StarFlow: A Script-Centric Data Analysis Environment

Elaine Angelino, Daniel Yamins, and Margo Seltzer

School of Engineering and Applied Sciences, Harvard University,
33 Oxford St., Cambridge, MA 02138
{elaine,margo}@eecs.harvard.edu, yamins@fas.harvard.edu
http://www.eecs.harvard.edu/~margo/

Abstract. We introduce StarFlow, a script-centric environment for data analysis. StarFlow has four main features: (1) extraction of control and data-flow dependencies through a novel combination of static analysis, dynamic runtime analysis, and user annotations, (2) command-line tools for exploring and propagating changes through the resulting dependency network, (3) support for workflow abstractions enabling robust parallel executions of complex analysis pipelines, and (4) a seamless interface with the Python scripting language. We describe real applications of StarFlow, including automatic parallelization of complex workflows in the cloud.

Keywords: automatic parallelization, automatic updating, computational workflows, control flow, data-flow, data analysis, dependency tracking, provenance, Python, workflow abstraction.

1 Introduction

Many people analyze data by writing pipelines of scripts: short programs written in high-level languages such as Python that parse input, call numerical analysis routines, and write output.

Scripts plus data files are powerful because they are very flexible: they allow users to mix and match many kinds of data formats and analysis routines, output files where convenient, write code that performs complicated computational tasks, re-use code in different places, and put related functions into the same file. While script pipelines are less rigid than databases, they are more prone to disorganization. Scripts and data live in conventional file systems, where dependency relationships are exposed only at runtime, and provenance of data is easily lost.

The data analysis work cycle consists of basic actions: create an analysis pipeline and execute its initial run, modify input data or an analysis function and propagate the change, add an analysis function and re-execute the pipeline, and create related pipelines based on an abstract workflow. In this context, it is difficult and annoying to remember what functions were called with what parameters to produce what files, to re-run a long chain of downstream scripts when an upstream data file or script is modified, to capture repeated patterns of analysis, to parallelize execution, and to communicate or replicate analyses.

D.L. McGuinness, J.R. Michaelis, and L. Moreau (Eds.): IPAW 2010, LNCS 6378, pp. 236–250, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Data analysts who write scripts are thus confronted by fundamental data management challenges: identifying dependencies, propagating changes, parallelizing work, sharing data and code, and archiving relevant information. Dependency tracking and workflow management tools would help them by making recomputation automatic and efficient and by making sharing easier.

These programmers are an important and unique user group. They are comfortable with and depend on writing code, and as a result are unwilling to depend on tools that depart from the scripting environment. At the same time, they are not sophisticated software engineers; they write code as a means to produce analytic results, not to produce code as an end result.

A workflow tool for these users must integrate in a simple way with the existing scripting environment. By focusing on these users in this environment, we have designed a dependency tracking system and workflow engine with novel features. A key observation is that scripts plus data files already contain workflows in the sense that they implicitly describe a dependency graph. This insight motivates both design constraints and a unified and flexible framework for managing dependencies across multiple workflows that may exist separately or overlap within a user's file system.

Dependency tracking systems can explicitly capture the provenance of data analysis and enable workflow tools for managing, generating and executing analysis pipelines. Existing tools track dependencies by combining dynamic runtime analysis, static analysis, and/or user annotations; their specific choices restrict when and what dependencies can be extracted and thus when and how they can be used to drive actions. User annotations plus static analysis extract control flow prior to runtime execution, enabling automatic parallelization. Even without annotations, dynamic analysis extracts both information and control flow. Whether dynamic or static, control flow dependency tracking at the level of *functions* facilitates incremental recomputation.

StarFlow strategically uses all three methods of dependency tracking while integrating seamlessly with a script-based programming environment[1]. This unique combination of features makes StarFlow widely applicable, from single-purpose analysis pipelines written "on the fly" to complex workflows in a high-performance computing environment.

Below, we introduce a design framework for data analysis workflow engines and describe existing implementations (§2). Next, we describe StarFlow's implementation (§3), user scenarios (§3.4), workflow abstraction and automatic parallelization of complex workflows in the cloud (§4).

2 Features of a Workflow Engine for Data Analysis

A workflow engine for data analysis can be evaluated by: (1) how and at what level of granularity it tracks dependency relationships between data and analysis functions, (2) what user actions it supports using those dependencies, (3)

[1] See http://bitbucket.org/dyamins/starflow/ for StarFlow source code and documentation.

whether and how it supports workflow abstraction, and (4) how it integrates with a programming environment. We use this framework to describe our design and to classify existing workflow tools (Table 2).

2.1 Tracking Dependencies

A set of scripts implies a dependency network of links between data and functions. A function may **depend on** file inputs, **create** file outputs, and **use** other functions (Fig. 1). There are three complementary sources of dependency information: user annotations, static code analysis, and dynamic runtime analysis. Each technique has strengths and weaknesses (Table 1).

User annotation of dependencies allows a workflow tool to be aware of dependencies without having to extract them, and is a widely used technique. The familiar Unix `make` utility requires that a user create a `Makefile`, explicitly specifying file targets, their dependencies, and commands transforming one to another. Although `Makefile`s are notoriously difficult to maintain, they are still the de facto standard way to specify source code dependencies. Workflow management systems also ask users to explicitly describe both information and control flow; there are many in the scientific (e.g. Galaxy[26], GenePattern[16], Kepler[18], Knime[3], Pegasus[9], Taverna[22], Vistrails[4]) and business (e.g. clario[5], Pentaho Data Integration [8]) communities. Their users construct workflows by connecting functional "nodes" with well-defined input/output types.

Static analysis of code can automate some of this manual annotation, but in the general case cannot completely capture information or control flow; these

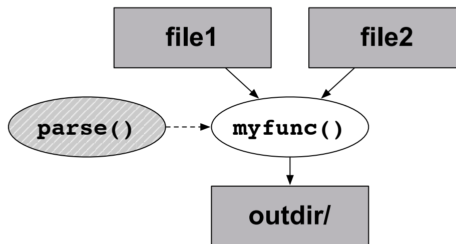

Fig. 1. The function `myfunc()` depends on input files 'file1' and 'file2', creates output directory 'outdir/', and uses the function `parse()`. Arrows are in the direction of information flow.

Table 1. Three complementary methods for tracking information flow and control flow: dynamic runtime analysis, static analysis of code, and user annotation

	Runtime analysis	**Static analysis**	**User annotation**
Information flow data dependencies	Accurate, but sometimes too late	Difficult	Acceptable if lightweight
Control flow functional dependencies	Accurate, but mostly unnecessary	Usually possible	Very annoying

Table 2. Comparison of data tracking implementations

	Runtime analysis	**Static analysis**	**User annotation**
`make`			Specify file targets, their file dependencies & executable commands in a `Makefile`
`make + Automake`	`depcomp` determines source file dependencies during compilation	When `depcomp` fails, `makedepend` determines source file dependencies	Specify C/C++ source files in a simplified `Makefile`
Workflow management systems			Specify node parameters & data flow in a GUI or flow language
`IncPy`	Modified Python interpreter tracks file I/O & function calls; memoizes function returns		
`StarFlow`	File I/O interception & stack trace inspection in the Python interpreter	Abstract syntax tree analysis of Python code tracks **function**-level control flow	Specify data flow & non-Python control flow directly in function definition lines

are Turing-undecidable problems. In practice, static analysis can often extract a highly accurate description of control flow. For example, `makedepend` augments the standard `make` utility by using static analysis to automatically extract C source code file dependencies. Static analysis can even extract dependencies at the level of functions because their syntax makes them easy to parse from the abstract syntax tree. Data-flow dependencies are difficult to extract because they are not explicit in the abstract syntax tree, e.g. they may be implicit in a concatenation of strings.

Dynamic analysis captures the actual information and control flow generated during runtime execution of scripts. File input/output interception captures data file dependencies, and stack trace inspection captures functional dependencies. Pure runtime systems use only dynamic runtime analysis. For example, provenance-aware storage systems (PASS) automatically track provenance at the level of files and processes dynamically at runtime [21], while `IncPy` is a modified Python interpreter that dynamically tracks file I/O and computational results at the level of function calls [13]. `Automake` is another dynamic analysis tool that automates the construction of `Makefiles` [24].

The **granularity** of dependency tracking determines what actions it can support. Notably, `make`-like tools track control flow at the level of *files*, but practical incremental recomputation requires tracking at the level of *functions*.

2.2 Using the Dependency Network

Knowing the dependency network supports three activities: dependency exploration, automatic change propagation, and pipeline extraction and sharing.

Dependency exploration involves querying the dependency network to understand where files and functions come from and their upstream and downstream dependencies. A query might concern only dependency structure (e.g., "On what Python modules does this output depend?") or could take into account other information, such as the file modification times of dependency targets relative to their sources (e.g., "Do I need to rerun this analysis?").

Automatic change propagation involves the use of a "smart" updating engine that queries the dependency network to support incremental recomputation; it updates targets by (re-)executing the minimal set of control flow components necessary. When the user invokes `make`, it examines the file modification times of targets relative to their dependencies to determine those in need of updating and executes the minimal sequence of necessary commands. The Panda project is developing a formalism and algorithms for provenance-based refresh in data-oriented workflows [14]. `IncPy`'s dynamic analysis and memoization facilitates fine-grain incremental recomputation.

Extraction and sharing of an analysis pipeline between users is facilitated by knowing its dependency network.

2.3 Workflow Abstraction

Once a user develops an analysis pipeline, she often needs to apply it to a potentially large number of similar analyses. If we view the overall pipeline as an abstract workflow, then each of these pipelines becomes an instance of that abstract workflow. Workflow environments differ in whether and how they allow users to represent abstract workflows, concrete instances or both. Those that do support workflow abstraction additionally differ in whether they support programmatic instantiation of concrete pipelines from abstract workflows. Scripting environments support but do not typically come with ready tools for workflow abstraction, while workflow management systems emphasize workflow abstraction but not necessarily programmatic instantiation.

2.4 Integration with the Programming Environment

There are two fundamental approaches to providing dependency tracking capabilities: make the workflow management system the center of the system or integrate dependency tracking into the programming environment. Workflow management systems tend to do the former while integrated development environments (IDEs) do the latter. It is often a design goal of workflow management systems to support novices who do not want to write programs [18]. For example, Taverna replaces a regular programming environment with a GUI for manipulating an XML-based flow language, Scufl [20]; users can also directly write in Scufl to annotate dependencies. IDE-based systems provide a unified interface for code developement that decreases the distance between where a user edits and executes scripts. Eclipse's C/C++ Development Tooling (CDT) IDE includes standard `make` build, plus a GUI for writing `Makefile`s and invoking `make` [12]. While the Chimera virtual data system is script-based, it requires use of a virtual data language (VDL) [11].

An important extension of a workflow engine's integration with the programming environment is its support for distributed computing on a grid or in the cloud. Users often have computational needs at multiple scales, from jobs they want to run a personal computer to high performance computing (HPC) problems; many of the scientific and business workflow tools mentioned already can be deployed in a variety of environments. Other workflow management solutions are specifically for distributed systems, such as Azkaban and Oozie for Apache Hadoop by LinkedIn and Yahoo!, respectively [7,15]. Some tools, including PiCloud and pomsets, specialize in workflow management for cloud services, e.g., Amazon's EC2 [10,23].

3 StarFlow

StarFlow is a data analysis environment that is script-centric, has `make`-like tools, tracks dependencies at the level of functions rather than files, and is constrained in scope to the level of a scripting language. Our implementation of the StarFlow workflow engine has four main features: (1) dependency tracking of both information and control flow via a novel combination of static analysis, dynamic analysis, and user annotations, (2) command-line tools supporting dependency exploration, automatic updating, and pipeline extraction and sharing, (3) workflow abstractions and concrete analysis pipeline instances, and (4) a seamless interface with Python. Although our initial implementation is for Python, our design principles and algorithms are broadly applicable. Sections §3.2 and §3.3 describe how StarFlow tracks and uses dependencies, and section §3.4 presents various usage scenarios.

3.1 Design Principles

A few basic principles guide StarFlow's design. First, users express dependencies only in their code. This design choice makes sharing dependency information a consequence of sharing code, and so these actions do not have to occur separately. Second, StarFlow is designed to place a minimal burden on the user, implying that any required annotations must be lightweight. Finally, StarFlow is for programmers who aren't software engineers, and so it is script-centric and simple.

3.2 Tracking Dependencies

StarFlow uses a combination of dynamic analysis, static analysis and user annotations to track data and functional dependencies.

User annotations. Although user annotations in StarFlow are purely optional, they enable parallelization and dependency querying before a script has ever been run. They also make sharing dependency provenance a side effect of sharing code. Such annotations are simple declarations within Python functions that expose the inputs and outputs of the function. For example,

```
def myfunc(depends_on=('file1','file2'), creates='outdir/'):
```

indicates that `('file1','file2')` and `'outdir/'` are the file names of the inputs and output of `myfunc`, respectively. The user can also annotate non-Python functional dependencies, e.g. a Perl script, with an analogous parameter, `uses`.

When functions specify input and output file namess via parameters, the user can write a simple one-line annotation to describe information flow. It is a Python decoration, indicated by `@activate`, consisting of two lambda expressions, one representing the inputs (`depends_on` annotations) and the other representing the outputs (`creates` annotations). Upon function invocation, the decoration maps the parameters to the appropriate lambda expression. For example, in

```
@activate(lambda x: (x[0], x[1]), lambda x: x[2])
def myfunc(infile1, infile2, outdir):
```

the first lambda represents the `depends_on` values, mapping to the first two parameters `(x[0] and x[1])`, while the second lambda represents the `creates` values, mapping to the third parameter `x[2]`. Thus, like in the previous example, `(infile1, infile2)` are the inputs and `outdir` is the output of `myfunc()`. This is particularly useful for workflow abstraction (§4) but also for any function whose inputs and/or outputs are specified at runtime.

Static analysis. StarFlow uses static code analysis to determine most control flow prior to runtime execution. First, it examines `import` statements to determine what external modules a script depends on. Then, it uses Python's built-in `compiler.ast` module to access the abstract syntax tree to determine the functional dependencies within a module. Static analysis cannot determine conditional control flow, and StarFlow has different methods for approximating dependencies in different scenarios. For example, it extracts control flow in all conditional clauses, but never extracts control flow in an `eval` statement.

Dynamic analysis. During runtime, for each function executed at the top of the Python stack, StarFlow uses `sys.settrace` to set a trace function. StarFlow walks the stack and examines all function calls to extract control flow and intercepts file I/O functions to extract information flow. This produces a trace of all function calls specifying the stacks where they were invoked, as well as what I/O operations were performed and what files they involved. By setting an environment variable, the user can control how StarFlow uses runtime dependencies: they can be simply logged, or they can be compared to the results of static analysis and user annotations to check for consistency.

Dependency representation. As a result of using the three methods of dependency tracking, StarFlow determines the dependency network; we describe its representation here.

`LoadLiveModules()` takes a set of directories and recursively determines all the Python modules inside those directories. The user can pass `LoadLiveModules()` a set of regular expression filters to conditionally select modules and functions and can maintain a `LiveModules` configuration file to set the default input.

`LinksFromOperations()` determines the dependency network corresponding to a list of Python modules. It uses static code analysis and extracts user annotations to construct the dependency list including both information and control flow. `LinksFromOperations()` caches the compiled bytecode of user-generated functions so that irrelevant changes, such as edits to comments or changes to unrelated functions in the same module, do not result in changes to the dependency network. It returns the `LinkList`, a table whose records correspond to dependency links and whose columns describe the links. For example, this `LinkList` describes the dependencies in Figure 1:

Link Type	Link Source	Source File	Link Target	Target File	Update Script	Update ScriptFile
DependsOn	file1	file1	`myfunc()`	mymodule.py	None	mymodule.py
DependsOn	file2	file2	`myfunc()`	mymodule.py	None	mymodule.py
CreatedBy	`myfunc()`	mymodule.py	outdir/	outdir/	`myfunc()`	mymodule.py
Uses	`parse()`	mymodule.py	`myfunc()`	mymodule.py	None	mymodule.py

There are four files: 'file1', 'file2', 'mymodule.py', and 'outdir/'. The four links represent that: (1-2) the `myfunc()` function depends on the files 'file1' and 'file2', and is inside of the Python module 'mymodule.py', (3) the 'outdir/' directory is created by the function `myfunc()` inside of 'mymodule.py', and (4) the `myfunc()` function uses the function `parse()`, and both are in 'mymodule.py'.

The `LinkList` is stored on-disk in a serialized format. The `LinkList` can trivially be represented in a tabular format (CSV) or XML or RDF consistent with the Open Provenance Model (OPM) [6]. We could easily allow users to edit pipelines by directly editing the `LinkList`, but do not currently do so.

3.3 Exploring, Updating and Sharing

StarFlow includes a set of Python command-line tools for exploring dependencies, propagating changes, and extracting and sharing analysis pipelines.

Exploring dependencies. `DownstreamLinks()` takes a list of source dependencies and propagates down through the dependency network to return a list of downstream dependencies. Its default behavior uses file time stamps to propagate only through dependencies in need of updating, i.e., dependencies whose targets' time stamps are older than those of their sources. When `Forced = True`, it ignores time stamps and instead propagates through all downstream dependencies. `UpstreamLinks()` is an analogous function for upstream dependencies.

`ShowUpdates()` uses `DownstreamLinks()` to determine and print a readable report describing what Python functions to execute, and in what order, to update dependency targets relative to their sources, without actually calling them.

Propagating changes. StarFlow's automatic updating engine supports two styles of change propagation. `Update()` uses `ShowUpdates()` to implement **downstream updating**, so changes to the dependency network trigger execution of downstream functions. `Make(Targets)` implements **upstream updating** in the spirit of `make`, so targets are made by executing upstream functions that have changed or whose upstream dependencies have changed. For both functions, the user can force re-execution by passing `Forced = True`. Both can propagate changes through a restricted dependency network, i.e., a filtered `LinkList`. The user can pass a list of regular expression filters mapping to a list of Python functions and specify default filters from a configuration file.

StarFlow's automatic updating engine combines change propagation with a set of optional "smart" features: (1) consistency checking that can issue an error or warning if user annotations contradict runtime file I/O, (2) Unix-style `diff` checking between each round of updates, so that if a set of updates produces no changes, unnecessary downstream updates are cancelled, (3) data archiving and managed exception handling so that if user scripts throw errors, downstream updates are cancelled and previous versions of data restored, and (4) storing of sha-1 checksums after each round of computation to detect corrupt data.

Extracting and sharing. `Extract(Targets)` uses `UpstreamLinks(Targets)` to find all code modules and data sources required to recompute `Targets` and then extracts them into a zipped archive. The result of `Extract(Targets)` can then be integrated into another user's StarFlow environment with `Integrate()`.

3.4 Basic Use Case

StarFlow enables a highly organized real-time data analysis development cycle where the user can automatically update her pipelines every time she edits scripts or data. Consider a user-generated Python module containing several parameterized functions for basic data processing and analysis:

```
def Parser(infile, outfile):
  X = open(infile)
  Y = remove_header(X)
  Z = pivot(Y)
  save(Z, outfile)

def Cluster(infile, outfile, distfunc, param=None):
  X = open(infile)
  C = hcluster(X, distfunc, param)
  save(C, outfile)

def PCA(infile, outfile):
  X = open(infile)
  Y = pca(X)
  save(Y, outfile)
```

```
  def Compare(PCAfile, Clusterfile, outfile):
  X1 = open(PCAfile)
  X2 = open(Clusterfile)
  Y = compute_error(X1, X2)
  save(Y, outfile)
```

These functions read input data files, process their contents, and write output data files. They depend on other functions located either in the same module or imported from elsewhere. Regular user interaction at the Python interpreter, without StarFlow, might look like this:

```
>> from my_module import *
>> Parser('raw_data.csv', 'data.csv')
>> PCA('data.csv', 'pca.csv')
>> Cluster('data.csv', 'euc.csv', EuclideanDistance)
>> Compare('pca.csv', 'euc.csv', 'error1.csv')
>> Cluster('data.csv', 'geo.csv', GeometricDistance)
>> Compare('pca.csv', 'geo.csv', 'error2.csv')
```

StarFlow enables the user to track the dependencies of these sorts of operations. Suppose the user wants to use the `depends_on` and `creates` annotations to record the first four function calls from the above interpreter session. She could add the following lines to `my_module.py`:

```
def ParseBigInput(depends_on='raw_data.csv', creates='data.csv'):
  Parser(depends_on, creates)

def DoPCA(depends_on='data.csv', creates='pca.csv'):
  PCA(depend_on, creates)

def ClusterEuclid(depends_on='data.csv', creates='euc.csv'):
  Cluster(depends_on, creates, EuclideanDistance)

def Comp(depends_on=('pca.csv', 'euc.csv'), creates='error.csv'):
  Compare(depends_on[0], depends_on[1], creates)
```

With the information flow annotated, StarFlow can determine the complete dependency network prior to runtime (Fig. 2). The user opens the Python terminal and initializes StarFlow by importing its modules.

```
>> from starflow.interactive import *
```

Before executing anything, the user can type `ShowUpdates()` to see what functions will run and in what order:

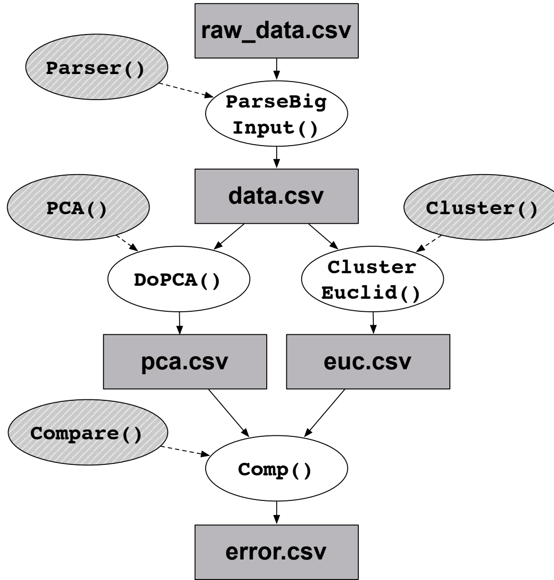

Fig. 2. Dependency graph extracted by StarFlow. Arrows are in the direction of information flow. Files are rectangles and functions are ovals. For example, the function `DoPCA()` depends on the file `data.csv`, creates the file `pca.csv`, and uses the function `PCA()`.

```
>> ShowUpdates()
Round 1:  my_module.ParseBigInput
Round 2:  my_module.DoPCA, my_module.ClusterData
Round 3:  my_module.Comp
```

The output of `ShowUpdates()` corresponds to the breadth-first parallelization scheme that StarFlow can implement automatically. As before, the user can make edits to data or functions and propagate incremental changes by calling `Update()` or related tools. When using StarFlow with the `depends_on` and `creates` annotations, the user may find that she only needs to type two commands at the prompt – `ShowUpdates()` and `Update()` – to review and propagate changes as she develops her analysis pipelines.

Later, the user edits her scripts and data, and wants to propagate these changes. First, she makes a small change to the file `raw_data.csv`. She types `Update()`, and StarFlow re-executes each of the function calls she typed at the prompt because they are all downstream of the `raw_data.csv` file. Next, she makes a small change to the `hcluster` function. Now when she types `Update()`, StarFlow re-executes only the function calls to `Cluster`, because it depends directly on `hcluster`, and `Compare`, because it is downstream of `Cluster`.

4 Workflow Abstraction

StarFlow supports a simple metaprogramming syntax that allows the user to construct abstract workflows and then instantiate concrete analysis pipelines from them. The user represents a workflow by a simple data model for a list of concrete workflow steps, the `OpList`. Each step corresponds to a concrete function call with inputs and outputs and is represented as a three-tuple: a unique string name, a function, and a tuple of function parameters. Actual concrete workflows are instantiated by passing the `OpList` to the `Actualize()` templating engine. `Actualize(OpList, 'path.py')` writes out a Python module, 'path.py' where each step corresponds to a hard-coded function with `depends_on` and `creates` annotations. For example, this script:

```
def instantiator(creates='instances.py'):
  L = []
  for i in ['a', 'b', 'c']:
      L += [('step_'+i, myfunc, ('in1_'+i, 'in2_'+i, 'out_'+i))]
  Actualize(L, 'instances.py')
```

instantiates three concrete instances of a one-step workflow, where `myfunc` is `@activate` decorated, as in §3.2. Each workflow step is automatically written out as a separate function in 'instances.py':

```
def step_a(depends_on=('in1_a', 'in2_a'), creates='out_a')
  myfunc('in1_a', 'in2_a', 'out_a')

def step_b(depends_on=('in1_b', 'in2_b'), creates='out_b')
  myfunc('in1_b', 'in2_b', 'out_b')

def step_c(depends_on=('in1_c', 'in2_c'), creates='out_c')
  myfunc('in1_c', 'in2_c', 'out_c')
```

By combining workflow abstraction with automatic updating, we have developed a parallelization engine. Users can exploit this engine by writing an abstract workflow that generates many concrete instances. When configured for parallelization, StarFlow's `Update()` command materializes these instances, computes their dependency network and partitions them into parallelizable groups. We then use a grid scheduler to dispatch the parallel jobs on available machines. The next section shows how we have integrated StarFlow with Amazon's Elastic Compute Cloud (EC2) [17] to perform automatically, parallelized web download and analysis.

Applying parallelization to abstract workflows. We combine StarFlow with StarCluster [25] to enable automatic parallelization of workflows in a high performance cloud setting. StarCluster manages the creation and adminstration of clusters hosted on Amazon's EC2, connecting to SunGrid Engine for job scheduling and load balancing.

Below we illustrate a representative and simple scenario; it is embarrassingly parallel and contains just one of many possible analyses of interest. Suppose the user wants to download data from the U.S. Environmental Protection Agency about facilities or sites subject to environmental regulation [2]. There is one downloadable file for each of 50 states, and the user wants to call the function `pairwise_comparison()` for each pair of states. She writes this module, using `Actualize` to automatically produce 'EPA_instances.py':

```
01 urlroot = 'http://www.epa.gov/enviro/html/frs_demo/'
02 urlroot += 'geospatial_data/state_files/state_combined_'
03
04 def EPA(depends_on='states.txt', creates='EPA_instances.py'):
05
06     L = []
07     statelist = open('states.txt','r').read().strip().split(',')
08
09     for S in statelist:
10         L += [('get_'+S, wget, (urlroot+S+'.zip', S+'.zip'))]
11         L += [('unzip_'+S, unzip, (S+'.zip', S+'/'))]
12
13     for i in range(0, 49):
14         S1 = statelist[i]
15         for j in range(i+1, 50):
16             S2 = statelist[j]
17             L += [('compare_'+S1+'_'+S2, pairwise_comparison,
18             (S1+'/data.csv', S2+'/data.csv', S1+'_'+S2+'.csv'))]
19
20     Actualize(L, 'EPA_instances.py')
```

In the first `for` loop (lines 9-11), the user downloads and unzips the data, producing two rounds of 50 function executions that, within a round, can be run in parallel. For each state, a large CSV file (≈100 MB) is unarchived. Next, she analyzes all pairs of states, generating a third round of 1225 parallelizable function executions.

The user starts a 10-node cluster on EC2 with StarCluster, opens a Python terminal and initializes StarFlow. When she runs `Update()`, StarFlow executes `analyze.EPA()`, which writes out 'EPA_instances.py'. StarFlow automatically detects the functions inside of this new module, determines their dependencies and how to run them in parallel on 10 nodes, and then does so.

5 Future Directions

Although Starflow is currently Python-specific, we'd like to take the underlying principles and design and apply them to other scripting languages, such as

Perl and R, to determine how generally applicable the ideas are. We also would like to to extend StarFlow to the interactive shell in two ways: (i) given a variable, automatically update its value in response to upstream changes, and (ii) given a sequence of commands, automatically generate a script from the minimal sequence needed to produce a set of targets. We have developed and plan to improve a GUI for StarFlow that integrates browsing of files, dependencies, data and metadata. We are working on more comprehensive parallelization and workflow tools. We will integrate StarFlow's dependency tracking infrastructure with a version control system such as Mercurial [19].

6 Conclusion

StarFlow provides a powerful, script-centric environment for data analysis. It strategically combines dynamic runtime analysis, static analysis of code, and user annotations to provide fine-grain propagation. StarFlow enables workflow abstraction and automatic parallelization, and we have implemented StarFlow in the cloud.

Acknowledgments. We thank P. C. Sabeti (Harvard), who supported our initial effort, and whose students and collaborators provided us with a valuable case study. In particular, I. Shlyakhter, a member of the Sabeti Lab, provided us with valuable insight on workflow abstraction. Finally, we thank the reviewers, members of the PASS group (Harvard), and P. J. Guo (Stanford) for many helpful comments on this manuscript.

References

1. Proceedings of the 2010 USENIX Workshop on the Theory and Practice of Provenance, San Jose, CA, USA. USENIX (February 22, 2010)
2. United States Environmental Protection Agency. Epa frs facilities state combined csv files download, `http://epa.gov/enviro/html/frs_demo/geospatial_data/geo_data_state_combined.html`
3. Berthold, M.R., Cebron, N., Dill, F., Gabriel, T.R., Kotter, T., Meinl, T., Ohl, P., Thiel, K., Wiswedel, B.: Knime - the konstanz information miner: version 2.0 and beyond. SIGKDD Explor. Newsl. 11(1), 26–31 (2009)
4. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T.: Vistrails: visualization meets data management. In: SIGMOD 2006 Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 745–747. ACM, New York (2006), General Chair-Yu, Clement and General Chair-Scheuermann, Peter and Program Chair-Chaudhuri, Surajit
5. clario Analytics. clario, `http://clarioanalytics.com`
6. Clifford, B., Freire, J., Gil, Y., Groth, P., Futrelle, J., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Simmhan, Y., Stephan, E., den Bussche, J.V.: The open provenance model core specification, v1.1 (2009), `http://eprints.ecs.soton.ac.uk/18332/1/opm.pdf`

7. LinkedIn Corporation Azkaban, http://sna-projects.com/azkaban/
8. Pentaho Corporation, Kettle: Pentaho data integration, http://kettle.pentaho.org
9. Deelman, E., Blythe, J., Gil, A., Kesselman, C., Mehta, G., Patil, S., Su, M.-h., Vahi, K., Livny, M.: Pegasus: Mapping scientific workflows onto the grid, pp. 11–20 (2004)
10. Elkabany, K., Staley, A., Park, K.: Picloud - cloud computing for science. simplified. In: SciPy 2010 Python for Scientific Computing Conference, Austin, TX (July 2010)
11. Foster, I., Vckler, J., Wilde, M., Zhao, Y.: Chimera: A virtual data system for representing, querying, and automating data derivation. In: Proceedings of the 14th Conference on Scientific and Statistical Database Management, pp. 37–46 (2002)
12. The Eclipse Foundation. Eclipse c/c++ development tooling project, http://www.eclipse.org/cdt
13. Guo, P.J., Engler, D.: Towards practical incremental recomputation for scientists: An implementation for the python language. In: TaPP 2010 [1] (2010)
14. Ikeda, R., Widom, J.: Panda: A system for provenance and data. In: TaPP 2010 [1] (2010)
15. Yahoo! Inc., Oozie, http://yahoo.github.com/oozie/
16. Kuehn, H., Liberzon, A., Reich, M., Mesirov, J.P.: Using genepattern for gene expression analysis. Curr. Prot. in Bioinformatics, 7.12.1–7.12.39 (2008)
17. Amazon Web Services LLC. Amazon elastic compute cloud (ec2), http://aws.amazon.com/ec2
18. McPhillips, T., Bowers, S., Zinn, D., Ludaschera, B.: Scientific workflow design for mere mortals. Future Generation Computer Systems 25(5), 541–551 (2009)
19. Mercurial. Mercurial, http://mercurial.selenic.com
20. Missier, P., Belhajjame, K., Zhao, J., Roos, M., Goble, C.: Data lineage model for taverna workflows with lightweight annotation requirements. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 17–30. Springer, Heidelberg (2008)
21. Muniswamy-Reddy, K.-K., Holland, D.A., Braun, U., Seltzer, M.I.: Provenance-aware storage systems. In: USENIX Annual Technical Conference, General Track, pp. 43–56. USENIX (2006)
22. Oinn, T.M., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, R.M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics 20(17), 3045–3054 (2004)
23. Pan, M.J.: pomsets: workflow management for your cloud. In: SciPy 2010 Python for Scientific Computing Conference, , Austin, TX (July 2010)
24. The GNU Project, Gnu automake, http://www.gnu.org/software/automake
25. Riley, J.: Starcluster - numpy/scipy computing in the cloud. In: SciPy 2010: Python for Scientific Computing Conference, Austin, TX (July 2010)
26. Taylor, J., Schenck, I., Blankenberg, D., Nekrutenko, A.: Using galaxy to perform large-scale interactive data analyses. Curr. Prot. in Bioinformatics, 10.5.1–10.5.25 (2007)