A General-Purpose Provenance Library

Peter Macko
Harvard University

Margo Seltzer
Harvard University

Abstract

Most provenance capture takes place inside particular tools – a workflow engine, a database, an operating system, or an application. However, most users have an existing toolset – a collection of different tools that work well for their needs and with which they are comfortable. Currently, such users have limited ability to collect provenance without disrupting their work and changing environments, which most users are hesitant to do. Even users who are willing to adopt new tools, may realize limited benefit from provenance in those tools if they do not integrate with their entire environment, which may include multiple languages and frameworks.

We present the Core Provenance Library (CPL), a portable, multi-lingual library that application programmers can easily incorporate into a variety of tools to collect and integrate provenance. Although the manual instrumentation adds extra work for application programmers, we show that in most cases, the work is minimal, and the resulting system solves several problems that plague more constrained provenance collection systems.

1 Introduction

Adoption of provenance among computational scientists is low, because most existing systems require that users adopt a particular toolset. For example, PASS [9] and Story Book [10] require users to run a modified version of the operating system, StarFlow [4] requires the use of Python, Kepler [2], VisTrails [5], and many other workflow engines require the use of a particular engine – but users are often reluctant to change their work environments. Moreover, many provenance systems require users to conduct *all* their data analysis in a single programming language or an environment, while in reality, heterogeneity is the norm. For example, it is not uncommon to process data produced by a simulation written in C or Java using a Perl script and then process it further using a shell script or a combination of analysis programs.

Taking these and several other previously identified lessons [3] into account, we developed the Core Provenance Library (CPL), which can be integrated easily into existing scripts and applications. The CPL is portable across operating systems, has multiple language bindings, and can use any one of a variety of database backends. Applications can use the library's API to disclose provenance by creating provenance objects and describing data and control flow between them. Existing provenance and workflow systems can use the same API to disclose provenance to the CPL, so that in the end, the provenance from different tools in the system is together and integrated. The library takes care of persistent storage, cycle detection and resolution, query, and visualization in addition to providing an API to access the collected provenance programmatically.

Even though the manual instrumentation adds extra work, we find that such work requires little effort and the resulting system solves several problems that plague other systems [3], such as version disconnect and provenance integration. It enables applications to export provenance at whatever granularity is most logical to them. For example, a provenance system that automatically collects provenance might not know that two objects, which are not directly related from its viewpoint, are, in fact, an older and a newer version of the same object. We discuss this further in Section 2.1.

2 Disclosing Provenance

The main design considerations of the provenance library are simplicity and the ability to integrate provenance across multiple applications easily.

An application using the library calls `cpl_attach` to initialize the library for the current process and to attach it to the appropriate storage backend. The application then uses the library to obtain unique IDs corresponding to its logical objects, such as actors in workflow engines or tables in databases, using `cpl_create_object` for new objects or `cpl_lookup_object` for extant objects. It then calls `cpl_data_flow` whenever an object uses data from another object, such as when an actor in a workflow reads the contents of a file or when it transmits a computed value to another actor in the workflow. The application uses `cpl_control` to indicate that an object influences the execution of a task without passing any data; for example, a loop operand in a workflow engine influences execution inside the loop. The application can attach arbitrary key/value pairs to objects using `cpl_add_property`. Finally, the application calls `cpl_detach` before it exists (such as via `atexit`). There is no need to clean up any other state, such as closing or otherwise disposing of provenance objects; the library does this automatically. This automated cleanup was an explicit design decision to simplify integration of the library to existing applications. Appendix A presents the API in more detail.

The CPL internally assigns a globally unique ID to each object. Externally, it identifies an object using a combination of three fields:

- **Namespace**: The name of an application or a component of the system to which the object belongs. For example, an actor in a workflow engine should be defined in the engine's private namespace, while a file should be defined in the file system's namespace. This field is important to prevent name collisions between unrelated applications and to facilitate sharing of objects from cooperating applications.
- **Name**: The local name of the object.
- **Type**: The type of the object, such as whether it is a file or a process. This can be a standard (CPL-defined) or a user-defined type.

If there are multiple objects with the same namespace, name, and type, the lookup operation returns the most recently created object by default; others can be retrieved using `cpl_lookup_object_ext`.

The library defines several special namespaces for shared objects, such as for files that reside on a file system, and special convenience functions to access some of them. For example, `cpl_lookup_file` looks up (and optionally creates, if necessary) a unique ID that can be used to reference a specific file. This facilitates provenance integration between non-cooperating applications, since it does not require them to pass IDs of their respective objects to each other.

The CPL also tags each object with the MAC address of the machine on which it was created, so that an application can easily share an object and its provenance over a network. The library automatically versions each object using the Cycle Avoidance algorithm [9] to avoid creating cycles in provenance and so that applications can reference older versions of objects as appropriate.

The library annotates each provenance record with *provenance of provenance*, so that each piece of provenance can be attributed to a specific instance of an application running on a specific computer. It records the user name, time, name of the binary image, command line arguments, and the MAC address of the machine. This is useful, for example, to answer the "where does this piece of provenance come from?" question, to list provenance objects produced by a specific instance of an application, and to get information about an application that produced a specific object (see Section 5.1).

2.1 Advantages

A critical characteristic of this approach is that applications can disclose provenance to the CPL in a way that is most logical and appropriate for them, eliminating several problems that plague other systems [3].

The first problem systems frequently encounter is version disconnect: For example, most text editors save a file by first creating a temporary file and then moving it into place, replacing the older version of the file. Systems such as PASS [9] that collect provenance by observing system calls frequently miss this connection and treat the old and the new version of the document as different objects, even though the version connection is obvious to the user. An editor that uses the CPL can choose whether to treat a new version of the document as a new version of the old document or as a separate object connected via a data dependency relationship, depending on what is most logical for the application.

The second problem CPL addresses is provenance integration: Cooperating applications can share and reference the GUIDs of their objects, eliminating many uses of the *alternate* relation from the W3C provenance standard [8]. If the object is a file on a file system (or a different kind of shared object), even non-cooperating applications can easily reference it uniformly.

Another problem that frequently arise is the disconnect between different notions of provenance: For example, StarFlow [4] statically analyzes a Python program to determine its call graph and the files that it reads and writes, instead of capturing dynamic information at runtime. It can export this to the CPL, while exporting it to PASS was more difficult in our experience.

2.2 Disadvantages

While using the CPL requires effort on the part of application programmers, we have found the effort to be quite small; we present anecdotal evidence in Section 5.

The second disadvantage is that the library cannot guarantee that the provenance is complete or even correct. Programmers can forget to disclose relevant information, they can inadvertently disclose erroneous information, or they can misuse standard types, thus introducing errors into later inference over the provenance. We will need to gain more experience with a variety of users to assess the significance of these possibilities.

3 Accessing Provenance

The CPL includes an API for programmatically accessing the collected provenance. It includes functions such as `cpl_get_object_ancestry` to list ancestors and descendants of an object or `cpl_get_object_info` to retrieve information about an object, such as its namespace, name, type, creation time, current version, and provenance of provenance. Appendix A contains more details.

The CPL distribution also includes a command-line tool for issuing simple provenance queries on files, and it is also integrated with Orbiter [7] for visualization.

4 Implementation

We implemented the library in C++ and currently support C, Java, and Perl bindings, plus a command-line tool for instrumenting shell scripts and for manually disclosing provenance. Our own use of this tool has proven indispensable, as it enables us to disclose the provenance of simple operations that we perform from the command line, such as copying files. The CPL works on Linux, OS X, and with some limitations on Windows.

Each application that uses the CPL loads it as a shared library and makes a call to `cpl_attach` as described above, which in turn opens its database connection. There is thus one running instance of the CPL embedded in each provenance-enabled process, instead of each application communicating with a single service. The individual instances of the CPL rely on the database for caching and on shared semaphores to perform synchronization in a few rare instances when it is necessary.

The advantage of this approach is simplicity from the user's point of view, since there is no need to worry about running a separate service. The disadvantage is that this puts more stress on the database, because it limits the kinds of information the library can cache. On the other hand, our experience shows that database caches are effective, and we save on the IPC overhead that we would have if the CPL were running as a separate process.

The CPL is designed to work with multiple database engines – both relational and graph databases. We implemented ODBC and SPARQL/RDF [6] drivers, which we tested on MySQL, PostgreSQL, and 4store [1]. The library communicates with the drivers using a high-level interface, in which the individual API functions are designed to be sufficiently expressive so that the database driver can implement them efficiently, leveraging the specific features of the database. At the same time, the actions are designed to be simple enough that they can be trivially decomposed into smaller actions if necessary. Almost all tasks in practice can be performed using a single SQL or SPARQL query. The CPL requires that each database operation is atomic and durable and preserves database consistency. The CPL does not require the atomic guarantee across multiple operations. The documentation accompanying the source code contains more details (see Appendix B).

5 Use Cases

We have used the CPL in two projects: GraphDB Bench (a part of the Tinkubator [11] project) and Kepler [2].

5.1 GraphDB Bench

GraphDB Bench [11] is a tool for benchmarking graph databases – both to compare different databases to each other and to better understand performance of a single database system. The benchmark is organized as a series of operators (such as ingesting a GraphML file, adding a vertex, getting the *k*-hop neighborhood of a node) acting on an instance of a graph database. The program writes result, the elapsed time, and the memory usage of each completed operation to a .csv file.

We instrumented the benchmark to disclose data flow from the database to every operator that accesses it and from each operator to the .csv file with the results. The benchmark also discloses data flow from operators that modify the database (such as ingest) to the instance of the database. We annotate every operator with its parameters, such as the value of *k* in *k*-hop microbechmarks or the values of model parameters in random graph generators. The instrumentation involved adding or modifying approximately 270 lines of the source code (excluding comments), most of which was copy and paste.

The result is that we easily track how each .csv was produced – such as what operators were used, which instance of the database they operated on, and what parameters we used to configure the operators and the benchmark tool. We can also learn additional information about the benchmark by examining provenance of provenance, which includes the command-line arguments passed in to the Java Virtual Machine running the benchmark, such as the maximum size of the Java heap.

The collected provenance also enables us to query the ancestors of a database in order to learn which operators modified it and from which GraphML file the database was originally constructed. This is useful, so that for example, if we find out that the database was not modified after the initial ingest, we can reuse it safely for subsequent benchmarks. Otherwise we might need to restore it to the original state.

5.2 Kepler

Kepler [2] is a workflow engine that automatically collects provenance into one of several storage backends, such as a SQL database, an OPM file, or a text file.

It took us less than an hour to implement basic CPL support. We based our implementation on the source code of Kepler's OPM module, and the work was relatively straightforward. The module discloses data dependencies between actors and artifacts just like OPM module, except it stores the actor parameters as properties instead of as artifacts with data dependency relationships with their respective actors. This significantly de-clutters the provenance graph and enables our visualization tool to display the actor properties in a simple, concise table.

We further extended our CPL module to integrate its provenance with the file system by disclosing data flow from an input file to a file-reading actor just before it is executed, and from a file-writing actor to its output file after it is executed. This required on average 1-2 lines of actor-specific code in the provenance module per each actor that does file I/O to retrieve the file name from its parameters. This enabled us to easily track provenance across multiple workflows and data processing stages outside of the workflow engine.

6 Related Work

PASS [9] enables applications to disclose their provenance using the Disclosed Provenance API (DPAPI) and to integrate it with the provenance that it automatically infers from observing system calls. An advantage of the DPAPI is its deep integration with the operating system, which enables the application programmers to bundle provenance information with the `write` system call (called `pawrite` in PASS), so that the provenance is disclosed if and only if the write succeeds. The primary advantages of the CPL over the DPAPI are its portability, ease of use, and the ability to query provenance. The CPL does not require a modified operating system to run on, and it allows the applications to look up the objects they previously created based on their namespace, name, and type, without requiring them to remember their IDs and version numbers. The CPL further solves the problems outlined previously in Section 2.1.

In terms of the scope and the purpose of the project, the CPL is perhaps closest to VisTrails SDK [12], which allows VisTrails's OEM partners to embed the VisTrails Provenance Engine within their applications, enabling provenance capture without the need of substantial changes to their products. The SDK is currently under development and thus not publicly available, so we are unfortunately unable to compare it to the CPL.

7 Conclusion

We designed the CPL as a general-purpose provenance library and demonstrated its use in applications and data analysis pipelines. The CPL provides an easy, viable solution for collecting provenance in heterogeneous environments. It solves several problems that plague existing provenance systems, integrates provenance across multiple programs and environments, and provides API to access the provenance and facilitate its integration with provenance analysis tools.

References

[1] 4STORE.ORG. 4store - scalable RDF storage, 2012. `http://4store.org/`.

[2] ALTINTAS, I., BARNEY, O., AND JAEGER-FRANK, E. Provenance collection support in the Kepler scientific workflow system. *IPAW* (2006).

[3] ANGELINO, E., BRAUN, U., HOLLAND, D. A., MACKO, P., MARGO, D., AND SELTZER, M. Provenance integration requires reconciliation. In *TaPP* (2011).

[4] ANGELINO, E., YAMINS, D., AND SELTZER, M. I. StarFlow: A script-centric data analysis environment. In *IPAW* (2010), vol. 6378 of *LNCS*, Springer, pp. 236–250.

[5] CALLAHAN, S. P., FREIRE, J., SANTOS, E., SCHEIDEGGER, C. E., SILVA, C. T., AND VO, H. T. VisTrails: visualization meets data management. In *SIGMOD* (2006), ACM, pp. 745–747.

[6] CLARK, K. G., FEIGENBAUM, L., AND TORRES, E. SPARQL Protocol for RDF, 2008. Published online on January 15th, 2008 at `http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/`.

[7] MACKO, P., AND SELTZER, M. Provenance map orbiter: Interactive exploration of large provenance graphs. In *TaPP* (2011).

[8] MOREAU, L., MISSIER, P., BELHAJJAME, K., CRESSWELL, S., GIL, Y., GOLDEN, R., GROTH, P., KLYNE, G., MCCUSKER, J., MILES, S., MYERS, J., AND SAHOO, S. The PROV data model and abstract syntax notation. W3C working draft, W3C, Feb. 2012. http://www.w3.org/TR/2012/WD-prov-dm-20120202/.

[9] MUNISWAMY-REDDY, K.-K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference* (June 2009).

[10] SPILLANE, R. P., SEARS, R., YALAMANCHILI, C., GAIKWAD, S., CHINNI, M., AND ZADOK, E. Story book: An efficient extensible provenance framework. In *TaPP* (2009).

[11] TINKERPOP. Tinkubator wiki, 2012. `https://github.com/tinkerpop/tinkubator/wiki`.

[12] VisTrails SDK, 2012. `http://www.vistrails.com/sdk.html`.

A API Overview

We present a brief overview of selected API functions from CPL's C bindings; Perl and Java bindings are omitted for brevity. Perl bindings are similar to C bindings. We designed the Java API from scratch to make it very Java-like.

A more complete documentation for C bindings is included in the design documentation and Doxygen comments (see Appendix B for a link to obtain a copy of the library). The Perl and Java bindings are documented in a `man` page and Javadoc comments, respectively.

The CPL functions fall into the following five categories:

- **Attach/Detach**: Functions to attach/detach the CPL from the database backend.
- **Object Creation and Lookup API**: Functions to lookup and create provenance objects.
- **Disclosed Provenance API**: Functions used to disclose provenance.
- **Provenance Access API**: Functions used to access and query the recorded provenance.
- **Helper Functions**: Miscellaneous functions.

In C, all functions return an error or a success code, which can be tested using the macro `CPL_IS_OK`; actual return values are returned via pointer arguments. Functions in languages that support exceptions, such as Java and Perl, return the return values directly and throw exceptions on errors.

A.1 Attach/Detach Functions

`cpl_attach(backend)`: Initialize the CPL bindings for the current process and attach to the specified database backend, which is an object created using either:

- `cpl_create_odbc_backend(connection_str, type, *out)`: Connect to a relational database using ODBC.
- `cpl_create_rdf_backend(url_query, url_update, type, *out)`: Connect to a triplestore or a graph database using SPARQL/RDF.

`cpl_detach()`: Detach from the database backend.

A.2 Object Creation and Lookup API

`cpl_create_object(namespace, name, type, container, *out)`: Create a new object with the given name and type in the given namespace. The `container` argument is currently unused; it is reserved for future use when we implement provenance containment.

`cpl_lookup_object(namespace, name, type, *out)`: Lookup and return a previously created object; return the latest object if there are multiple objects with the matching namespace, name, and type.

`cpl_lookup_object_ext(namespace, name, type, flags, iterator, context)`: Lookup and return all objects with the matching namespace, name, and type; pass information about each object plus the caller-defined `context` to the specified `iterator` callback function. The C++ CPL extensions provide several iterators for common operations, such as collecting the returned objects in a `std::list` or `std::vector`.

`cpl_create_or_lookup_object(namespace, name, type, container, *out)`: Lookup and return an existing object with the matching namespace, name, and type. Create it if it does not already exist.

`cpl_lookup_by_property(key, value, iterator, context)`: Retrieve all objects with the matching key/value pairs and pass each to the specified `iterator` function alongside with the caller-provided `context`. The C++ CPL extensions provide several common callback functions.

`cpl_lookup_file(name, flags, *out, *out_version)`: Get a provenance object and its version that corresponds to the specified file on the local file system. Create the provenance object if necessary.

A.3 Disclosed Provenance API

`cpl_data_flow(data_dest, data_source, type)`
`cpl_data_flow_ext(data_dest, data_source, data_source_ver, type)`: Disclose a data flow from `data_source` to `data_dest` of the given CPL-defined type (use `CPL_DATA_GENERIC` as default; an example of another type is `CPL_DATA_COPY` to be used when making an exact copy of the given object).

`cpl_control(object_id, controller, type)`
`cpl_control_ext(object_id, controller, controller_ver, type)`: Disclose a control flow, in which `object_id` was controlled by `controller`, and the operation was of the given CPL-defined type (such as `CPL_CONTROL_GENERIC` or `CPL_CONTROL_START`).

`cpl_add_property(object_id, key, value)`: Associate an arbitrary key/value pair with the given object.

A.4 Provenance Access API

`cpl_get_version(object_id, *out)`: Get the current version of the object.

`cpl_get_current_session(*out)`: Return the ID of the current session – the aforementioned *provenance of provenance* tag associated with the current process.

`cpl_get_session_info(session_id, *out)`: Return information about the given session, such as the user name and the command line of the process. The caller is responsible to destroy the returned `struct` by calling `cpl_free_session_info(info_struct)`.

`cpl_get_object_info(object_id, *out)`: Return information about the specified provenance object, such as its namespace, name, type, version, creation time, and the session that created it (i.e. the provenance of provenance of the object). The caller is responsible to destroy the returned `struct` by calling `cpl_free_object_info(info_struct)`.

`cpl_get_version_info(object_id, version, *out)`: Get information about the specific version of the provenance object, such as its creation time and the session that created it (i.e. the provenance of provenance). The caller is responsible to destroy the returned `struct` by calling `cpl_free_version_info(info_struct)`.

`cpl_get_object_ancestry(object_id, version, direction, flags, iterator, context)`: Get all ancestors or descendants (depending on the value of the `direction` argument) of the given object; pass each encountered provenance edge to the specified `iterator` function alongside with the caller-provided `context`. The C++ CPL extensions provide several common callback functions.

`cpl_get_properties(object_id, version, key, iterator, context)`: Retrieve all properties of the given object that match the specified `key`, or all properties if the `key` is `NULL`, and pass each to the specified `iterator` function alongside with the caller-provided `context`. The C++ CPL extensions provide several common callback functions.

A.5 Helper Functions

Miscellaneous functions for manipulating object IDs, such as `cpl_id_copy()` and `cpl_id_cmp()` in C or the corresponding overloaded operators in C++, and `cpl_error_string()` to translate a return code to a message string.

B Availability

A copy of the library, including additional documentation and API specification, can be obtained from:

`code.google.com/p/core-provenance-library`