Provenance Integration Requires Reconciliation

Elaine Angelino
Harvard University

Uri Braun
Harvard University

David A. Holland
Harvard University

Peter Macko
Harvard University

Daniel Margo
Harvard University

Margo Seltzer
Harvard University

Abstract

While there has been a great deal of research on provenance systems, there has been little discussion about challenges that arise when making different provenance systems interoperate. In fact, most of the literature focuses on provenance systems in isolation and does not discuss interoperability – what it means, its requirements, and how to achieve it. We designed the Provenance-Aware Storage System to be a general-purpose substrate on top of which it would be "easy" to add other provenance-aware systems in a way that would provide "seamless integration" for the provenance captured at each level. While the system did exactly what we wanted on toy problems, when we began integrating StarFlow, a Python-based workflow/provenance system, we discovered that integration is far trickier and more subtle than anyone has suggested in the literature. This work describes our experience undertaking the integration of StarFlow and PASS, identifying several important additions to existing provenance models necessary for interoperability among provenance systems.

1 Motivation

We are entering the third generation of provenance research. The first generation defined provenance formally; the second generation produced a number of different kinds of provenance-collection systems at different abstraction levels. This third generation addresses interoperability. The Open Provenance Model (OPM) is a solid basis for this next generation effort [11], but it is not sufficient. Interoperability between systems requires the ability to exchange, parse and interpret provenance [3].

The OPM specification addresses the first two of these requirements by proposing a standard semantic framework so that any system can read the provenance exported by another system. It defines six provenance relationships: *used*, *wasGeneratedBy*, *wasControlledBy*, *wasTriggeredBy*, and *wasDerivedFrom*, and specifies rules for performing inference across these edges. While the OPM makes concrete the notion of multiple provenance accounts, it does not provide any framework for *reconciliation* among them. E.g., there is no explicit mechanism to indicate that entity X in one account is the same as entity Y in another account.

Through our own attempts at integrating multiple provenance systems, we have identified several areas where the OPM falls short of enabling meaningful integration and interpretation of provenance across multiple accounts. We identify new provenance relationships, *stands-for* and *version*, that extend the expressivity of the OPM to support such integration.

2 Background

Data systems increasingly collect and use provenance [4, 7, 9, 10, 12, 15, 2, 14]. Even though there has been substantial effort to standardize provenance representation [11], there has been little actual integration of provenance systems. Different systems collect provenance at differing levels of abstraction, storing this provenance in private repositories with system-specific access methods. If we could combine the provenance from multiple systems, we could pose queries and reason about workflows that span multiple systems. This integration offers functionality that no single system can provide on its own.

We cast the following discussion in the context of the StarFlow and PASS systems, because we are familiar with them, and, to the best of our knowledge, PASS is the only system that explicitly addresses integration [12]. However, provenance integration is a general need; as long as databases, workflow systems, operating systems, and document management systems all capture provenance, users will need to integrate provenance.

StarFlow [1] is a provenance-enabled Python environment for data analysis. It uses a combination of annotations, static analysis, and dynamic analysis to create data and control flow dependency graphs. It uses this provenance to support, among other things, automatic as-needed recomputation of derived data when something changes upstream. It operates at the level of single data files and individual Python functions; thus it knows to re-run an analysis if one of these entities is updated. Updates to Python source files only trigger recomputation of outputs that depend on the functions that were changed; outputs depending on other unmodified functions can be left as is. Since StarFlow tracks only the objects it explicitly manages, it cannot detect or act on changes in external Python libraries or to the Python interpreter.

The Provenance-Aware Storage System (PASS) [13] takes an approach quite different from StarFlow. PASS collects and manages provenance at the operating system level, in terms of the operating system's processes and file system objects. To PASS, a Python program in StarFlow looks like a single invocation of the Python interpreter that read many input files and produced many output files. The specific Python-level connections between those input and output files are hidden.

The example Python code below illustrates both the power and limitations of these two systems.

```
import stats

def parse(x):
    y = stats.load(x)
    return (y.info, y.data)

def cluster(x, outfile, param=None):
    y = stats.cluster(x, param)
    stats.save(y, outfile)

def pca(x, outfile):
    (w, v) = stats.pca(x)

def compare(pca_file, clust_file, outfile):
    x1 = open(pca_file)
    x2 = open(clust_file)
    y = compute_error(x1, x2)
    save(y, outfile)

def main(depends_on="dat.in", creates="dat.out"):
    (info, data) = parse("dat.in")
    if info == "pca":
        pca(data, "dat.out")
    else:
        cluster(data, "dat.out")
```

The program contains four functions, `parse`, `pca`, `cluster` and `compare`, plus a wrapper that reads an input file, `dat.in`, calls `parse`, and then conditionally calls one of `pca` or `cluster`, but never calls `compare`. Both `pca` and `cluster` are long-running functions that produce an output file, `dat.out`, whose contents vary depending on which routine writes it. If we modify `compare`, even though our Python source file has changed, StarFlow realizes that the change does not affect `dat.out` and does not indicate that the program needs to be run. If we modify either `pca` or `cluster`, then we would rerun the analysis, even if the function we modified was not the one that actually produced the current output file. In contrast, once we modify the Python source file, PASS would see only that an input used to produce `dat.out` changed, necessitating a re-run. Suppose that we modified the `stats` Python package. StarFlow does not track dependencies on external Python libraries, so it would not detect this or rebuild `dat.out`, but PASS would. Using only StarFlow, we would erroneously believe that `dat.out` was up-to-date.

One might suggest that we simply extend StarFlow to query the data PASS collects. Unfortunately, this approach neither scales nor generalizes—it requires that every provenance system know about every other provenance system, potentially creating an N by N problem. Instead, we propose that systems use the PASS layering paradigm to transmit data and its provenance among provenance-aware components, only one of which needs to interact directly with a provenance store. We use PASS as the layer that interacts with the provenance store and the PASS Disclosed Provenance API (DPAPI) for provenance transmission, but any system that supports the PASS layering model would be sufficient. This approach avoids quadratic growth in interconnections, and it provides a simple way to issue queries across the combined provenance of all provenance-aware components.

The rest of this paper discusses our experience undertaking just such an integration, focusing on the unstated assumptions we and others make in building provenance systems and the implications thereof. In Section 3 we discuss the philosophy behind PASS and how we expected it to integrate with other provenance systems. In Section 4 we introduce StarFlow and its view of dependencies. In Section 5 and Section 6 we describe our expectations and experience integrating the two systems, identifying small but important changes to existing provenance models that are crucial to enabling provenance interoperability. We conclude in Section 7.

3 PASS, layering, and the DPAPI

PASS is an operating-system based provenance collection substrate. PASS intercepts system calls, extracts provenance relationships between files and processes, and then records this provenance first to a log and then ultimately into a set of indexed databases. The PASS architecture is described in detail in earlier work [12].

While extending PASS to collect and manage provenance in network-attached storage, we realized that we needed to collect and manage provenance on both clients and servers. Upon deeper reflection, we realized that this was a more general problem, that in any real system, provenance would flow through many layers of software, potentially residing on different machines, and that each layer in that software stack has different, but important information to report about data provenance. For example, the file system may know which system libraries were used and which ones have changed recently, while a language interpreter knows exactly how particular inputs influence particular outputs. Depending on the queries posed against the resulting provenance store, either or both of those types of information might prove useful.

The second PASS prototype [12] embodies this notion in its Disclosed Provenance API (DPAPI), which makes it possible to stack provenance systems atop one another, map entities at the different layers to each other, and transmit provenance through the multiple systems. The DPAPI is appealingly simple: provenance-aware read and write calls augment transferred data with collections of provenance records that describe them; another call freezes an object, so that any future modification of that object creates a new version; two other calls write provenance to disk and facilitate later retrieval; and the last call allows a software layer to enter objects from its local environment into the provenance database.

The DPAPI is based upon a set of implicit assumptions, which we make explicit here. In later sections, we will revisit these assumptions to examine how well they hold up in the presence of integration.

Objects at any given layer have obvious mappings to one object in the layer beneath it. Different levels in a system manipulate different types of objects. For example, the operating system manipulates files and processes; languages manipulate variables and functions; databases manipulate queries and tuples or records. PASS and the DPAPI assume that although different layers manipulate different types of objects, there are obvious mappings between objects – files contain database records; processes are executions of programs or functions; files contain function definitions.

Data evolves and it is important to track versions of both processes and objects. It seems obvious that data versioning is important, because data may change over time. Even if we physically represent different versions as unique objects, it is important to track the relationship between these objects and identify them as versions. More subtly, processes also have versions. Whenever a process absorbs new input, its provenance changes, creating a new version of the process. It is important to capture these changes to accurately describe process and file dependencies.

Versions are created by explicit operations that act upon one version and produce a new one. New versions are created when some process acts upon an old version, i.e., new information may be added to an object, the object may be transformed, etc.

We turn to StarFlow and its own set of assumptions.

4 StarFlow

StarFlow is a workflow system embedded in a programming language environment, specifically Python. Its target audience is script programmers, who are comfortable performing sophisticated data analysis using Python, but are not computer scientists and are not interested in learning programming tools such as `make`. They have the same data management needs as other computational scientists: they want to be able to identify the modules and externalities on which their tools depend; they would like to rerun their analyses only when necessary; and they would like to be able to package up a particular analysis and all its dependencies to give to a colleague.

StarFlow uses a combination of annotations, static analysis, and dynamic analysis to create data and control flow dependency graphs, and then uses these graphs to answer questions such as "On what code/data does this output depend?" and "What would I need to execute to bring this file up to date?"

Users annotate their analysis routines by explicitly stating the input files that an analysis uses and the output files it produces. Then, the StarFlow static analyzer constructs the data and control flow dependency graphs. At runtime, StarFlow's dynamic analyzer intercepts file read and write operations to verify that the files being accessed match those described by the annotations or to create dependencies not captured by annotations. Python functions address the queries described above by traversing the dependency graph.

StarFlow also makes some implicit assumptions that we make explicit here.

Users will use the Python interpreter as their execution environment. The anticipated StarFlow user does not manually execute their functions once they have written and annotated them. Instead, they rely on the StarFlow command `Update`, which re-executes only what is necessary. When users want to see what functions need to be run, they use `ShowUpdates`.

Even before users have run an analysis, they may want to understand the dependency network. StarFlow's annotations combined with static analysis allow users to explore their dependency networks even before they have ever executed a workflow.

Dynamic analysis provides error checking capabilities, but it not fundamental to dependency tracking.

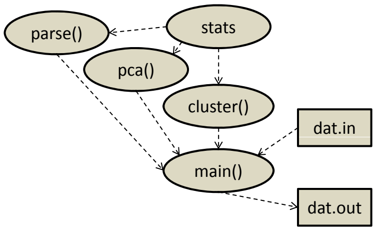

Figure 1: Dependency graph extracted by StarFlow for the file dat.out. Arrows are in the direction of information flow. Files are rectangles; functions or modules are ovals. The function main depends on the file dat.in and creates dat.out. It uses the functions parse, pca, and cluster, which all depend on the stats library.

StarFlow assumes that annotations and static analysis capture data and control flow and that the only purpose of dynamic analysis is to verify that the functions behave as annotated with respect to their inputs and outputs.

The only items of interest are Python modules, user code and data files.

Having now made the assumptions of the two systems explicit, we can dive into a discussion of our plan for integrating the two systems and the inevitable clashes that arose from having different sets of assumptions.

5 The Plan

As StarFlow was already creating a dependency graph, it seemed straightforward to have StarFlow call the DPAPI, thereby linking Python functions to the files containing them therebyeusing PASS as the storage substrate for the StarFlow provenance. Our theory was that each addition to StarFlow's dependency network would result in a provenance-aware write to the file system and that StarFlow's recomputation functionality would turn into simple queries over the PASS data.

Returning to the code example, Figsures 1 and 2 show the dependency graphs that StarFlow and PASS create, respectively, for the output file dat.out. We wanted to create objects to represent the ovals in Fig. 1 and then transmit to PASS the edge relationships. Rather than coding complicated graph traversals in StarFlow, it could issue simple queries via the PASS query engine.

The combined system would provide the functionality discussed in Section 2. StarFlow's function-level tracking helps us avoid needlessly re-executing when a function that we do not call gets changed, and PASS's tracking of files outside StarFlow's purview prevents us from missing relevant updates. A query for changes in the ancestry of an output file can traverse a combined provenance graph produced by StarFlow and PASS.

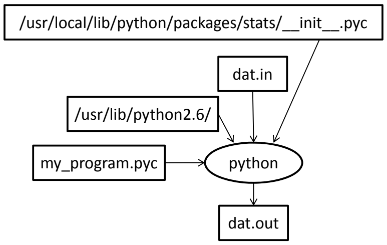

Figure 2: Dependency graph extracted by PASS during runtime for the file dat.out.

6 The Reality

This integration was different from what we anticipated.

6.1 Referencing Non-Existent Objects

The first challenge we encountered was in trying to record dependencies created by StarFlow's static analysis phase. During static analysis, StarFlow may try to create dependencies for files that do not yet exist. That is, when StarFlow observes that main creates dat.out, it creates a dependency between the StarFlow objects representing the function main and the output, dat.out. However, dat.out corresponds to a PASS object that does not yet exist. The DPAPI assumes that layers will want to create mappings to objects in different layers, but here, the corresponding object does not yet exist.

StarFlow's static analysis is doing something other than collecting provenance. Provenance is a record of what *happened*, while static analysis predicts what *will happen*; that prediction may be incomplete or context insensitive. It is incomplete if the program uses constructs that are undecidable, e.g., Python's eval. It is context insensitive in that during static analysis, we do not know whether the pca function or the cluster function will be called, and records both as dependencies. This approach captures negative information flow. It does not produce a precise dependency graph, but rather the superset of graphs that might arise from execution.

Although this case seems potentially problematic, and in fact, did give us cause to question whether our interfaces were sufficient, the DPAPI in both its implementation and design addresses the issue neatly. StarFlow's dat.out and PASS's dat.out are not the same object, but do share a special relationship, not captured in existing provenance models. StarFlow's dat.out is an abstract object corresponding to an intended output; PASS's dat.out is the output of a specific execution. The missing piece is a way to associate the abstract and concrete representations of these objects, and this is where StarFlow's dynamic analysis comes to our aid.

When we execute the main program, StarFlow's sys-

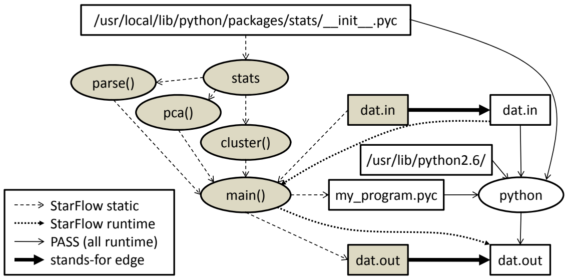

Figure 3: Combined dependency graph from StarFlow layered on top of PASS for the file `dat.out`. The two different line types distinguish those edges created during static analysis (connecting shaded shapes) and those created by PASS's dynamic analysis (unshaded shapes). The bold edges are created during StarFlow's dynamic analysis.

tem call interception detects the creation (or modification) of `dat.out`. It then traverses the current call stack until it finds a function that has a `creates` dependency. When it locates the dependency in `main`, it creates two new provenance edges: one is a normal output edge between the StarFlow object for `main` and the newly created PASS object for `dat.out`. The second is an edge between StarFlow's `dat.out` and PASS's `dat.out`. Figure 3 shows the graph resulting from the integration of StarFlow and PASS provenance, highlighting the edges created during this dynamic analysis.

This newly created edge is not a version edge nor is it an input or output edge. Instead, it is a *stands-for* edge, indicating that an object in one account represents an object in a different account. This is the provenance equivalent of a symbolic link. These new edges identify relationships that are not represented in existing provenance models, yet are crucial to facilitating interoperability and reconciliation of multiple accounts.

This new edge type solves a second, more subtle problem. Each successful analysis execution create a new version of the output file, `dat.out`. However, static analysis identifies only a single instance of `dat.out` when it determines that `main` creates `dat.out`. Without this new edge type, there is no obvious way to reconcile the single-version view of the world with the multi-version view of the world.

Some workflow systems distinguish between workflow abstractions or templates and concrete workflow instances [5, 8], and in doing so deal with an example of this problem. A concrete workflow instance is derived from a workflow template plus a set of concrete data inputs. A specific object in a workflow template corresponds to an object in each of possibly many workflow instances. For example, the output of the first step of a workflow template may correspond to a new file for each concrete instantiation. These semantics describe a workflow at multiple layers of abstraction or granularity, and *stands-for* edges can be used to explicitly reconcile across these layers.

6.2 Version Disconnection

Our previous discussion suggests that version edges are also important in provenance modeling and representation. PASS assumes that versions result from a process acting upon a previous version, creating a new version. The notion of multiple versions is important to many provenance systems. For example, VisTrails tracks versions of workflows [4], and some of the vocabularies introduced by the W3C Provenance Incubator Group include the notion of version [6].

Returning to our example, suppose that someone installs a newer version of the Python library, `stats`. The installation process never reads the existing file; it simply overwrites it. Similarly, text editors frequently write new data into temporary files and then move them into place, replacing the original. While PASS tracks this process and captures the relationships between the old file, installation, and the new file, it does not know and cannot record that the new file is a later version of the original.

To some extent, this is an implementation failure in PASS that can be addressed. The file system already detects when an operation replaces an existing object because it needs to delete that replaced object. We could create provenance records to associate the over-writing file with the over-written file. Unfortunately, it is not always the case that this association is correct; it is application-specific whether the two files are related. In the examples above, the two objects are obviously related. However, when a user saves an editor buffer to a file named `tmp`, she probably does not want to associate the new contents of `tmp` with the old.

This conundrum suggests that we need to allow provenance systems to explicitly create version edges between two existing nodes. We want to allow the provenance engine at one layer to specify meaningful edges between objects at lower layers. Put another way, this is a case where the knowledge of the relationship between objects is not present at the layer that manages those objects. The idea that certain meaningful relationships between objects can *only* be created by an agent external to a specific provenance system does not appear in existing provenance models.

6.3 Provenance of Provenance

Allowing external agents (i.e., provenance systems at different layers) to add provenance brings into question its quality or trustworthiness. When we developed the DPAPI, we encountered the question of whether to record the identity of such an agent. This is the provenance of provenance, which introduces obvious problems – how do we avoid infinite recursion?

We need to distinguish provenance from different sources; otherwise we could fall victim to provenance spam, where processes add meaningless or even deceptive provenance. We believe that the right approach is to rely on the operating system's definition of identity to describe the source of provenance records.

Unless otherwise noted, provenance data in PASS was produced by PASS. The emerging provenance standard, the OPM, contains the notion of a controlling process, and while the intended use is to describe a process that invokes another process, it also accurately describes the agent contributing provenance. Thus, we can use PASS-generated provenance to identify external provenance by creating *wasControlledBy* edges between the process adding provenance and the generated provenance records. By taking advantage of the operating system's existing notion of processes and existing concepts from OPM, we can identify the source of DPAPI additions without falling prey to infinite recursion. Other provenance systems will need to make similar decisions – a system can choose to augment provenance passed to it with its own notion of identity or it can assume responsibility for the provenance, allowing PASS to attribute the provenance to the process in which it was created.

7 Conclusions

We have identified two new edges that we believe are fundamental to provenance and interoperability: *stands-for* and *version*. These edges are absent from the OPM and are not represented in other provenance systems.

The *stands-for* edge allows agents to state that two objects are the same. In addition to addressing the questions that arose in our experience, this edge also addresses the issuing of reconciling multiple accounts of an event, stating that the accounts really do represent the same event.

The *version* edge is a stronger statement than the OPM *wasDerivedFrom* edge. We find versions essential for representing both objects (OPM artifacts) and processes [13]. OPM's *wasDerivedFrom* edges capture input relationships, while *version* edges embody a notion of object identity that is quite different.

These additional requirements for interoperability are not overly burdensome. From an implementation perspective, the DPAPI is powerful enough to express them. From a modeling perspective, the new edge types are easily incorporated into existing provenance models, such as the OPM. We think that layering is the right way to integrate provenance systems and that small, but important, modifications to existing models enable integration.

References

[1] Angelino, E., Yamins, D., and Seltzer, M. Starflow: A script-centric data analysis environment. In *Proceedings of the Third International Provenance and Annotation Workshop (IPAW 2010)* (2010).

[2] Bose, R., and Frew, J. Composing lineage metadata with xml for custom satellite-derived data products. In *Proceedings of the Sixteenth International Conference on Scientific and Statistical Database Management* (2004).

[3] Braun, U. J., Seltzer, M. I., Chapman, A., Blaustein, B., Allen, M. D., and Seligman, L. Towards query interoperability: Passing plus. In *Proceedings of the 2nd conference on Theory and practice of provenance* (Berkeley, CA, USA, 2010), TAPP'10, USENIX Association, pp. 3–3.

[4] Callahan, S. P., Freire, J., Santos, E., Scheidegger, C. E., Silva, C. T., and Vo, H. T. Vistrails: visualization meets data management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM, pp. 745–747.

[5] Gil, Y., Ratnakar, V., Deelman, E., Mehta, G., and Kim, J. Wings for pegasus: creating large-scale scientific applications using semantic representations of computational workflows. In *Proceedings of the 19th national conference on Innovative applications of artificial intelligence - Volume 2* (2007), AAAI Press, pp. 1767–1774.

[6] Group, W. P. I. Provenance xg final report: W3c incubator group report. `http://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/`, December 2010.

[7] Ikeda, R., and Widom, J. Panda: A system for provenance and data. In *TaPP '10* (2010), Stanford InfoLab.

[8] Koop, D., Santos, E., Bauer, B., Troyer, M., Freire, J., and Silva, C. T. Bridging workflow and data provenance using strong links. In *Proceedings of the 22nd international conference on Scientific and statistical database management* (Berlin, Heidelberg, 2010), SSDBM'10, Springer-Verlag, pp. 397–415.

[9] Kuehn, H., Liberzon, A., Reich, M., and Mesirov, J. P. Using genepattern for gene expression analysis. *Curr. Prot. in Bioinformatics* (2008), 7.12.1–7.12.39.

[10] McPhillips, T., Bowers, S., Zinn, D., and Ludaschera, B. Scientific workflow design for mere mortals. *Future Generation Computer Systems 25*, 5 (2009), 541–551.

[11] Moreau, L., Kwasnikowska, N., and den Bussche, J. V. The foundations of the open provenance model. April 2009.

[12] Muniswamy-Reddy, K.-K., Braun, U., Holland, D., Macko, P., Maclean, D., Margo, D., Seltzer, M., and Smogor, R. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*.

[13] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.

[14] Simmhan, Y. L., Plale, B., and Gannon, D. A framework for collecting provenance in data-centric scientific workflows. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services* (2006).

[15] Taylor, J., Schenck, I., Blankenberg, D., and Nekrutenko, A. Using galaxy to perform large-scale interactive data analyses. *Curr. Prot. in Bioinformatics* (2007), 10.5.1–10.5.25.