

In Snakemake, multiple named wildcards are allowed in input and output files. To determine if a rule can create a requested file, its defined output files are matched against the requested file. Thereby, wildcards are replaced by the Python regular expression `.+` per default (i.e. any non empty string is matched in a greedy fashion). The substring a wildcard matches to is then propagated to the input files. When the rule `map_reads` is requested to create a certain file, e.g. `100.1.sai`, the wildcard `{sample}` matches `100` and `{group}` matches `1`, so that the input file `100.1.fastq` is expected. Multiple wildcards may in some cases introduce the problem of ambigous matches. To avoid such problems, wildcards can be restricted to particular regular expressions. In this case, to force the greedy matching to the intended solution, `{group}` could be replaced by `{group,[12]}`, such that the regular expression `[12]` is used instead of `.+`. The regular expression syntax is that of Pythons `re`

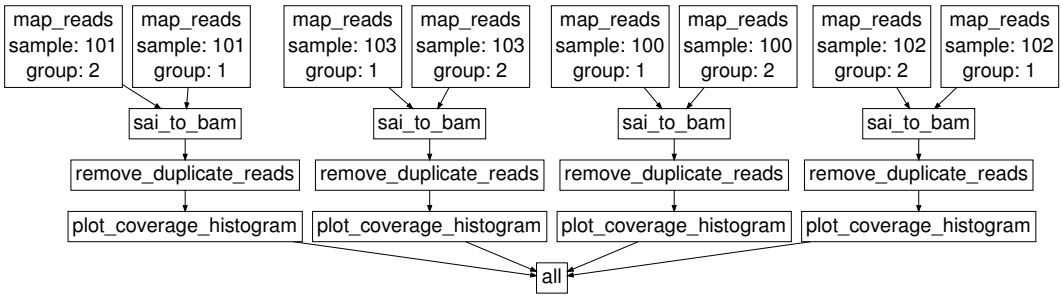

Figure 1 Directed acyclic graph of jobs for the example workflow from Listing 1.

module, which is almost identical to PERL 5's regular expressions, except for some corner cases.

3 The Snakemake Execution Engine

When Snakemake is invoked with a Snakefile, it determines a plan of rule executions that are needed to create the requested output. We call the planned execution of a rule a *job*. Since one rule can be executed for multiple wildcard values, more than one job for each rule is possible. A job may need certain input files that are created by other jobs, which gives rise to a directed acyclic graph (DAG) that represents the execution plan (see Figure 1). The nodes of the DAG are jobs, and a directed edge between job A and B means that the rule underlying job B needs the output of job A as an input file. A path in the DAG represents a sequence of jobs that have to be executed serially. Importantly, two disjoint paths in the DAG can be executed independently from each other, i.e., in parallel.

On top of per-job parallelism, the number of threads that a single job uses can be specified in a rule, via the parameter `threads` (line 12 of Listing 1), which defaults to 1 if it is not specified. The number of threads can be accessed in the shell command by the variable `threads` and thus passed to external tools that support multithreading. Snakemake tries to maximize parallelism (i.e. minimize the number of idle cores) up to a given threshold of available CPU cores. Considering that each job can use one or more threads as explained above, the parallel job scheduling problem can be cast as follows.

▶ **Definition 2** (Parallel Job Scheduling)**.** Let J be the set of jobs that are ready to execute (i.e. do not depend on missing input files), and t_j be the number of threads for job $j \in J$. Let T be the given threshold of available cores; if a job requests more threads (i.e. $t_j > T$), the threads are reduced to T. Thus the problem is to find E^* among all $E \subseteq J$ such that $\sum_{j \in E} \min(t_j, T)$ is maximized while the sum remains bounded by the number of currently idle cores.

The parallel job scheduling problem is a classical binary knapsack problem and can be solved by dynamic programming in pseudo-polynomial time. Given current CPU core numbers, the solution is instantaneous. The problem is solved every time a new job becomes ready to execute and idle cores exist. Snakemake first executes the jobs in the solution E^*.

Solving above problem to schedule jobs allows Snakemake to scale to environments with a hard limit of used CPU cores, e.g., when multiple users share the same compute server, by choosing T appropriately. Further, using only as many threads as there are cores available can be beneficial for performance since it reduces the amount of context switching.

Apart from running on single machines, Snakemake contains a generic mechanism that allows the execution of jobs on a batch system or a compute cluster engine. For this, a submit command that handles shell scripts (e.g. `qsub`) and a shared file system accessible by all cluster nodes has to be available. Snakemake compiles every job into a shell script and submits it with the given command once it is ready to execute. It then queries about the existence of a certain guard file that indicates that the job is finished in regular intervals.

Per default, Snakemake only executes rules if the output files are not present or the modification time of the input files is newer. Together with the automatic deletion of output files from incomplete rule executions (e.g. due to a failing shell command), this enables Snakemake to avoid duplicate work when resuming workflows.

To analyse the workflow, Snakemake provides options to perform a dry-run without actual execution of jobs, give the reason for each executed job and print the DAG to the *graphviz* `.dot` format [1] for visualization (see Figure 1).

4 Conclusion

Snakemake is a pythonic workflow system inspired by GNU Make, that allows to define a workflow in terms of rules that create output files from input files and thereby automatically handles dependencies and parallelization. It is the first workflow system to support multiple named wildcards and output files in rules. We show that this is especially useful in bioinformatics workflows. Further, Snakemake offers a simple Python-like syntax with high readability, and allows to efficiently intermix Python and shell commands. This removes the need to write external shell or PERL scripts for simple format conversions, and additionally allows to run complex algorithms within the workflow. Further, this allows to make use of web services, e.g. using specialized libraries or the built-in Python http client and XML parsing facilities.

Our approach exemplifies an automaton-based implementation of a domain-specific language without the need to generate a full parser or creating syntactical overhead by overloading language features.

By optimizing the schedule of jobs against a given threshold of available cores while taking also intra-job parallelism into account, a Snakefile scales from single core workstations over multi-core servers to compute clusters of different architectures, without the need to modify the workflow.

Acknowledgements We thank Tobias Marschall (CWI Amsterdam) and Marcel Martin (TU Dortmund) for their tremendously helpful testing work, feature requests and comments.

References

1 Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

2 Jeremy Goecks, Anton Nekrutenko, James Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, 2010.

3 Leo Goodstadt. Ruffus: A lightweight python library for computational pipelines. *Bioinformatics*, 26(21):2778–2779, November 2010.

4 Florian Halbritter, Harsh J. Vaidya, and Simon R. Tomlinson. GeneProf: analysis of high-throughput sequencing experiments. *Nature Methods*, 9(1):7–8, December 2011.

5 Shawn Hoon, Kiran Kumar Ratnapu, Jer-Ming Chia, Balamurugan Kumarasamy, Xiao Juguang, Michele Clamp, Arne Stabenau, Simon Potter, Laura Clarke, and Elia Stupka. Biopipe: A flexible framework for Protocol-Based bioinformatics analysis. *Genome Research*, 13(8):1904–1915, August 2003.

6 J.D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science Engineering*, 9(3):90 –95, June 2007.

7 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009.

8 Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, August 2009.

9 Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The genome analysis toolkit: A MapReduce framework for analyzing Next-Generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, September 2010.

10 Matthew Meyerson, Stacey Gabriel, and Gad Getz. Advances in understanding cancer genomes through second-generation sequencing. *Nature Reviews Genetics*, 11(10):685–696, September 2010.

11 Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

12 Simon P Sadedin, Bernard Pope, and Alicia Oshlack. Bpipe: A tool for running and managing bioinformatics pipelines. *Bioinformatics*, April 2012.

13 Richard M. Stallman and Roland McGrath. *GNU Make - A Program for Directing Recompilation*. 1991.

14 Masahiro Tanaka and Osamu Tatebe. Pwrake: a parallel and distributed flexible workflow management tool for wide-area data intensive computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, page 356–359, New York, NY, USA, 2010. ACM.

15 K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S Jun, and J. Tsujii. Design and implementation of GXP make - a workflow system based on make. *Future Generation Computer Systems*, 2011.