Provenance-Enabled Automatic Data Publishing

James Frew, Greg Janée, and Peter Slaughter

Earth Research Institute
University of California, Santa Barbara
{frew,gjanee,peter}@eri.ucsb.edu
http://eri.ucsb.edu

Abstract. Scientists are increasingly being called upon to publish their data as well as their conclusions. Yet computational science often necessarily occurs in exploratory, unstructured environments. Scientists are as likely to use one-off scripts, legacy programs, and volatile collections of data and parametric assumptions as they are to frame their investigations using easily reproducible workflows. The ES3 system can capture the provenance of such unstructured computations and make it available so that the results of such computations can be evaluated in the overall context of their inputs, implementation, and assumptions. Additionally, we find that such provenance can serve as an automatic "checklist" whereby the suitability of data (or other computational artifacts) for publication can be evaluated. We describe a system that, given the request to publish a particular computational artifact, traverses that artifact's provenance and applies rule-based tests to each of the artifact's computational antecedents to determine whether the artifact's provenance is robust enough to justify its publication. Generically, such tests check for proper curation of the artifacts, which specifically can mean such things as: source code checked into a source control system; data accessible from a well-known repository; etc. Minimally, publish requests yield a report on an object's fitness for publication, although such reports can easily drive an automated cleanup process that remedies many of the identified shortcomings.

Keywords: provenance, publishing, curation

1 Introduction and Background

In computing environments there is often tension between freedom of expression and exploration on the one hand, and constraint and assertion on the other. More precisely, the creation of a computational artifact may require creativity and freeform exploration, but at some end point the environment requires that certain assertions be true. The question is, where, when, and how should those assertions be enforced?

1.1 Programming Environments

Programming languages provide a familiar example for many. Creating a program is a fundamentally creative activity, but in the end, the computational

environment requires that the program be syntactically and semantically valid[1] to be executed. Different programming environments have take different approaches to enforcing program validity. At the flexibility-maximizing end of the spectrum, many environments allow programs to be created with near-complete freedom using text editors. Assertion checking about programs is then deferred to a compilation step (for compiled languages) and/or to execution time (for interpreted languages). Other environments, particularly "visual" or GUI-based environments, provide syntax-aware and even limited-semantics-aware editors that constrain expressions during creation. These environments attempt to maintain the assertion that the program is at least syntactically correct during its creation. In a kind of *reductio ad absurdum*, we might imagine an environment that requires that a program be valid at *every* step of its construction, thus obviating the need for any later assertion checking. But this is a direction that programming environments have not gravitated toward. Even in the visual programming environments, there is a recognized need that programmers be allowed to create expressions that violate language rules and semantics, at least temporarily. Code may be sketched out initially; there may be references to entities that don't exist yet; test code insertions may create deliberate errors; and so forth.

1.2 Scientific Programming

The creation of science data products is analogous. In the end, we require that a data product be of sufficient quality, and that it have been produced using sufficiently rigorous and repeatable methods, that it is worthy of entering the scientific record. But creating the product requires experimentation and creativity, particularly in the formative stages, and many assertions that we would like to be true at the end will not be true during the entire process of creation. We may, for example, use false or test data, shortcut production steps out of expediency, and many other things of that nature. Moreover, computational science data is produced by software, so the points in the preceding section regarding software creation apply as well.

Scientific workflow environments provide assertion support similar to syntax-aware programming language editors in that desirable end assertions are maintained through the whole process of development. However, we argue that much data production is done outside workflow systems, and even for those scientists working in workflow systems, the flexibility scientists require means that they may drop out of the environment to perform one-off experiments.

1.3 Publishing Science Data Products

By "data publication" we mean the packaging and distribution of scientific datasets. As in traditional publication of scientific literature, data publication

[1] Semantically valid only in the sense that all language requirements are satisfied; we consider no deeper semantics here.

involves selecting and formatting content; naming and attributing the resulting artifact, and then making it available via well-known distribution channels.

Significantly, only the first step–selection–is an intrinsic result of scientific computation, namely as the computation's result set, and thus happens automatically. The remaining steps are often an added burden at publish time. Publication standards often dictate a specific format (e.g., GeoTIFF[2] or KML[3] for Earth science array and vector data, respectively) that differs from that used internally in a scientific computation environment. Similarly, published data objects must often adhere to naming conventions (e.g., DOIs for datasets or persistent URIs for data granules or services) that differ from more convenient internal names (e.g., filenames or database queries). These formats in turn often dictate or enable specific distribution mechanisms (e.g., geospatial web services[4].)

In addition to requiring specific formats and naming conventions, data publication requires the availability of additional descriptive information. Much of this *metadata* is available as an automatic consequence of the computational environment (e.g., array dimensions, creation datetimes, etc.), but much traditionally is not (e.g., descriptive text, names of responsible parties, etc.) and must be discovered or supplied as part of the publication process. These metadata collectively serve the attribution role analogous to authorship in traditional publishing. Historically, a key missing piece of this attribution has been the connections between a data object and its antecedents, such as can now be supplied by provenance.

2 Provenance-Enabled Automatic Data Publishing

We believe that provenance can greatly simplify the transition between experimental and published products by simplifying the assembly and validation of the metadata necessary to publish a science data object. Key to our approach is exploiting the ES3 system, which captures the necessary provenance without imposing restrictions on relatively unstructured experimental environments. In this section we define provenance as ES3 implements it, describe the ES3 system, and then describe how ES3-collected provenance can help automate the publication process.

2.1 Computational Provenance

Computational provenance refers to knowledge of the origins and processing history of a computational artifact such as a data product or an implementation of an algorithm [1]. Provenance is an essential part of metadata for Earth science data products, where both the source data and the processing algorithms change over time. These changes can result from errors (e.g., sensor malfunctions or incorrect algorithms) or from an evolving understanding of the underlying systems

[2] `http://trac.osgeo.org/geotiff`
[3] `http://opengeospatial.org/standards/kml`
[4] `http://ogcnetwork.net/services`

and processes (e.g., sensor recalibration or algorithm improvement). Occasionally such changes are memorialized as product or algorithm "versions," but more often they manifest only as mysterious differences between data products that one would otherwise expect to be similar. Provenance allows us to better understand the impacts of changes in a processing chain, and to have higher confidence in the reliability of any specific data product.

2.2 Transparent Provenance Collection with ES3

ES3 is a software system for automatically and transparently capturing, managing, and reconstructing the provenance of arbitrary, unmodified computational sequences [3]. *Automatic* acquisition avoids the inaccuracies and incompleteness of human-specified provenance (i.e., annotation.) *Transparent* acquisition avoids the computational scientist having to learn, and be constrained by, a specific language or schema in which their problem must be expressed or structured in order for provenance to be captured.

Unlike most other provenance management systems [1, 6, 9], ES3 captures provenance from running processes, as opposed to extracting or inferring it from static specifications such as scripts or workflows. ES3 provenance management can thus be added to any existing scientific computations without modifying or re-specifying them.

ES3 models provenance in terms of processes and their input and output files. We use "process" in the classic sense of a specific execution of a program. In other words, each execution of a program or workflow, or access to a file, yields new provenance events.

Relationships between files and processes are observed by monitoring read and write accesses. This monitoring can take place at multiple levels: system calls (using `strace`), library calls (using instrumented versions of application libraries), or arbitrary checkpoints within source code (using automatically invoked source-to-source preprocessors for specific environments such as IDL[5]). Any combination of monitoring levels may be active simultaneously, and all are transparent to the scientist-programmer using the system. In particular, system call tracing allows ES3 to function completely independently of the scientist's choice of programming tools or execution environments.

An ES3 provenance document is the directed graph of files and processes resulting from a specific invocation event (e.g., a "job"). Nested processes (processes that spawn other processes) are correctly represented. In addition to retrieving the entire provenance of a job, ES3 supports arbitrary forward (descendant) and/or reverse (ancestor) provenance retrieval, starting at any specified file or process.

ES3 is implemented as a provenance-gathering client and a provenance-managing server (Figure 1). The client runs in the same environment as the processes whose provenance is being tracked.

[5] `http://www.ittvis.com/idl`

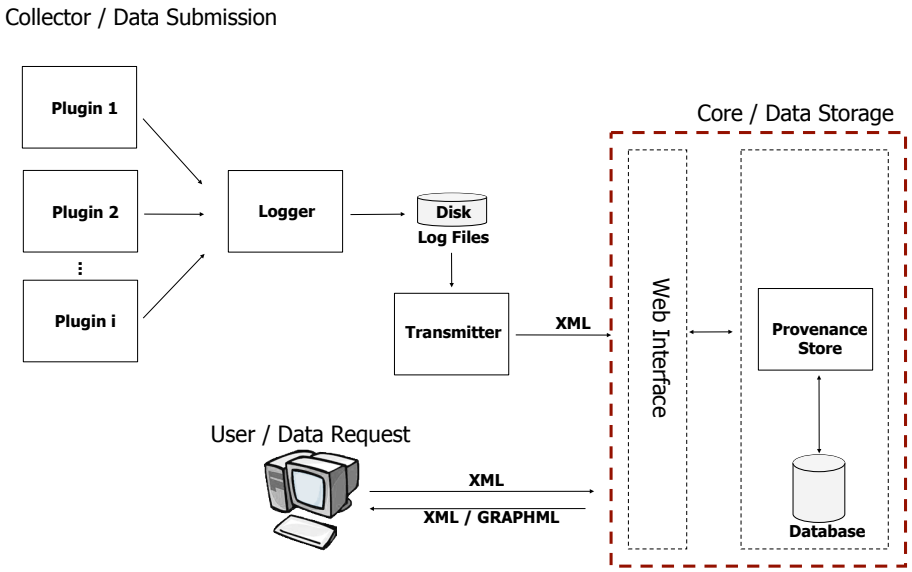

Fig. 1. ES3 architecture

The client is a set of *logger* processes that intercept raw messages from the various monitoring modes (*plugins*) and write them to log files. A common *transmitter* client asynchronously scans the log files, assembles the provenance events into a time-ordered stream, assigns UUIDs to each file and process being tracked, and submits a raw provenance report to the *ES3 core* (server).

The ES3 core is an XML database with a web service middleware layer that supports insertion of file and provenance metadata, and retrieval of provenance graphs. File metadata allows ES3 to track the one-to-many correspondence between external file identifiers (e.g., pathnames) and internal (UUID) references to those files in provenance reports. Provenance queries cause the ES3 core to assemble a provenance graph (by linking UUIDs) starting at a specified process or file and proceeding in either the ancestor or descendant direction. The graphs are returned serialized in various XML formats (ES3 native, GraphML [2], etc.), which can be rendered by graph visualization clients (e.g., Graphviz[6]; yEd[7]).

ES3's native provenance model is a proper subset of the Open Provenance Model (OPM) [7]. ES3 *processes* and *files* correspond to OPM *processes* and *artifacts*, respectively. ES3 does not support OPM's notion of *agent*, since this entity role cannot be transparently determined from the events ES3 monitors.

[6] `http://graphviz.org`
[7] `http://yworks.com/yed`

2.3 Using Provenance to Drive Publication Decisions

The provenance collected by ES3 represents the complete processing history of a digital object and its antecedents (assuming those antecedents were likewise generated under ES3's observation, or have provenance provided by a similarly capable environment). While extremely valuable as metadata [6] in its own right, this provenance can also be exploited to help determine an object's fitness for publication.

Our basic assumption is that a large part of what determines an object's fitness for publication is the *process* by which that object was produced, and the *components* which were combined or transformed in order to produce it. An object produced by scientific codes with known significant bugs, or from source datasets with known errors, may not be suitable for publication, regardless of how well formatted, documented, and presented it is. Or, even if the codes and data used to produce an object are demonstrably acceptable, they may not be suitably *curated*, and this lack of a guaranteed level of future availability to users of the object may be sufficient to disqualify it for publication.

A provenance-driven publication process thus involves traversing a candidate object's provenance to some suitable depth and evaluating whether the object's antecedents justify a decision to publish the object. We envision, and are now prototyping, this decision process as *assertion-based*, with different sets of assertions applied to different categories of antecedents (e.g., programs vs. data files). Managing the assertions separately from the provenance allows us to tailor the assertions to differing levels of rigor—for example, publishing data to a small community of experts may impose fewer constraints on the data's antecedents than publishing to a public repository.

2.4 Comparable Work

One of the primary rationales for collecting provenance information is to enable reproducibility of a process [4]. Our approach to data publication implies that objects that satisfy our publication criteria are more likely (albeit not guaranteed) to be reproducible. An alternative would be to attempt to recreate the entire environment in which a process executed, in order to enable its precise reproduction.

We are aware of two unpublished systems which take this approach. CDE [5] uses system call tracing to package the executable code, input data, and Linux environment (system libraries, language interpreters, etc.) necessary to run a process into a single distributable object, runnable on any Linux system. `lbsh` [8] uses command-line scraping to determine a process' inputs and outputs, and optionally prepare a (less comprehensive than CDE) package intended to facilitate the process' re-execution.

Our approach is less concerned with *immediate* reproducibility, and more with *long term* reproducibility. Instead of immediately preparing a package that will guarantee an object's reproducibility, we strive to ensure, through our publishability assertions, that such a package *could be* prepared as needed at some arbitrary future time.

3 Worked Example: Publishing Global Ocean Color Data

To illustrate our proposed approach, let us consider a sample computational process, in this case a process that derives an ocean color product from an antecedent product. Figure 2 is a greatly simplified (i.e., many inputs and outputs have been elided) version of an automatically gathered ES3 provenance trace. It depicts a shell script (the outer box) invoking an IDL interpreter (the inner

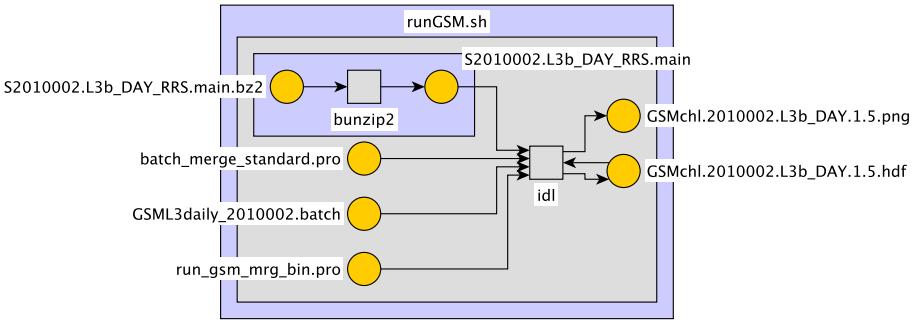

Fig. 2. provenance trace

box), which in turn executes an IDL program, which in turn does the real work: reads an input data file, writes an HDF output file, reads the HDF file back in, and finally produces a PNG preview file.

This type of provenance trace is entirely typical and there is nothing in its structure to indicate its purpose: it could be producing a product to be published, or it could be any manner of test or experiment. Our assertion checking process is initiated only when an output is identified by the scientist as a candidate for publishing. That is, many such traces may be automatically captured by ES3, and most at some point culled, and it is only the act of publishing that triggers any deeper examination of the traces.

There are no fixed assertions to be checked, as the assertions are configurable and will depend greatly on the context. Generally, however, assertions fall into two categories:

- Provenance, or, What did we do? Do we have sufficient information describing the processing that would allow a reader to unambiguously re-create the processing, in principle if not in actuality?
- Confirmation, or, Did we really do what we think we did? It is all too easy to mistakenly operate on the "wrong" file, because the file has the same name as the correct file or because the file's contents have changed without our knowledge. We believe that a publishing system should be able to catch such mistakes.

Returning to our sample process in Fig. 2, the provenance assertions include:

- Does the input data file have a provenance statement? Does it have, for example, ES3 or OPM metadata or another, less structured statement of its source?
- Is the IDL code held in a source code repository, and do the versions of the code used in the processing correspond to committed (i.e., registered) versions in that repository?
- Has information about the computational environment been recorded, e.g., the version of IDL?

The confirmation assertions include:

- If a checksum of the input data file's contents was recorded at the time of its download from an external source, does the checksum match the checksum at the time of the file's use?
- If a correspondence has been made between source code versions and the overall data product version, were the "correct" versions of the source code used?

Notice how the preceding assertions are relevant only when publishing a data product. During development and experimentation there are any number of reasonable reasons why they may be violated.

Implementing such publish-time assertion checking in ES3 requires that we be able to distinguish and characterize different kinds of files participating in the computational process. In a raw ES3 trace, all files referenced during the process are equivalent in ES3's view, being distinguished only as inputs or outputs. But the types of publish-time assertions we want to make depend on the different roles files play in the computational process. What distinguishes an input data file from a source code file in Fig. 2 is that the former was obtained from an external source and is expected to have a provenance statement, while the latter was locally developed and is managed in a source code system, and thus the assertions to be checked are correspondingly different. In our proposed framework, such distinctions can be made ad hoc and on a file-by-file basis, but we expect that general configuration rules will obviate the need for most such fine-grained specification. For example, input data files may be defined to be any files residing in a designated directory. Residency in source code systems can be determined opportunistically, by interrogating source code repositories (e.g., the local CVS server) and by looking for repository artifacts (e.g., RCS and Mercurial repository directories).

A generic restriction on the publication process is the *accessibility* of the published object's antecedents. We assume that any files we wish to check against our publication rules are either directly accessible to the publication process, or have sufficiently complete and trustworthy provenance that assertion checking can be based on metadata alone.

4 Conclusion

Our provenance-driven data publication scheme is a work-in-progress. We are implementing it as a stand-alone `publish` service that, given a digital object, will proceed as follows:

1. Request the object's provenance from ES3.
2. For each antecedent object, test that object against the appropriate publication assertions.
3. List which assertions were violated.
4. If additionally directed, and where possible, take automatic actions (e.g., check code into a repository) to remedy assertion violations.

Of course there will be situations where step 4 fails to automatically render an object fit for publication—for example, publication may require the availability of antecedent files or programs that have been deleted since the object was created. In such cases, we believe the "fitness report" generated by `publish` will be invaluable documentation, especially if, lacking suitable alternatives, the objects must be published anyway.

References

1. Bose R., Frew, J.: Lineage Retrieval for Scientific Data Processing: A Survey. ACM Computing Surveys. 37:1, 1–28 (2005) doi:10.1145/1057977.1057978
2. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report: structural layer proposal. In: Mutzel, P., Junger, M., Leipert, S. (eds.) Graph drawing. LNCS, vol. 2265, pp. 109–112. Springer, Berlin (2002) doi:10.1007/3-540-45848-4
3. Frew, J., Metzger, D., Slaughter, P.: Automatic capture and reconstruction of computational provenance. Concurrency and Computation: Practice and Experience. 20, 485–496 (2008) doi:10.1002/cpe.1247
4. Gil, Y., Cheney, J., Groth, P., Hartig, O., Miles, S., Moreau, L., da Silva, P.P.: Provenance XG Final Report. W3C Provenance Incubator Group, `http://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/` (2010)
5. Guo, P.: CDE: Automatically create portable Linux applications. `http://www.stanford.edu/~pgbovine/cde.html`
6. Moreau, L.: The Foundations for Provenance on the Web. Foundations and Trends in Web Science. 2:2–3, 99–241 (2010) doi:10.1561/1800000010
7. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., Van den Bussche, J. (2010) The Open Provenance Model core specification (v1.1). Future Generation Computer Systems. (In Press) (2010) doi:10.1016/j.future.2010.07.005
8. Osterweil, E., Zhang, L.: lbsh: Pounding Science into the Command-Line. `http://www.cs.ucla.edu/~eoster/doc/lbsh.pdf`
9. Simmhan, Y. L., Plale, B., Gannon, D.: A survey of data provenance in e-science. ACM SIGMOD Record. 34, 31–36 (2005) doi:10.1145/1084805.1084812