noWorkflow: Capturing and Analyzing Provenance of Scripts

Leonardo Murta[1], Vanessa Braganholo[1(✉)], Fernando Chirigati[2], David Koop[2], and Juliana Freire[2]

[1] Universidade Federal Fluminense, Niterói, Brazil
{leomurta,vanessa}@ic.uff.br
[2] New York University, New York, USA
{fchirigati,dakoop,juliana.freire}@nyu.edu

Abstract. We propose noWorkflow, a tool that transparently captures provenance of scripts and enables reproducibility. Unlike existing approaches, noWorkflow is non-intrusive and does not require users to change the way they work – users need not wrap their experiments in scientific workflow systems, install version control systems, or instrument their scripts. The tool leverages Software Engineering techniques, such as abstract syntax tree analysis, reflection, and profiling, to collect different types of provenance, including detailed information about the underlying libraries. We describe how noWorkflow captures multiple kinds of provenance and the different classes of analyses it supports: graph-based visualization; differencing over provenance trails; and inference queries.

1 Introduction

While scripts are widely used for data analysis and exploration in the scientific community, there has been little effort to provide *systematic* and *transparent* provenance management support for them. Scientists often fall back on Workflow Management Systems (WfMSs), which provide infrastructure to automatically capture the input, intermediate, and output data involved in computations, allowing experiments to be managed, assessed, and reproduced [12,16,18]. Although WfMSs play an important role in bridging the gap between experimentation and provenance management, they have limitations that have hampered a broader adoption, notably: moving to a new environment can be difficult and requires a steep learning curve, and wrapping external scripts and libraries for use in a WfMS is time-consuming. In addition, data analysis tasks that use multiple tools require each to be integrated with the WfMS. When this is not possible (or desirable), scientists often run scripts to orchestrate analyses and connect results obtained from multiple tools.

Collecting provenance of scripts when not using a WfMS is challenging. First, unlike most pipelines supported by dataflow-based systems, *scripts can encode a control flow and include cycles*, which makes it more difficult to identify which functions contributed to the generation of a given data product. Second, *determining the correct level of granularity to capture is hard*: very fine-grained provenance may overwhelm scientists with a large volume of data to analyze, while a

© Springer International Publishing Switzerland 2015
B. Ludäscher and B. Plale (Eds.): IPAW 2014, LNCS 8628, pp. 71–83, 2015.
DOI: 10.1007/978-3-319-16462-5_6

coarser granularity may omit important information. In contrast, workflows in a WfMS have well-defined boundaries for such capture, which are determined by how the underlying computational modules are wrapped. Finally, since *scripts run outside of a controlled environment* such as a WfMS, one cannot make many assumptions (e.g., the presence of a version control system) beyond the involvement of source code and an interpreter/compiler, which makes it difficult to track library dependencies and changes in files.

Some of the existing approaches that do not require a WfMS rely on scientists to modify the experiment scripts to include annotations or calls to provenance capture functions [1,3,7]. Such approaches are intrusive, time-consuming, and error-prone. Others require scientists to use a version control system to track changes to the source code, or are not entirely automatic, requiring input from scientists [3,10]. There are also approaches that capture provenance at the operating system level [6,8,17], which monitor system calls and track processes and data dependencies between these processes. These systems, however, do not have visibility into what happens inside the scripts underlying the processes.

In this paper, we propose a new approach to capture provenance of scripts that addresses the aforementioned challenges. We review the existing types of provenance representation and argue that, in the absence of a controlled environment, a new kind of provenance – *deployment provenance* – is necessary to capture detailed data about the underlying libraries. We then present noWorkflow (**n**ot **o**nly **Workflow**), a tool that implements the proposed approach, and describe how it *transparently* captures provenance of scripts, including control flow information and library dependencies. noWorkflow is *non-intrusive* and relies on techniques from Software Engineering, including abstract syntax tree analysis, reflection, and profiling, to collect different types of provenance without requiring a version control system or an instrumented environment. The tool supports three different types of analyses, including visualization and query mechanisms, to help scientists explore the captured provenance and debug the execution, as well as to enable reproducibility. Although noWorkflow was developed for Python, a language with significant adoption by the scientific community, the ideas presented here are language-independent and can be applied to other scripting languages.

2 Provenance of Scripts

WfMSs provide a controlled environment in which workflows are executed—the workflow engine orchestrates the invocation of the computational modules of a workflow. Since provenance is captured for these invocations, the provenance granularity is determined by how computations are modeled inside the workflow system, i.e., how libraries are wrapped. Scripts, in contrast, lack this well-defined structure and the isolation provided by the workflow engine. Thus, to capture the provenance of scripts, we have to address two important challenges: how to represent information about the environment and how to determine the level of provenance granularity.

2.1 Provenance Representation

There are two types of provenance for scientific workflows: prospective and retrospective [5]. Prospective provenance describes the structure of the experiment and corresponds to the workflow definition, the graph of the activities, and their associated parameters. Retrospective provenance captures the steps taken during the workflow execution, and while it has similar (graph) structure, it is constructed using information collected at runtime, including activities invoked and parameter values used, intermediate data produced, the execution start and end times, etc. The wrapping required by a WfMS to orchestrate the execution of modules from a tool or library naturally creates a level of abstraction for the execution: the module is a black box and its details are hidden. Because the wrapped libraries are integrated with the WfMS, it is possible for the system to track and control them, e.g., to detect that a wrapped library has changed and to upgrade the workflows accordingly [13].

For scripts, this abstraction is absent. Therefore, it is important to capture detailed information about the computational environment (e.g., library dependencies and environment variables) where the script runs. Consider, for example, the Python script in Fig. 1, which runs a simulation to predict weather using historical data about temperature and precipitation. For simplicity of exposition, the real (and expensive) simulation performed by *simulate* is defined in a separate module (*simulator*) not shown in the example. This script depends on 703 distinct modules, although only four are explicitly declared (lines 1–4). Suppose we run the experiment script once and obtain a result. If later, software is installed (or upgraded) that silently updates one of the modules on which the experiment script depends, the next execution may produce a different result, even though its source code remains unchanged. If these dependencies are not systematically captured, it may be difficult to understand why results are different between executions that are apparently identical.

The provenance needed here is neither prospective nor retrospective, and it needs to be captured right before execution. Borrowing terms from software engineering, where software goes through three phases, i.e., *definition*, *deployment*, and *execution* [9], we define three types of provenance needed for scripts:

- *Definition Provenance* captures the structure of the script, including function definitions, their arguments, and function calls; it corresponds to prospective provenance.
- *Deployment Provenance* captures the execution environment, including information about the operating system, environment variables, and libraries on which the script depends. As discussed before, this may change from one execution to another, even if the source code remains the same. In addition, it extends beyond dependencies a programmer explicitly defines, and the concrete library versions that are loaded depend on the deployment environment.
- *Execution Provenance* captures the execution log for the script (e.g., function activations, argument values, and return values); it corresponds to retrospective provenance.

```
01. import csv
02. import sys
03. import matplotlib.pyplot as plt
04. from simulator import simulate
05.
06. def run_simulation(data_a, data_b):
07.     a = csv_read(data_a)
08.     b = csv_read(data_b)
09.     data = simulate(a, b)
10.     return data
11.
12. def csv_read(f):
13.     reader = csv.reader(open(f, 'rU'), delimiter=':')
14.     data = []
15.     for row in reader:
16.         data.append(row)
17.     return data
18.
19. def extract_column(data, column):
20.     col_data = []
21.     for row in data:
22.         col_data.append(float(row[column]))
23.     return col_data
24.
25. def plot(data):
26.     # getting temperature
27.     t = extract_column(data, 0)
28.     # getting precipitation
29.     p = extract_column(data, 1)
30.     plt.scatter(t, p, marker='o')
31.     plt.xlabel('Temperature')
32.     plt.ylabel('Precipitation')
33.     plt.savefig('output.png')
34.
35. # main program
36. data_a = sys.argv[1]
37. data_b = sys.argv[2]
38. data = run_simulation(data_a, data_b)
39. plot(data)
```

Fig. 1. Example of a Python script (*simulation.py*) that predicts temperature and precipitation in the near future.

2.2 Provenance Granularity

As discussed above, in WfMSs, provenance is captured at the level of an activity, and what happens inside an activity is not taken into account by the provenance infrastructure. In contrast, such boundaries are not well-defined in the context of scripts. Thus, an important question is how to determine the level of granularity at which to capture provenance for scripts. One alternative would be to use approaches that capture provenance at the operating system level [6,17]. Since these systems intercept system calls (e.g., file reads and writes, execution of binaries), they produce a high volume of very fine-grained information that represent data dependencies between processes. It can be difficult to explore this information and connect it to the underlying experiment specification. Consequently, identifying which experiment activity influenced the generation of a given data product can be challenging. On the other hand, if we consider the entire script as a black-box, and capture provenance at a coarse granularity, it

would be impossible to know which functions contributed to the generation of a given data product.

We posit that functions in a script are a suitable choice for provenance capture—this is most likely to be meaningful to users since it is closer to the experiment specification. We note, however, that even this level may be overwhelming. For instance, profiling the (very small and simple) script of Fig. 1, we observed 156,086 function activations. This includes functions called by functions that are used in the main experiment script, such as `plt.scatter` (line 30). Clearly, analyzing this volume of information is hard and time-consuming; an alternative is to capture only the activations related to functions that are defined by the programmer (i.e., that have user-defined functions as source or target). In the example, this entails all activations related to the main program along with functions `run_simulation`, `csv_read`, `extract_column`, and `plot`. This approach significantly reduces the amount of captured information, and makes it easier for users to keep track of what is happening throughout the execution.

3 noWorkflow

As a proof of concept, we built noWorkflow, a command line tool written in Python that transparently captures provenance of Python scripts. Running noWorkflow is as simple as running a Python script: `now run <script>`. In noWorkflow, the execution of a given experiment script is called a *trial*. Each trial is assigned a sequential identification number that is automatically generated. Provenance for each trial is captured and stored for future analysis. The system distinguishes a *function call* from a *function activation*: the former is related to definition provenance and can be captured by statically analyzing the source code while the latter is related to execution provenance. For example, in Fig. 1, `data.append` is a single function call (line 16), but it may have many activations at runtime, with different arguments and return values, because it is inside a `for` loop. In what follows, we describe how noWorkflow, in the absence of a controlled execution environment, captures and stores the different types of provenance (see Fig. 2). We also discuss useful analyses that can be performed over script provenance.

3.1 Provenance Capture

Definition Provenance. To capture definition provenance, noWorkflow uses the *abstract syntax tree* (AST) of the script to identify all user function definitions, function calls, arguments, and global variables referenced in the script. We chose user-defined functions as the granularity for provenance capture (Sect. 2.2), and the AST is used to capture the source code of each function definition. In the example (Fig. 1), the source code of `run_simulation` (lines 6–10) is entirely stored, which allows the tool to monitor the evolution of each function definition independently. In addition, noWorkflow stores the source code of the

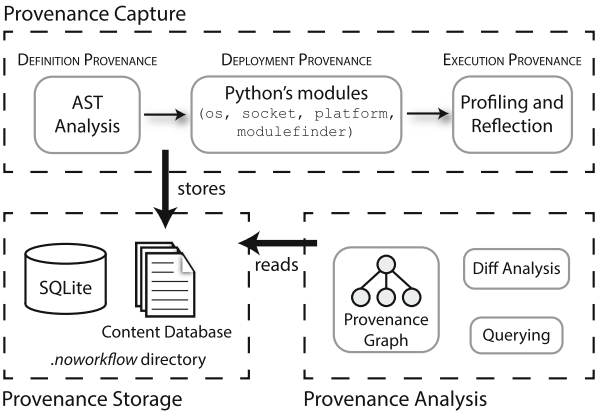

Fig. 2. Architecture of noWorkflow.

entire script. All this information is associated with an experiment trial, thus making it possible to know which function definitions belong to a specific trial.

Each function is then analyzed to capture the objects referenced inside it, including arguments, function calls, and global variables. These objects are associated with the corresponding function definition. Consider for example the function `run_simulation` in Fig. 1: noWorkflow captures two arguments (`data_a` and `data_b`, on line 6), and two function calls (`csv_read` on lines 7 and 8, and `simulate` on line 9). Despite the fact that `csv_read` is called twice in `run_simulation`, we register this information only once as a dependency from `run_simulation` to `csv_read`. At runtime, noWorkflow is able to distinguish between different function activations of the same function call as well as different activations of different calls from the same function definition.

Deployment Provenance. noWorkflow captures two different types of deployment provenance: environment and module (i.e., library) dependencies. This provenance is captured right before the execution of the experiment script begins, and is associated with an experiment trial. noWorkflow uses libraries provided by Python to capture environment information, including `os` to capture operating system information, `socket` to capture the host name, and `platform` to capture information about the machine architecture and Python environment. noWorkflow also uses Python's `modulefinder` library to find the transitive closure of all module dependencies. For each module that this library finds, our tool stores the library name, version, file name (including its full path), and source code (if available).

It is possible that environment and module dependencies change during the script execution. In this case, to precisely capture this information, deployment provenance would need to be gathered dynamically, right before each function activation. However, since this situation is very rare (and advised against), and to avoid introducing a large overhead, we have opted for capturing deployment provenance right before executing the script.

Execution Provenance. Execution provenance includes function activations, argument values, return values, global values, start and finish times for each function activation, as well as their context, and the content of all files manipulated by the experiment script during execution. noWorkflow captures this information through *profiling* and *reflection*.

noWorkflow implements specific methods of the Python profiling API and registers itself as a listener. During an execution, the profiler notifies the tool of all function activations in the source code. Notice that this goes very deep into the execution flow—recall that our simple simulation script has 156,086 function activations. As discussed before, to avoid overloading users with large volumes of information, thus overcoming the granularity challenge, noWorkflow only registers function activations related to user-defined functions. For the script in Fig. 1, noWorkflow registers that `csv_read` calls `data.append` (line 16), but it does not register functions that `data.append` calls. At this moment, we also capture the start time of the function activation, together with the values of every argument, return, and globals that may be involved in the function activation.

While monitoring only user-defined functions reduces the volume of information to be captured, it may miss an important aspect of the experiment: file access. Explicit *open* system calls in the script will be captured, but if *open* is called from a function not defined by the scientist (e.g., `plt.savefig` on line 33 of Fig. 1), this information would be missed by noWorkflow. noWorkflow addresses this issue by using reflection to alter the behavior of a system call. We implement a new function that overwrites the system's *open* function and alters its behavior so that every time *open* is called, we capture the content of the file, store it, call the original *open* system call, and then capture and store the file's content *again*. Thus, noWorkflow preserves the content before and after a file is accessed, allowing us to detect, for instance, if a file has been modified.

Notice that reflection is not enough to identify which function called *open*. To make this association, noWorkflow uses an *activation stack*: every time there is an activation of a user-defined function, it is pushed onto the stack, and when the activation finishes, it is popped from the stack. When *open* is called, the function on top of the stack is tagged as being responsible for opening the file. Figure 3 shows an example: when `plt.savefig` is called from the user-defined function `plot` (line 33), its activation is pushed to the stack; when *open* is called to save *output.png*, `plt.savefig` will be on top of the stack, thus allowing noWorkflow to link it to the modified file. Right before popping an activation from the stack, its end time and return value are registered. If a function is activated several times, noWorkflow registers all activations and links them with the activation on top of the stack that triggered them. This allows noWorkflow to keep track of function activation dependencies, together with the source code line that corresponds to this call and all information previously discussed in this section.

3.2 Provenance Storage

Because transparency is one of our goals, noWorkflow includes an embedded storage mechanism that does not require any installation or configuration. All provenance is automatically stored to disk in a directory named *.noworkflow* in the script directory. This directory holds both a relational database for structured data and a database for file contents. These databases are linked together by means of SHA1 hash codes.

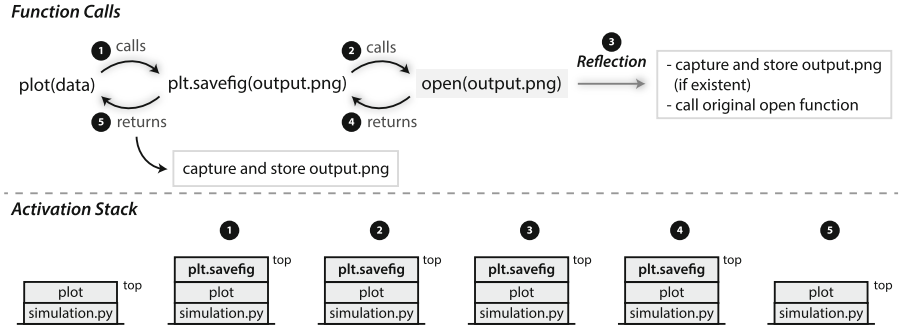

Fig. 3. Example of how reflection and activation stack work on noWorkflow. When *open* is called (2), the file is captured before executing the original system call function (3), and since `plt.savefig` is on top of the stack, noWorkflow knows that this function is the one responsible for opening the file.

noWorkflow uses SQLite to store structured data which includes definition provenance (e.g., function definitions and objects they reference, including function calls), deployment provenance (e.g., environment variables and module dependencies), and execution provenance (e.g., runtime information about trials, file accesses, function activations, and object values). Hash codes are also stored whenever possible, e.g., SHA1 hashes of the source code of a function and of files before and after access. In contrast, file contents are stored directly to disk in what we call the *content database*. To avoid OS limitations regarding the number of files that can be stored in a directory, we use the same strategy Git uses to store files: file content is stored in a directory that corresponds to the first two characters of its SHA1 hash in a file named by the remaining characters of the SHA1 hash. noWorkflow maintains all files involved with the experiment, and all SHA1 hashes stored in the relational database have a counterpart file stored in the content database. Data in the SQLite database is always associated with a given execution of the experiment script (i.e., a trial). This allows noWorkflow to save disk space: whenever the hash code of a given file is the same, the hash is stored in the database, but not the file itself again. In addition, the provenance storage in noWorkflow eases *reproducibility*: scientists can simply share the *.noworkflow* directory with their collaborators to exchange provenance data.

3.3 Provenance Analysis

While captured provenance aids reproducibility, another important goal is facilitating the *analysis* of provenance to locate, understand, and compare techniques. The current version of noWorkflow supports three different analysis techniques: *graph-based*, *diff-based*, and *query-based*.

Graph-Based Analysis. Graph-based analysis is facilitated by *visualizing* the provenance of a trial in a graph which provides an overview of the script execution and supports comprehension of both functional and non-functional attributes. However, the provenance of even simple scripts may consist of a large number of function activations, particularly in the presence of loop structures, which may lead to visualization overload problems. For this reason, noWorkflow first *summarizes* the provenance before producing its activation graph. Our overall approach is based on a three-step strategy: summarization, construction, and drawing.

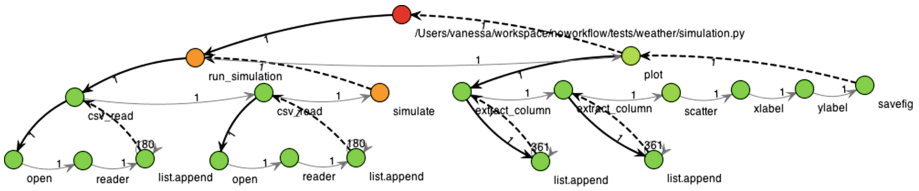

Fig. 4. Graph-based visualization generated from the example in Fig. 1.

The summarization step aggregates different activations of a function call if they belong to the same context (i.e., same loop). The idea is to aggregate the provenance by activation stack, function call line number, and function name. Therefore, each function call may have multiple activations together with their activation arguments, return values, and timestamps. The second step consists of building a graph from the vertices generated by the summarization step and edges extracted from the function activation sequence. There are three types of edges: *call*, when a function calls another function; *sequence*, when two functions are called in sequence within the same activation stack; and *return*, when a function finishes its execution and the control flow returns to the function in the top of the stack. Finally, the third step is rendering the graph. Each vertex is labeled with the function name and is colored according to the traffic light scale (shades from green to yellow to red) [4]: function calls with faster activations are colored in shades of green, while the ones with slower activations are colored in shades of red. Each edge displays the number of times the control flow passed through it, and each edge type has a different shape to ease the visual distinction: *call* edges are thicker and darker, *sequence* edges are thinner and lighter, and *return* edges are dashed, thicker, and darker. There is also a tooltip window that provides detailed information about each node (activation).

Figure 4 shows the graph-based visualization generated from the example of Fig. 1. From the graph, we can observe that the script called both `run_simulation` and `plot` in sequence. It is also possible to see that `run_simulation` is much slower than `plot`, and that there are four loop structures in the script, summarized by noWorkflow: two loops occurring inside `csv_read` and two loops occurring inside `extract_column`.

Diff-Based Analysis. In some provenance analysis scenarios, it is crucial to contrast two trials to understand why results differ. There are many aspects that influence the generation of an output, including script modifications, environment variable changes, and module updates. noWorkflow provides a mechanism to contrast two trials and identify changes that may influence the results. This mechanism allows comparison of the basic attributes of trials (e.g., date, script, and arguments), environment variables, and module dependencies, showing which attributes have changed, and which variables and modules have been added, removed, or replaced. This is especially useful for reproducibility, since it becomes easy to compare two executions of the same experiment in different environments. Additionally, our diff-based strategy can be easily extended to support object-specific diffs.

Query-Based Analysis. Since provenance data is stored in a relational database, SQL would be a natural choice for the query language. However, SQL is known to be very inefficient for recursive queries, and queries that employ transitive closures would be hard to write and take a long time to process. To overcome this limitation, we provide an inference-based query mechanism based on Prolog. noWorkflow is able to export Prolog facts and rules of a given trial which can then be used to query the collected provenance data. The facts follow the same structure of the relational tables that we use to store provenance data. To make queries easier, noWorkflow also provides a set of Prolog inference rules. As an example, the rule `access_influence` can be used to find out which files may have influenced the generation of a given file. Running the query `access_influence(File, 'output.png')` returns a list of files that may influenced the generation of *output.png*, which, in the case of our example, are *data1.dat* and *data2.dat*. Note that, since we export the Prolog facts, any Prolog system can be used. New rules can also be added by users.

4 Related Work

Different mechanisms for provenance capture have been proposed, and some can be applied to scripts. Tools that capture provenance at the operating system level [6,8,17] monitor system calls and track processes and data dependencies between these processes. Because the dependencies are recorded at the process level, it can be difficult to reconcile the provenance with the script definition as these systems cannot see what happens inside the processes. The provenance captured by noWorkflow is of a different nature—it represents dependencies within processes at the function level. In this sense, our approach is closer to the

work by Cheney et al. [2]. They proposed a formalism that uses techniques based on program analysis slicing to represent the provenance of database queries so that it is possible to show how (part of) the output of a query depends on (parts of) its input. In contrast, we focus on provenance of (general) scripts, not just database queries. Another important distinction is that noWorkflow captures additional dependencies: it captures deployment provenance and, in addition to function and variable dependencies, it also captures general data dependencies from file reads and writes.

Several tools capture provenance from scripts and connect it to the experiment data. Bochner et al. [1] proposed an API and a client library to capture provenance for Python scripts. Gavish and Donoho [7] introduce the notion of a Verifiable Computational Result (VCR), where every result is assigned a unique identifier and results produced under the exact same conditions have the same identifier to support reproducibility. Unlike noWorkflow, these tools are intrusive and require users to change their scripts and include specific API method calls. Sumatra [3] collects provenance information from Python scripts. It is able to capture input and output data produced by each run (as long as they are explicitly specified by the user), parameters, module dependencies, and platform information. It is also able to detect when a module the script depends on has changed. The source code, however, needs to live in a version control system so that changes from one version to another can be detected. ProvenanceCurious [10] is another tool that can infer data provenance from Python scripts. It also uses AST analysis to capture every node of the syntax tree, and it uses a graph to provide query capabilities. However, for every operation, it requires input from the users regarding whether or not the operation reads or writes persistent data—this information is transparently captured by noWorkflow.

The approach taken by Tariq et al. [19] makes use of the LLVM compiler framework to automatically insert provenance capture at each function entry and exit. Thus, similar to noWorkflow, their approach is transparent—users do not need to manually annotate their code. However, there are important differences between the two approaches. Since Tariq et al. rely on a compiler, they are restricted to capturing static information. noWorkflow, on the other hand, captures both static and dynamic information. The latter is crucial for interpreted languages such as Python, since the underlying program (and objects) can change during runtime. In addition, noWorkflow captures dependencies that involve global variables within a function; these are ignored by Tariq et al., since they do not capture what happens inside functions. While our current implementation selects user-defined functions to track, we would like to explore mechanisms such as the one used by Tariq et al. to allow users to have more control over the captured provenance.

5 Conclusions and Future Work

We have presented noWorkflow, an approach to capture provenance of experiment scripts. Compared to previous approaches, the main benefits of noWorkflow

are: (i) it is completely transparent—users do not need to instrument their code; (ii) it systematically captures three types of provenance—definition, deployment, and execution provenance—using non-intrusive mechanisms; (iii) it does not require users to change their modus operandi: scripts can be *outside* of a controlled environment and neither changes to the source code nor a version control system are required; (iv) it provides support for different kinds of analyses over the captured provenance data (graph-based, diff-based, and query-based); and (v) it simplifies reproducibility, allowing scientists to exchange provenance by sharing the *.noworkflow* directory with their peers. noWorkflow is available as open source software at https://github.com/gems-uff/noworkflow. Preliminary experiments show that its overhead is not burdensome.

One direction we plan to explore in future work is how to integrate provenance at different levels (e.g., operating system level with function level). We also plan to further investigate techniques for summarizing and visualizing provenance graphs [11,14], including all three types of provenance, as well as for contrasting different trials [15]. Last, but not least, we note that graph-based provenance analysis opens a vast range of opportunities for automated analysis, such as: reverse engineering workflows from scripts; optimizing scripts by either refactoring slow functions or running data mining algorithms to extract recurring execution patterns; identifying flaws in script execution; and showing the script evolution over time.

Acknowledgments. This work was supported in part by CNPq, FAPERJ, and the National Science Foundation (CNS-1229185, CNS-1153503, IIS-1142013).

References

1. Bochner, C., Gude, R., Schreiber, A.: A python library for provenance recording and querying. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 229–240. Springer, Heidelberg (2008)
2. Cheney, J., Ahamed, A., Acar, U.A.: Provenance as dependency analysis. Math. Struct. Comput. Sci. **21**, 1301–1337 (2011)
3. Davison, A.: Automated capture of experiment context for easier reproducibility in computational research. Comput. Sci. Eng. **14**(4), 48–56 (2012)
4. Diehl, S.: Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software. Springer, London (2007)
5. Freire, J., Koop, D., Santos, E., Silva, C.: Provenance for computational tasks: a survey. Comput. Sci. Eng. **10**(3), 11–21 (2008)
6. Frew, J., Metzger, D., Slaughter, P.: Automatic capture and reconstruction of computational provenance. Concurr. Comput. Pract. Exp. **20**(5), 485–496 (2008)
7. Gavish, M., Donoho, D.: A universal identifier for computational results. Procedia Comput. Sci. **4**, 637–647 (2011)
8. Guo, P.J., Seltzer, M.: BURRITO: wrapping your lab notebook in computational infrastructure. In: TaPP, p. 7 (2012)
9. van der Hoek, A.: Design-time product line architectures for any-time variability. Sci. Comput. Program. **53**(3), 285–304 (2004)

10. Huq, M.R., Apers, P.M.G., Wombacher, A.: ProvenanceCurious: a tool to infer data provenance from scripts. In: EDBT, pp. 765–768 (2013)
11. Koop, D., Freire, J., Silva, C.: Visual summaries for graph collections. In: 2013 IEEE Pacific Visualization Symposium (PacificVis), pp. 57–64 (2013)
12. Koop, D., Santos, E., Bauer, B., Troyer, M., Freire, J., Silva, C.T.: Bridging workflow and data provenance using strong links. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 397–415. Springer, Heidelberg (2010)
13. Koop, D., Scheidegger, C.E., Freire, J., Silva, C.T.: The provenance of workflow upgrades. In: McGuinness, D.L., Michaelis, J.R., Moreau, L. (eds.) IPAW 2010. LNCS, vol. 6378, pp. 2–16. Springer, Heidelberg (2010)
14. Macko, P., Seltzer, M.: Provenance map orbiter: interactive exploration of large provenance graphs. In: TaPP (2011)
15. Missier, P., Woodman, S., Hiden, H., Watson, P.: Provenance and data differencing for workflow reproducibility analysis. Concurr. Comput. Pract. Exp. (2013). doi:10.1002/cpe.3035
16. Mouallem, P., Barreto, R., Klasky, S., Podhorszki, N., Vouk, M.: Tracking files in the kepler provenance framework. In: Winslett, M. (ed.) SSDBM 2009. LNCS, vol. 5566, pp. 273–282. Springer, Heidelberg (2009)
17. Muniswamy-Reddy, K.K., Holland, D.A., Braun, U., Seltzer, M.: Provenance-aware storage systems. In: USENIX, p. 4 (2006)
18. Neves, V.C., Braganholo, V., Murta, L.: Implicit provenance gathering through configuration management. In: SE-CSE, pp. 92–95 (2013)
19. Tariq, D., Ali, M., Gehani, A.: Towards automated collection of application-level data provenance. In: TaPP, pp. 1–5 (2012)