Workflow management for data analysis with GNU Guix

Roel Janssen

June 9, 2016

Abstract

Combining programs to perform more powerful actions using scripting languages seems a good idea, until portability and parallel execution on computing clusters become the main concerns of the script. This paper attempts to build a domain-specific language for workflow management built on top of GNU Guix's language for package management to provide portable, shareable and reproducible workflow descriptions that can run on cluster computing setups.

Contents

1 Introduction

Users operate computers by running computer programs. When using computers to analyze data sets, we use a variety of small tools together to build a new abstraction layer to compute. A functional package manager takes care of making these programs available to users in a portable and reproducible way. [1] But how do we describe our use of the programs to get from an initial dataset to a specific dataset from which we draw conclusions? More importantly, how do we reproduce such data transformations at a later point in time?

1.1 Scripting languages

We could simply write each command invocation in a *script*, and possibly enhance it with variables and looping constructs to make it easier to modify. Of course we can only do this when we take the availability of the interpreter for the scripting language for granted.

Due to differences in operating system environments, writing portable scripts becomes as complex as writing full programs. The attempt of GNU Autotools[1] to enable developers to generate portable build configuration scripts serves a fine example of the complexity involved.

Inevitably, writing portable scripts involves dealing with deployment and execution differences among software distributions. How do we ensure the system provides all programs we use in the script? Can we use multiple computing nodes to run parts of the scripts in parallel? Will the script survive system upgrades?

1.2 How, what and why

Furthermore, we would explain the script's inner workings to another developer by expressing a different perspective of the script; its structure in terms of processes that make up the whole. So, in addition to expressing the *procedure* that describes *how* to achieve a certain result, we would like to describe the structure of information with which we express *what* parts work together and *why* they do so.

We describe the context of a procedure (in a structural relational perspective) and the actual "how-to" instructions as a *process*, which provides insight into the *what* perspective. We further abstract a set of processes connected by their input and output properties, or related practical use in a *workflow*, which provides insight into the *why* perspective. With a *workflow language*, we attempt to elegantly describe the relationships between processes and their procedures in order to determine an efficient way to execute the entire workflow.

Workflow execution programs need to take care of the environment of the programs, the order in which programs should run, and how the programs will run efficiently across multiple compute nodes in a computing cluster. We call the combination of these subjects *workflow management.*

In this document, we pursue a declarative approach to defining workflows, by extending GNU Guix's language (implemented in Scheme) for describing packages [2]. By doing so, we can describe processes with such a great precision that we can actually map the output of a process to the source code of individual programs.

[1] `https://gnu.org/s/autoconf/`

2 Motivation

We attempt to create a single solution to the challenges described below with a workflow management extension to GNU Guix:

- **Portable and reproducible software deployment**:
  Obtaining similar or even identical results starts with using identical software environments.
- **Shareable, portable and reproducible workflow process descriptions**:
  Enable sharing of recipes to run a program and its run-time configurable settings to obtain a specific result.
- **Integration with multi-computer execution models**:
  In bioinformatics, the programs run on high-performance computing environments because of the size of the data sets.

In *Software deployment for reproducible, multi-user software environments* [1] we found an extensible programming language for describing reproducible software deployment. Extending this language with workflow management effectively provides a single solution to the challenges stated above.

2.1 Portability and reproducibility among system distributions

In Free Software we dealt with portability among system distributions by creating a common set of programs that form the basis for all distributions and a set of macros to deal with the remaining differences.

For reproducible software deployment, we can use a functional package manager with which we can reproduce the programs that make up a software environment on all major software distributions based on Linux.

GNU Guix — an implementation of a functional package manager with an extensible language to deal with the build and deployment processes of software programs — provides software deployment features with a declarative language, which makes it a good starting point for a workflow management language.

2.2 Shareable, portable and reproducible workflows

A portable self-contained software environment (provided by a functional package manager) makes writing portable scripts easy. By using a language that provides meta-linguistic abstraction features, we can implement a mechanism for (parallel) execution of programs on multiple computers, so that the procedures do not have to deal with this complexity.

2.3 Declarative language properties

A *script* only provides an answer to the *how*, but leaves the *what* and *why* aspects completely in the dark. The declarative approach of the GNU Guix package management language displays the capability of describing the *what* and *why* aspects.

Automation requires understanding the computational problem in various contexts (*how*, *what* and *why*). Existing workflow management programs attempt to provide insight in the *what* and *why* aspects, but leave important details like software deployment up to external programs.

Designing an efficient mechanism or *framework* to handle these complexities could simplify the code that actually does the work. The possibility for meta-linguistic abstraction in LISP and Scheme [3] allows for a solution closer to what we humans can understand: a domain-specific language for workflow definitions.

The domain-specific language for package management of GNU Guix [2] serves as a basis and an inspiration for a simple yet effective declarative language for workflow management.

3 Workflow Description Language

First, by building on the purely functional deployment model [4] to deploy programs and their entire environment we attempt to maximize the precision of expressing workflows.

We do this by extending GNU Guix — an implementation of a purely functional package manager. In the *workflow description language*, a user only needs to specify which packages a process requires. GNU Guix can then build and deploy these packages and their (recursive) dependencies.

Secondly, the declarative nature of the language we define helps providing insight into the *what* aspect of workflow management by exposing the relationships between processes. We can derive the order in which processes must execute, and in extension, which processes can run in parallel to each other from a set of pairs in the form of `(A B)` where `A` depends on the successful completion of `B`.

Thirdly, because GNU Guix can provide a self-contained run-time environment for a procedure, we can distribute its execution to other computers in a network. We leave the coordination of remote code execution to existing job control systems.

We implement *record types* in Scheme, similar to those used for describing packages, to provide a format for expressing properties in a structured way. This allows for a declarative approach to defining processes and workflows, which makes our language a *description* language.

3.1 Processes

A *process* describes a computable task to perform. This essentially consists of two components: the transformation according to a specified algorithm and the implementation of a program to perform the task. By distinguishing between the *algorithm* (mathematical steps) and the *implementation* (computer program) used to perform the task, we can more easily find differences between what *should happen*, and what *actually happens*.

Processes live in the context of a workflow like functions live in the context of a program. In fact, we implement processes as functions in our workflow language, taking input as a parameter and returning the desired output.

Here we find an essential difference between a `package` and a `process`; with a package, we know about all variables in advance, while with processes, the input and output variables could differ each time we run it.

3.1.1 Example process definition

```
(define (rnaseq-fastq-quality-control in out)
  (process
    (name "rnaseq-fastq-quality-control")
    (version "1.0")
    (environment
     `(("fastqc" ,fastqc-bin-0.11.4)))
    (input in)
    (output (string-append out "/" name))
    (procedure
     (script
      (interpreter 'guile)
      (source
        (let ((sample-files (find-files in #:directories? #f)))
         `(begin
            ;; Create output directories.
            (unless (access? ,out F_OK) (mkdir ,out))
            (unless (access? ,output F_OK) (mkdir ,output))
            ;; Perform the analysis step.
            (map (lambda (file)
                   (when (string-suffix? ".fastq.gz" file)
                     (system* "fastqc" "-q" file "-o" ,output)))
                 ',sample-files))))))
    (synopsis "Generate quality control reports for FastQ files")
    (description "This process generates a quality control report
for a single FastQ file without any special options passed to
FastQC.")))
```

3.1.2 Executing the process's procedure

The `procedure` contains code to perform, and the `environment` describes the environment in which this code can successfully run. In a single expression, we captured the run-time environment and a way to use that environment to run the exact program including its dependencies as described by the package definitions.

When distributing code, we cannot assume the other machine has an identical software environment. So, before we attempt to run the Scheme code, we must create a suitable software environment for the procedure to run successfully. Setting the environment variables to the values suggested by the output of GNU Guix after installing a package takes care of this. We can obtain these values at a later time by by running:

```
guix package --search-paths
```

By using the intermediary form of a shell script, we can execute the code on any computing cluster. From a `process` description we can derive the contents of the shell script.

For the example `process` defined in section 3.1.1, the workflow execution engine can generate the following shell script:

```
#!/gnu/store/7cdd8s466qyjh64m0byq0rz9gk1jid40-bash-4.3.42/bin/sh
export PATH="/hpc/shared_profiles/rnaseq/bin:/hpc/shared_profiles/guile/bin"
```

```
guile -c '(begin
            (unless (access? "/hpc/example-out" F_OK)
              (mkdir "/hpc/example-out"))
            (unless (access? "/hpc/example-out/run-1" F_OK)
              (mkdir "/hpc/example-out/run-1"))
            (map (lambda (file)
                   (when (string-suffix? ".fastq.gz" file)
                     (system* "fastqc" "-q" file "-o" ,output)))
                 \'("/hpc/example-in/sample_R1_001.fastq.gz"
                    "/hpc/example-in/sample_R1_002.fastq.gz")))'
```

3.2 Workflows

When defining workflows, we want to express the order (*flow*) in which processes (*work*) should run. When processes can run in parallel, we would like to do so. To make this a computational problem, we need to describe the processes and their relationship to each other.

3.2.1 Example workflow definition

Taken from an existing Perl script, we can abstract the individual processes and describe their relationship to each other. We do not have to take care of writing our own parallel execution mechanism, because the workflow execution engine can compute this.

```
(define (rnaseq-pipeline in out)
  (workflow
   (name "rnaseq-pipeline")
   (version "1.0")
   (input in)
   (output (string-append
            out "/" name "-"
            (date->string (current-date) "~Y-~m-~d")))
   (processes
    '(rnaseq-initialize
      rnaseq-fastq-quality-control
      rnaseq-align
      rnaseq-add-read-groups
      rnaseq-index
      rnaseq-feature-readcount
      rnaseq-collect-alignment-metrics
      rnaseq-merge-read-features
      rnaseq-compute-rpkm-values
      rnaseq-normalize-read-counts
      rnaseq-differential-expression))
   (restrictions
    `((,rnaseq-fastq-quality-control ,rnaseq-initialize)
      (,rnaseq-align ,rnaseq-initialize)
      (,rnaseq-add-read-groups ,rnaseq-align)
      (,rnaseq-index ,rnaseq-add-read-groups)
      (,rnaseq-collect-alignment-metrics ,rnaseq-index)
      (,rnaseq-feature-readcount ,rnaseq-index)
      (,rnaseq-merge-read-features ,rnaseq-feature-readcount)
      (,rnaseq-compute-rpkm-values ,rnaseq-merge-read-features)
      (,rnaseq-normalize-read-counts ,rnaseq-merge-read-features)
      (,rnaseq-differential-expression ,rnaseq-merge-read-features)
     ))
   (synopsis "RNA sequencing pipeline used at the UMCU")
   (description "The RNAseq pipeline can do quality control on
FastQ and BAM files; align reads against a reference genome; count
reads in features; normalize read counts; calculate RPKMs and
perform DE analysis of standard designs.")))
```

From this example we can generate the overview graph displayed in Figure 1 and a sequential execution order graph displayed in Figure 2. A parallel execution order graph would look similar to the upside-down version of the overview graph.

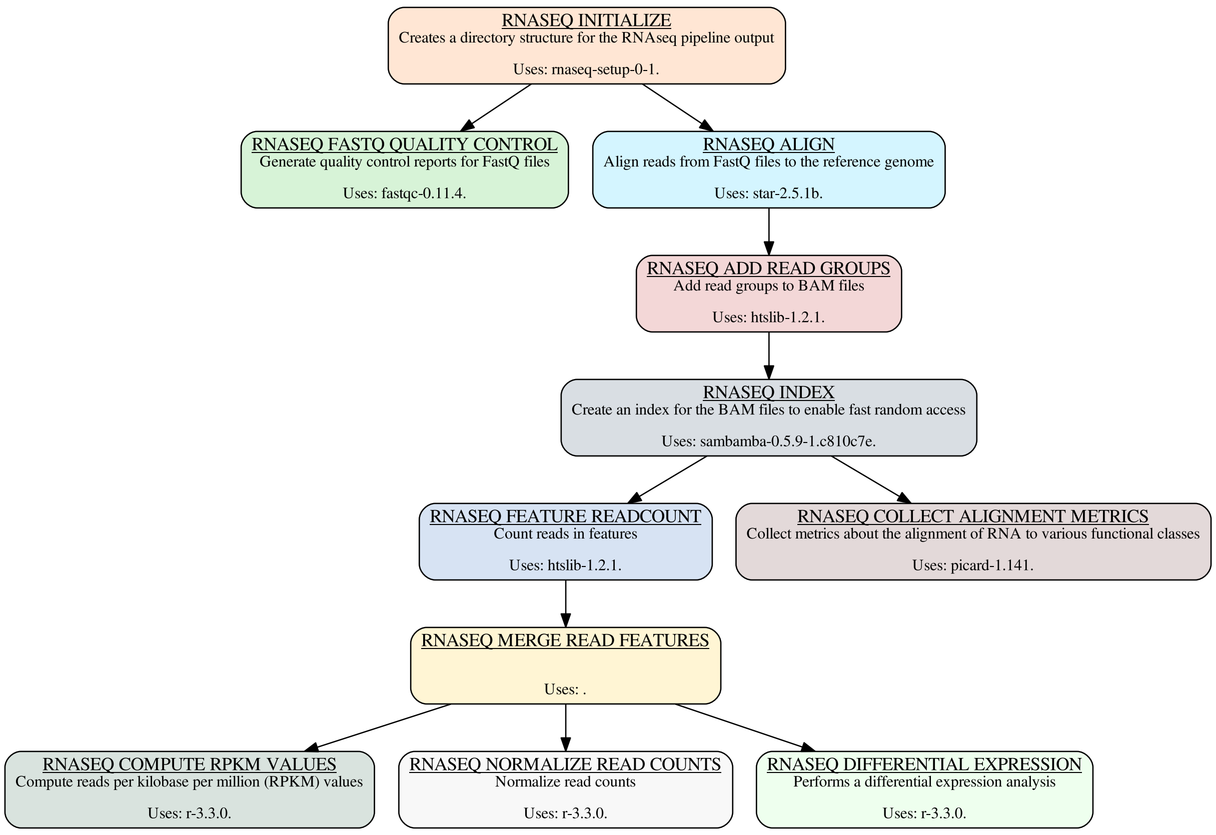
RNASEQ INITIALIZE
Creates a directory structure for the RNAseq pipeline output
Uses: rnaseq-setup-0-1.
RNASEQ FASTQ QUALITY CONTROL
Generate quality control reports for FastQ files
Uses: fastqc-0.11.4.
RNASEQ ALIGN
Align reads from FastQ files to the reference genome
Uses: star-2.5.1b.
RNASEQ ADD READ GROUPS
Add read groups to BAM files
Uses: htslib-1.2.1.
RNASEQ INDEX
Create an index for the BAM files to enable fast random access
Uses: sambamba-0.5.9-1.c810c7e.
RNASEQ FEATURE READCOUNT
Count reads in features
Uses: htslib-1.2.1.
RNASEQ COLLECT ALIGNMENT METRICS
Collect metrics about the alignment of RNA to various functional classes
Uses: picard-1.141.
RNASEQ MERGE READ FEATURES
Uses: .
RNASEQ COMPUTE RPKM VALUES
Compute reads per kilobase per million (RPKM) values
Uses: r-3.3.0.
RNASEQ NORMALIZE READ COUNTS
Normalize read counts
Uses: r-3.3.0.
RNASEQ DIFFERENTIAL EXPRESSION
Performs a differential expression analysis
Uses: r-3.3.0.

Figure 1: *Process overview as described by the code for an RNA sequencing pipeline. In a dependency graph, we would draw the arrows exactly the other way, which tells us the parallel execution graph.*

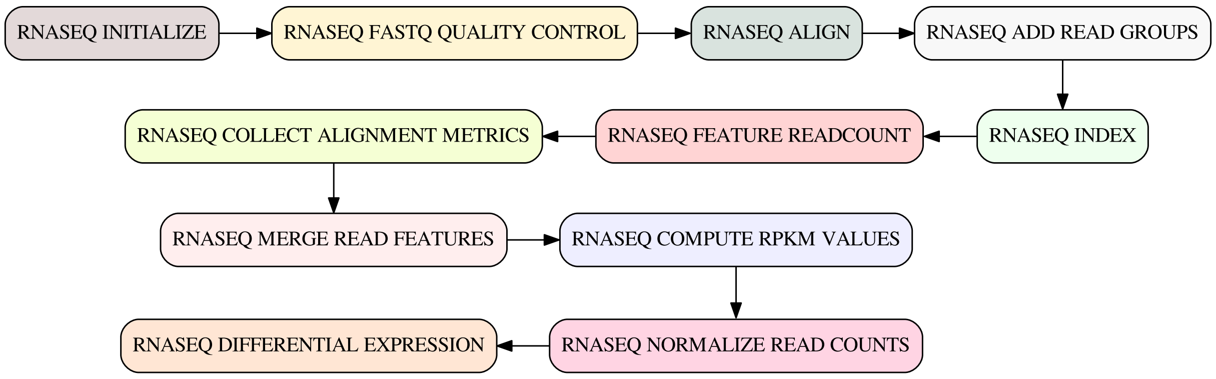
RNASEQ INITIALIZE
RNASEQ FASTQ QUALITY CONTROL
RNASEQ ALIGN
RNASEQ ADD READ GROUPS
RNASEQ INDEX
RNASEQ FEATURE READCOUNT
RNASEQ COLLECT ALIGNMENT METRICS
RNASEQ MERGE READ FEATURES
RNASEQ COMPUTE RPKM VALUES
RNASEQ NORMALIZE READ COUNTS
RNASEQ DIFFERENTIAL EXPRESSION

Figure 2: *A graph displaying the sequential order in which processes may execute to resolve all dependency constraints. An implementation for computing this order can be found in appendix A "Dependency pairs to exection order".*

3.2.2 Invoking workflows with GNU Guix

Remember that we implemented both the workflow and the processes as functions taking two arguments: `input` and `output`. As a result, we can specify where to find the input data and where to store the output data.

```
guix workflow --execute=rnaseq-pipeline \
              --input=/home/roel/pipelines/rnaseq-in \
              --output=/home/roel/pipelines/rnaseq-out
```

3.3 Computing the execution order

From the list of dependency pairs, we can identify *free* processes, which do not rely on data from others. We provide an implementation for this functionality in Appendix A.

4 Integration with the `guix` command

GNU Guix provides an interface for users to do package management. In addition to a traditional package management interface to install, remove and upgrade packages, it provides subcommands to generate dependency graphs and edit package recipes. Users of workflow execution programs expect similar functionality for their workflow descriptions.

Therefore, full integration of workflow management in GNU Guix involves implementing the relevant subcommands like `graph` for workflows so that users can run `guix graph my-workflow` and get the graph of a workflow as displayed in Figure 1.

4.1 Lay-out of existing functionality

GNU Guix's user interface consists of multiple Scheme modules connected by a 'main' module that directs a user's request to run the relevant Scheme functions. With Scheme — a dialect of Lisp — we build language abstractions to produce more suitable language constructs for a specific situation. For example, the `(guix packages)` module provides a `package` record type that allows the developer to describe a package without needing to write any of the actual run-time code for GNU Guix. The declarative language knows how to transform a package recipe into the instructions to pass to the build daemon.

Figure 3 displays a subset of the Scheme modules that make GNU Guix including the Scheme language interpreter for which we can generate dependency graph with GNU Guix itself.

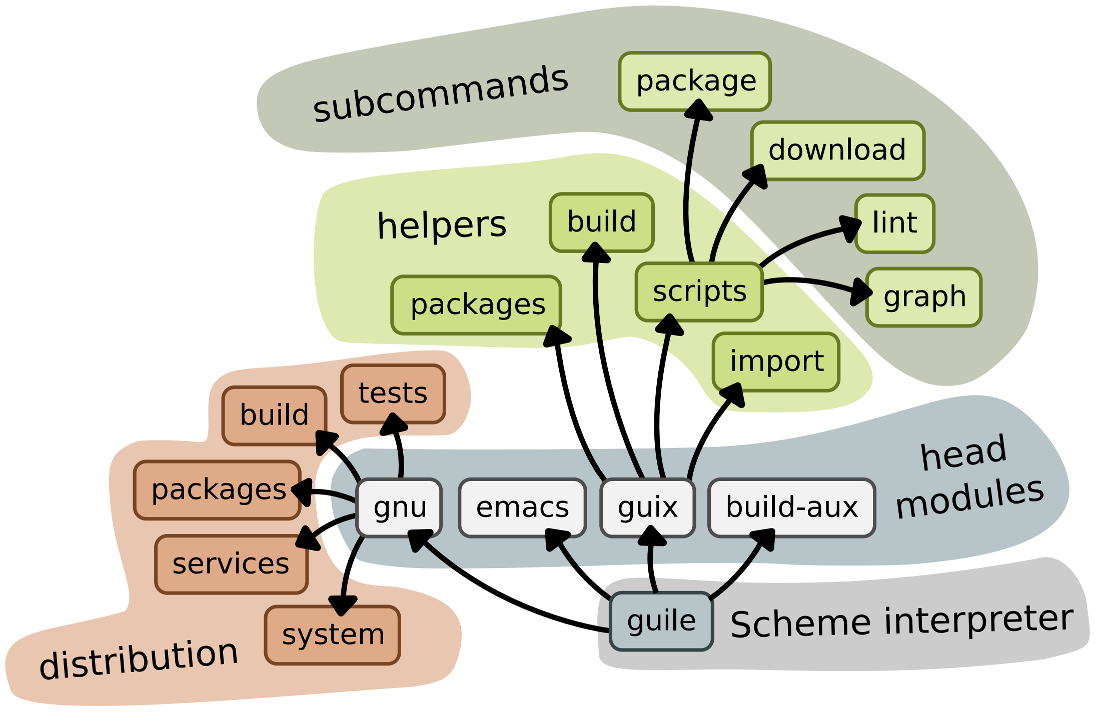

Figure 3: *The general structure of Scheme modules that make up GNU Guix.*

4.2 User interface

To implement the workflow language subcommand, we need to add the module (`guix scripts workflow`). Furthermore, to integrate the workflow language with the `graph` and `lint` subcommands, we need to modify the corresponding (`guix graph`) and (`guix lint`) modules.

Adding a workflow recipe involves creating a (`gnu workflows ...`) module, similar to the way to add a package recipe by creating a (`gnu packages ...`) module.

4.3 Workflow execution engine

For the workflow language to execute the `workflow` descriptions, we must implement a workflow execution engine.

For this workflow execution engine we need to implement record types for `workflow` and `process` in (`guix workflows`) and a dependency resolver which we do in (`guix workflow execution-order`).

The proof-of-concept implementation contains additional modules for helper functions to generate shell scripts from a `process` record in (`guix workflow execution-helper`).

4.4 Lay-out with the added functionality

Figure 4 displays the new and affected modules to implement workflow management to GNU Guix.

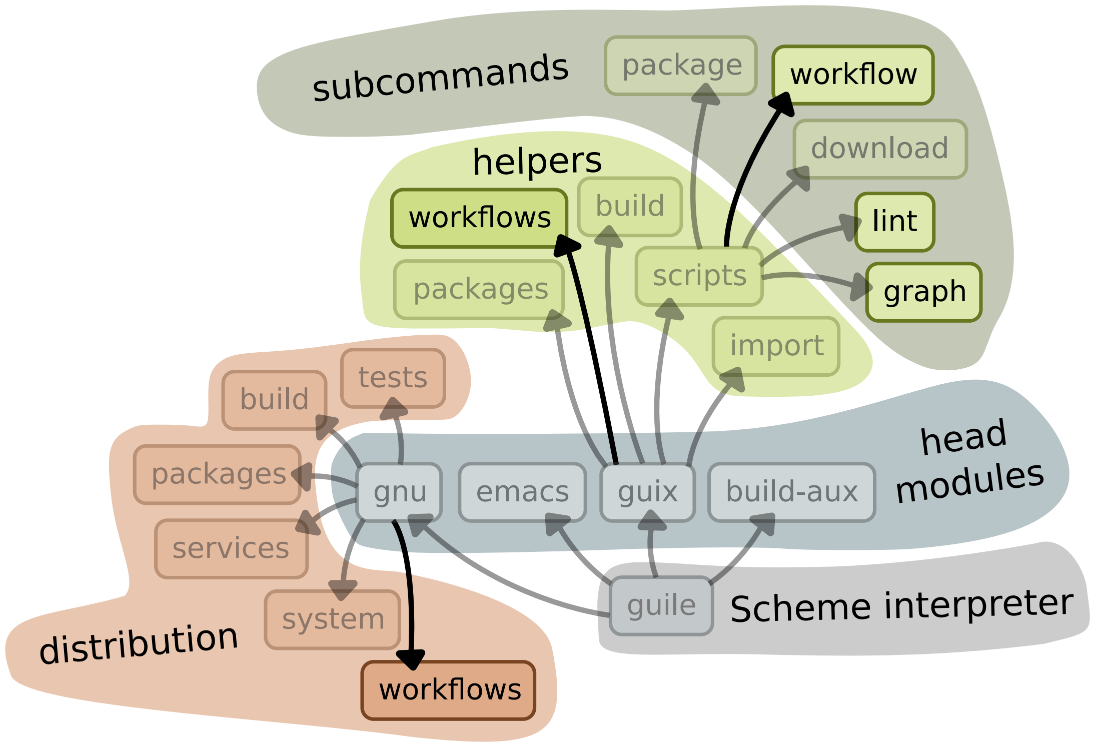

Figure 4: *The impact of adding workflow management to GNU Guix. Additional modules to implement and existing modules to adjust.*

As explained in section 4.2 *User interface* and section 4.3 *Workflow execution engine*, we need to implement three new modules: `(gnu workflows)`, `(guix workflows)` and `(guix scripts workflow)`. Integrating existing commands with workflow management requires adjusting `(guix scripts lint)` and `(guix scripts graph)`.

5 Conclusion

When building pipelines for data analysis, GNU Guix can provide both package management and workflow management. Implementing both packages and workflows as Scheme record types enables users to refer to a precise package and the corresponding dependency graph of that package when defining workflows.

Because GNU Guix provides a self-contained environment, the execution of processes in a workflow becomes as simple as running a command without the burden of checking for missing programs or different versions of programs.

Describing the relationships between processes from which a workflow execution engine can compute the execution order reduces the amount of code required to run a workflow correctly.

The sum of these simplifications make an elegant language extension to GNU Guix to define workflows.

6 Discussion

The reliance of one process on another seems expressed redundantly. We could detect whether a process depends on another by looking for the `process-output` function, which provides a way to say "the input of process A uses the output of process B". So, to gather dependency information from `process` records, we can look for inputs that use this function.

We can then describe dependency pairs. When process A depends on process B, we define the pair `(A B)`. When process A also depends on process C, we simply define another pair `(A C)` so that we get a list of pairs `((A B) (A C))`.

For the direct dependencies (where the input depends on the output of another process), we could compute the dependency pairs automatically.

Appendix A Dependency pairs to exection order

This appendix includes a possible implementation of a dependency solver written in Guile Scheme[2]. The `execution-order` function provides a simple interface to compute the execution order for a given set of processes and their restrictions.

```
(define* (process-depends-on process dependencies
                                    #:optional (depend-list '()))
  "Returns the list of processes PROCESS depends on."
  (if (null? dependencies)
    depend-list
    (let ((item (car dependencies)))
      (if (equal? process (car item))
        (process-depends-on process (cdr dependencies)
                            (append depend-list (cdr item)))
        (process-depends-on process (cdr dependencies)
                                    depend-list)))))

(define* (process-needed-by process dependencies
                                    #:optional (depend-list '()))
  "Returns the dependency pairs needed by PROCESS."
  (if (null? dependencies)
    depend-list
    (let ((item (car dependencies)))
      (if (equal? (list process) (cdr item))
        (process-needed-by process
                           (delete item dependencies)
                           (append (list item) depend-list))
        (process-needed-by process
                           (delete item dependencies)
                           depend-list)))))

(define* (compute-free-points processes dependencies
                              #:optional (free-points '()))
  "Returns a PROCESS that can be used to start the execution at."
  (if (null? processes)
    free-points
    (let ((process (car processes)))
```

[2] https://gnu.org/s/guile

```
      (if (null? (process-depends-on process dependencies))
        (compute-free-points (cdr processes) dependencies
                             (append free-points (list process)))
        (compute-free-points (cdr processes) dependencies
                             free-points)))))

(define (reduce-dependencies processes dependencies)
  "Removes dependencies on resolved PROCESSES."
  (if (null? processes)
    dependencies
    (reduce-dependencies
     (cdr processes)
     (lset-difference eq? dependencies
                      (process-needed-by (car processes)
                                         dependencies)))))

(define* (execution-order processes dependencies
                                    #:optional (order '()))
  "Returns the list of PROCESSES, re-ordered so it can be executed
and adhere to the dependencies provided in DEPENDENCIES."
  (if (null? processes)
    (reverse order)
    (let* ((resolvable (compute-free-points processes
                                            dependencies))
           (leftovers (lset-difference eq? processes resolvable)))
      (if (null? resolvable)
          #f
          (execution-order leftovers
                           (reduce-dependencies resolvable
                                                dependencies)
                           (append resolvable order))))))
```

References

[1] Roel Janssen. Software deployment for reproducible, multi-user software environments. 2016.

[2] Ludovic Courtès. Functional Package Management with Guix. In *European Lisp Symposium*, Madrid, Spain, June 2013.

[3] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[4] Eelco Dolstra. *The Purely Functional Software Deployment Model.* Institute for Programming research and Algorithmics, 2006.