ANL/MCS-TM-311

Provenance Management in Swift with Implementation Details

Mathematics and Computer Science Division
Technical Memorandum ANL/MCS-TM-311

ANL/MCS-TM-311

Provenance Management in Swift with Implementation Details

by
L.M.R. Gadelha Jr.,[1,2] B. Clifford,[3] M. Mattoso,[1] M. Wilde,[4,5] and I. Foster[4,5]
[1]Computer and Systems Engineering Program, Federal University of Rio de Janeiro, Brazil
[2]National Laboratory for Scientific Computing, Brazil
[3]Department of Astronomy and Astrophysics, University of Chicago, USA
[4]Computation Institute, University of Chicago, USA
[5]Mathematics and Computer Science Division, Argonne National Laboratory, USA

May 2010

This work was supported in part by CAPES Grant 5205-09-3, CNPq,
and the Office of Advanced Scientific Computing Research, Office
of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Contents

Provenance Management in Swift with Implementation Details

Luiz M. R. Gadelha Jr.[a,b], Ben Clifford, Marta Mattoso[a], Michael Wilde[c,d], Ian Foster[c,d]

[a]*Computer and Systems Engineering Program, Federal University of Rio de Janeiro, Brazil*
[b]*National Laboratory for Scientific Computing, Brazil*
[c]*Computation Institute, University of Chicago, USA*
[d]*Mathematics and Computer Science Division, Argonne National Laboratory, USA*

Abstract

The Swift parallel scripting language allows for the specification, execution and analysis of large-scale computations in parallel and distributed environments. It incorporates a data model for recording and querying provenance information. In this article we describe these capabilities and evaluate interoperability with other systems through the use of the Open Provenance Model. We describe Swift's provenance data model and compare it to the Open Provenance Model. We also describe and evaluate activities performed within the Third Provenance Challenge, which consisted of implementing a specific scientific workflow, capturing and recording provenance information of its execution, performing provenance queries, and exchanging provenance information with other systems. Finally, we propose improvements to both the Open Provenance Model and Swift's provenance system.

Key words: provenance, parallel scripting languages, scientific workflows

1. Introduction

The automation of large scale computational scientific experiments can be accomplished through the use of workflow management systems [9], parallel scripting tools [23], and related systems that allow the definition of the activities, input and output data, and data dependencies of such experiments. The manual analysis of the data resulting from their execution is usually not feasible, due to the large amount of information commonly generated by these experiments. Provenance systems can be used to facilitate this task, since they gather details about the design [14] and execution of these experiments, such as data artifacts consumed and produced by their activities. They also make it easier to reproduce an experiment for the purpose of verification.

The Open Provenance Model (OPM) [18] is an ongoing effort to standardize the representation of provenance information. It defines the entities *artifact*, *process*, and *agent* and the relationships *used* (between an artifact and a process), *wasGeneratedBy* (between a process and an artifact), *wasControlledBy* (between an agent and a process), *wasTriggeredBy* (between two processes), and *wasDerivedFrom* (between two artifacts). These relationships are used to assert causal dependencies between the entities defined in the model. A set of these assertions can be used to build a *provenance graph*. One of the main objectives of OPM is to allow the exchange of provenance information between systems. It also describes valid inferences that can be made from provenance graphs. More complex relationships between processes and artifacts can be derived using, for instance, transitivity.

The Swift parallel scripting system [25] [23] is a successor of the Virtual Data System (VDS) [13] [26] [6]. It allows the specification, management and execution of large-scale scientific workflows on parallel and distributed environments. The SwiftScript language is used for high-level specification of computations, it

Email addresses: `lgadelha@lncc.br` (Luiz M. R. Gadelha Jr.), `benc@hawaga.org.uk` (Ben Clifford), `marta@cos.ufrj.br` (Marta Mattoso), `wilde@mcs.anl.gov` (Michael Wilde), `foster@mcs.anl.gov` (Ian Foster)

Table 1: Database relation `processes`.

Attribute	Definition
`id`	the URI identifying the process
`type`	the type of the process: execution, compound procedure, function, operator

has features such as data types, data mappers, dataset iteration, conditional branching, and procedural composition. It allows the manipulation of datasets in terms of their logical organization. The XML Dataset Typing and Mapping (XDTM) [19] notation is used to define *mappers* between this logical organization and the actual physical structure of the dataset. Procedures perform logical operations on input data, without modifying them. SwiftScript also allows procedures to be composed to define more complex computations. By analyzing the inputs and outputs of these procedures, the system determines data dependencies between them. This information is used to execute procedures that have no mutual data dependencies in parallel. Swift supports common execution managers for clustered systems and grid environments, such as Condor [11], GRAM [7], and PBS [16]. It also supports Falkon [21], an execution engine that provides high job execution throughput; and SSH [24], for executing jobs via secure remote logins. Swift logs a variety of information about each computation. This information can be exported using tools included in Swift to a relational database that uses a data model similar to OPM. Our provenance approach focuses on gathering information about the relationship between data and processes at the SwiftScript level. We do not gather information about lower level processes involved in executing a parallel script with Swift, such as moving data to computational resources, and submitting tasks to execution managers, although this is logged and could be integrated.

The objective of this paper is to present and evaluate the local and remote provenance recording and analysis capabilities of Swift, and compare them with those of other provenance systems. In the sections that follow, we describe the provenance capabilities of the Swift system and evaluate its interoperability with other systems through the use of OPM. We describe the provenance data model of the Swift system and compare it to OPM. We also describe and evaluate activities performed within the Third Provenance Challenge (PC3) which consisted of implementing and executing a scientific workflow (Pan-STARRS' [12] LoadWorkflow), gathering and recording provenance information of its execution, performing provenance queries, and exchanging provenance information with other systems.

2. Data Model

In Swift, data is represented by strongly-typed single-assignment variables. Data types can be *atomic* or *composite*. Atomic types are given by *primitive* types, such as integers or strings, or *mapped* types. Mapped types are used for representing and accessing data stored in local or remote files. *Composite* types are given by structures and arrays. In the Swift runtime, data is represented by a *dataset handle*. It may have as attributes a value, a filename, a child dataset handle (when it is a structure or an array), or a parent dataset handle (when it is contained in a structure or an array).

Swift processes are given by invocations of external programs, invocations of internal procedures, built-in functions, and operators. Dataset handles are produced and consumed by Swift processes.

In the Swift provenance model, dataset handles and processes are recorded, as are the relations between them (either a process consuming a dataset handle as input, or a process producing a dataset handle as output). Each dataset handle and process is uniquely identified in time and space by a URI. This information is stored persistently in a relational database. The two key relational tables used to store the structure of the provenance graph are `processes`, which stores brief information about processes (see table 1), and `dataset_usage`, which stores produced and consumed relationships between processes and dataset handles (see table 2). Other tables (see figure 1) are used to record details about each process and dataset, and other relationships such as dataset containment.

Table 2: Database relation `dataset_usage`.

Attribute	**Definition**
`process_id`	a URI identifying the process end of the relationship
`dataset_id`	a URI identifying the dataset handle end of the relationship
`direction`	whether the process is consuming or producing the dataset handle
`param_name`	the parameter name of this relation

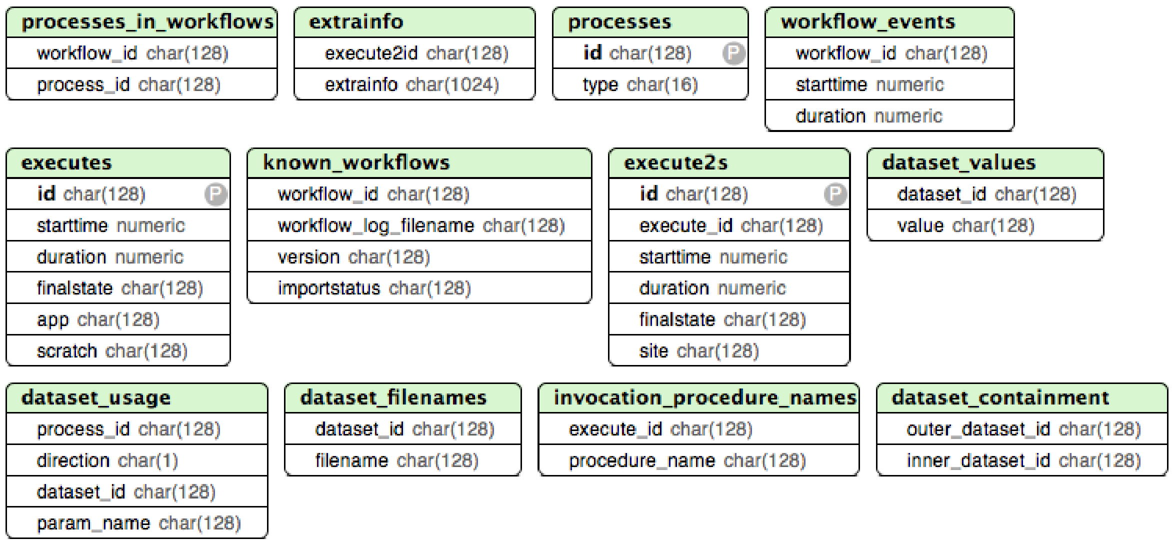


Figure 1: Swift's provenance database relations.

Listing 1: SwiftScript program for sorting a file.

```
type file;
app (file o) sortProg(file i) {
  sort stdin=@filename(i) stdout=@filename(o);
}
file f <"inputfile">;
file g <"outputfile">;
g = sortProg(f);
```

Consider the SwiftScript program in listing 1, which first describes a procedure (`sortProg`, which calls the external executable `sort`); then declares references to two files, (`f`, a reference to `inputfile`, and `g`, a reference to `outputfile`); and finally calls the procedure `sortProg`. When this program is run, provenance records are generated as follows:

- a process record is generated for the initial call to the `sortProg(f)` procedure;
- a process record is generated for the `@filename(i)` function invocation inside `sortProg`, representing the evaluation of the `@filename` function that Swift uses to determine the physical filename corresponding to the reference `f`;
- and a process record is generated for the `@filename(o)` function invocation inside `sortProg`, again representing the evaluation of the `@filename` function, this time for the reference `g`.

Dataset handles are recorded for:

- the string `"inputfile"`;
- the string `"outputfile"`;
- the file variable `f`;
- the file variable `g`;
- the filename of `i`;
- and the filename of `o`.

Input and output relations are recorded as:

- the `sortProg(f)` procedure takes `f` as an input;
- the `sortProg(f)` procedure produces `g` as an output;
- the `@filename(i)` function takes `f` as an input;
- the `@filename(i)` function produces the filename of `f` as an output;
- the `@filename(o)` function takes `g` as an input;
- and the `@filename(o)` function produces the filename of `g` as an output.

The Swift provenance model is close to OPM, but there are some differences. Dataset handles correspond closely with OPM artifacts as immutable representations of data. However they do not correspond exactly, dataset handles do not record provenance due to aliasing, such as when accessing arrays. Section 4 discusses this issue in more detail. The OPM entity “agent” is currently not represented in Swift's provenance model, however this could be represented, for instance, by the identity of the user that runs a workflow.

Except for *wasControlledBy*, the dependency relationships defined in OPM can be derived from the `dataset_usage` database relation. It explicitly stores the *used* and *wasGeneratedBy* relationships. Table 3

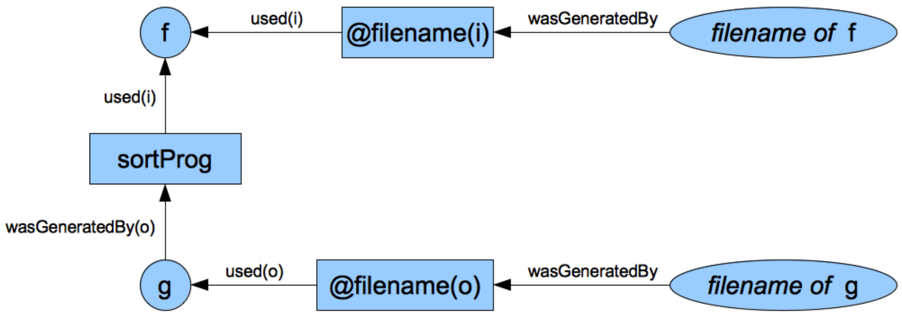
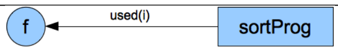
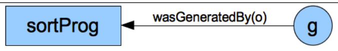
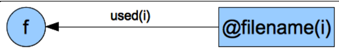
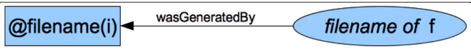
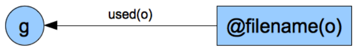
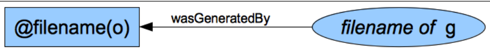


Figure 2: Provenance relationships of a `sortProg` execution.

Table 3: Equivalence between tuples in the `dataset_usage` table and OPM relationships.

Tuple in the `dataset_usage` table				OPM equivalent
`process_id`	`dataset_id`	`direction`	`param_name`	
`sortProg(f)`	`f`	In	`i`	
`sortProg(f)`	`g`	Out	`o`	
`@filename(i)`	`f`	In	`i`	
`@filename(i)`	filename of `f`	Out	`result`	
`@filename(o)`	`g`	In	`o`	
`@filename(i)`	filename of `g`	Out	`result`	

shows the equivalence between tuples stored in the `dataset_usage` table and OPM relationships. *wasTriggeredBy* and *wasDerivedFrom* dependency relationships can also be inferred from `database_usage`, in the `sortProg` example we have, for instance, $\mathtt{f} \xleftarrow{\text{wasDerivedFrom}} g$. Figure 2 shows the provenance relationships captured by Swift's provenance system for the `sortProg` example using OPM notation.

One of the main concerns with using a relational model for representing provenance is the need for querying over the transitive relation expressed in the `dataset_usage` table. For example, after executing the SwiftScript code in listing 2, it might be desirable to find all dataset handles that lead to `c`: that is, `a` and `b`. However simple SQL queries over the `dataset_usage` relation can only go back one step, leading to the answer `b` but not to the answer `a`. To address this problem, we generate a transitive closure table by an incremental evaluation system [10]. This approach makes it straightforward to query over transitive relations using natural SQL syntax, at the expense of larger database size and longer import time.

Swift's provenance data model is not dependent on a particular database model. A number of other forms were briefly experimented with during development [5]. The two most developed and interesting models were XML and Prolog. XML provides a semi-structured tree form for data. A benefit of this approach is that new data can be added to the database without needing an explicit schema to be known to the database. In addition, when used with a query language such as XPath, certain transitive queries become straightforward with the use of the `//` operator of XPath. Representing the data as Prolog tuples is a different representation than a traditional database, but provides a query interface that can express interesting queries flexibly.

Listing 2: Transitivity of provenance relationships.

```
b = p(a);
c = q(b);
```

3. PC3: Implementation and Queries

One of the main goals of PC3 was to evaluate OPM as a mechanism for interoperability between provenance systems. An astronomy workflow from the Pan-STARRS [12] project, called LoadWorkflow, was used for this purpose. It receives a set of CSV files containing astronomical data, stores the contents of these files in a relational database, and performs a series of validation steps. This workflow makes extensive use of conditional and loop flow controls and database operations. Database operations are somewhat outside the scope of usual Swift applications, which are generally file-oriented. A Java implementation of the component applications of LoadWorkflow was provided in the Provenance Challenge Wiki [2]. These components are declared in the SwiftScript implementation of the workflow as external application procedures. The procedural body of the SwiftScript code closely follows the LoadWorkflow specification since Swift has native support for decision and loop controls, given by the `if` and `foreach` constructs.

Initially, the mapped types used in the workflow are declared. Mapped types refer to data objects that do not reside in the main memory, which is the case for data files. Most of the inputs and outputs of the Java implementation of the workflow activities are files in XML format, represented in our Swift implementation as xmlfile. In order to manipulate the input and output values we had to convert some of these files into plain text files, represented as textfile, and then read them using the readData SwiftScript function.

```
type xmlfile;
type textfile;
```

The following part consists of using app declarations to define the workflow's component applications. This allows the invocation of executable applications of the Java implementation of the workflow from Swift. They define the applications' inputs and outputs, which are XML files in the LoadWorkflow case, and provide a reference that will allow Swift to find the actual application executable by looking at its application catalog. These app declarations are given by `ps_load_executable`, `ps_load_executable_threaded`, `ps_load_executable_db`, and `compact_database`. `ps_load_executable_db` and `compact_database` also have a reference to the LoadWorkflow database as input, which is given also by an XML file. The subsequent declarations are used to manipulate an XML file in order to extract boolean values, count entries, and extract entries. Finally, the stop app declaration simply refer to a shell script that returns an error code and is used to halt the workflow execution.

```
(xmlfile output) ps_load_executable(xmlfile input, string s) {
  app {
    ps_load_executable_app @input s @output;
  }
}

(xmlfile output) ps_load_executable_threaded(xmlfile input, string s, external thread) {
  app {
    ps_load_executable_app @input s @output;
  }
}

(xmlfile output) ps_load_executable_db (xmlfile db, xmlfile input, string s, external thread) {
  app {
    ps_load_executable_db_app @db @input s @output;
  }
}

compact_database (xmlfile db, external thread) {
  app {
    compact_database_app @db;
  }
}
```

Table 4: LoadWorkflow Activities - Pre-Load Section

Activity	**Input**	**Output**
IsCSVReadyFileExists: Verifies if the CSV root directory and the `csv_ready.csv` file exist.	string CSVRootPathInput, containing the path to the CSV root directory.	boolean IsCSVReadyFileExistsOutput, which is true if the verification succeeds, or false otherwise.
ReadCSVReadyFile: For each file listed in `csv_ready.csv`, it creates a CSVFileEntry, which consists of the path to the CSV file to be loaded, the path to the CSV header file containing the list of data columns, the number of rows in the file, the target database table, and the MD5 hash function value of the file. The columns names field is not populated by this activity.	string CSVRootPathInput, containing the path to the CSV root directory.	list ReadCSVReadyFileOutput of CSVFileEntry elements read from `csv_ready.csv`.
IsMatchCSVFileTable: Verifies if the tables to be loaded have corresponding data files.	list FileEntriesInput of CSVFileEntry elements read from `csv_ready.csv`.	boolean IsMatchCSVFileTablesOutput, which is true if the tables have corresponding CSV files, or false otherwise.
IsExistsCSVFileTable: Verifies if CSV data file and CSV header exist.	CSVFileEntry FileEntryInput.	boolean IsExistsCSVFileOutput, which is true if the CSV data file and CSV header files exist, or false otherwise.
ReadCSVFileColumnNames: Reads the list of column names present in the CSV data file from the CSV header file.	CSVFileEntry FileEntryInput.	CSVFileEntry FileEntryOutput, which results from updating the columns names field in the input using the values listed in the CSV header file.
IsMatchCSVFileColumnNames: Verifies if the columns expected for a target table are present in the CSV data file.	CSVFileEntry FileEntryInput, with the columns names field populated.	boolean IsMatchCSVFileColumnNamesOutput, which is true if column names listed in the CSV data files match the column names for the target database table, or false otherwise.

Table 5: LoadWorkflow Activities - Load Section

Activity	**Input**	**Output**
CreateEmptyLoadDB: Creates the database to which the CSV data files will be loaded. It returns a DatabaseEntry, which is a reference to the database containing its name and connection information.	string JobID, a unique job identifier for the batch of CSV data files.	A DatabaseEntry CreateEmptyLoadDBOutput.
LoadCSVFileIntoTable: Loads a CSV data file into the corresponding database table.	DatabaseEntry DBEntry, containing target table to load the CSV data file into. CSVFileEntry FileEntry, refering to the CSV data file to be loaded.	boolean LoadCSVFileIntoTableOutput, which is true is the load was successful, or false otherwise.
UpdateComputedColumns: Updates the computed columns in the table that was loaded. These columns are indicated by the value -999 in the CSV data file.	DatabaseEntry DBEntry, with the target table already loaded from the CSV data file. CSVFileEntry FileEntry, containing the name of target table in the database to update.	boolean UpdateComputedColumnsOutput, which is true if the columns were successfully updated, or false otherwise.

Table 6: LoadWorkflow Activities - Post-Load Section

Activity	**Input**	**Output**
IsMatchTableRowCount: Checks if number of rows loaded into table matches the expected.	DatabaseEntry DBEntry, where the target table is loaded and updated. CSVFileEntry FileEntry, containing the expected number of rows in the CSV data file and the target database table name.	bool IsMatchTableRowCountOutput, which is true if the number of rows in the target table matches the expected number of rows in the CSV data file.
IsMatchTableColumnRanges: Checks if the data loaded into database table columns is within the range of values expected.	DatabaseEntry DBEntry, where the target table is loaded and updated. CSVFileEntry FileEntry, containing the name of target table in the database to validate columns ranges.	bool IsMatchTableColumnRangesOutput, which is true if the data values of the columns in the target table are within the expected range, or false otherwise.
CompactDatabase: Compacts the database before concluding the workflow.	DatabaseEntry DBEntry, where all tables are loaded and validated.	None.

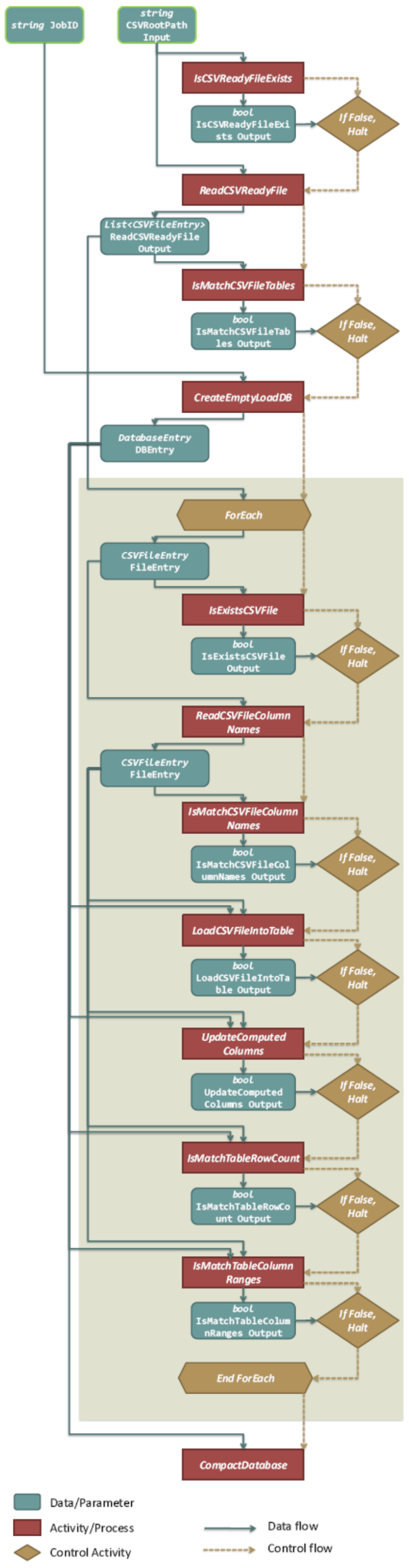


Figure 3: LoadWorkflow. Source: PC3 web site [1].

```
(textfile output) parse_xml_boolean_value(xmlfile input) {
  app {
    parse_xml_boolean_value_app @input @output;
  }
}

(textfile output) count_entries(xmlfile input) {
  app {
    count_entries_app @input @output;
  }
}

(xmlfile output) extract_entry(xmlfile input, int i) {
  app {
    extract_entry_app @input i @output;
  }
}

stop() {
  app {
    stop_app;
  }
}
```

The next part of the SwiftScript code is used for the declaration of compound procedures, which invoke other SwiftScript procedures instead of component programs. The `extract_boolean` procedure reads a text file and extracts the boolean value it contains. The checkvalid procedure simply tests a boolean value and halts the workflow if it is false. `ps_load_executable_boolean` and `ps_load_executable_db_boolean` are used to execute a workflow activity, they return a boolean value as output. The remaining procedures implement actual workflow activities by calling the other SwiftScript procedures defined.

```
(boolean output) extract_boolean(xmlfile input) {
  textfile text_out = parse_xml_boolean_value(input);
  output = readData(text_out);
}

(external out) checkvalid(boolean b) {
  if(!b) { stop(); }
}

(boolean output) ps_load_executable_boolean(xmlfile input, string s) {
  xmlfile xml_out = ps_load_executable(input, s);
  output = extract_boolean(xml_out);
}

(boolean output) ps_load_executable_db_boolean(xmlfile db, xmlfile input, string s, external thread) {
  xmlfile xml_out = ps_load_executable_db(db, input, s, thread);
  output = extract_boolean(xml_out);
}

(boolean output) is_csv_ready_file_exists(xmlfile input) {
  output = ps_load_executable_boolean(input, "IsCSVReadyFileExists");
}

(xmlfile output) read_csv_ready_file(xmlfile input) {
  output = ps_load_executable(input, "ReadCSVReadyFile");
}

(boolean output) is_match_csv_file_tables(xmlfile input) {
  output = ps_load_executable_boolean(input, "IsMatchCSVFileTables");
}

(xmlfile output, external outthread) create_empty_load_db(xmlfile input) {
  output = ps_load_executable(input, "CreateEmptyLoadDB");
}

(boolean output) is_exists_csv_file(xmlfile input) {
  output = ps_load_executable_boolean(input, "IsExistsCSVFile");
}

(xmlfile output) read_csv_file_column_names(xmlfile input, external thread) {
  output = ps_load_executable_threaded(input, "ReadCSVFileColumnNames", thread);
}
```

```
(boolean output) is_match_csv_file_column_names(xmlfile input) {
  output =  ps_load_executable_boolean(input, "IsMatchCSVFileColumnNames");
}

(boolean output) load_csv_file_into_table(xmlfile db, xmlfile input, external thread) {
  string dbcontent = readData(db);
  string inputcontent = readData(input);
  output = ps_load_executable_db_boolean(db, input, "LoadCSVFileIntoTable", thread);
}

(boolean output) update_computed_columns(xmlfile db, xmlfile input, external thread) {
  string dbcontent = readData(db);
  string inputcontent = readData(input);
  output = ps_load_executable_db_boolean(db, input, "UpdateComputedColumns", thread);
}

(boolean output) is_match_table_row_count(xmlfile db, xmlfile input, external thread) {
  string dbcontent = readData(db);
  string inputcontent = readData(input);
  output = ps_load_executable_db_boolean(db, input, "IsMatchTableRowCount", thread);
}

(boolean output) is_match_table_column_ranges(xmlfile db, xmlfile input, external thread) {
  string dbcontent = readData(db);
  string inputcontent = readData(input);
  output = ps_load_executable_db_boolean(db, input, "IsMatchTableColumnRanges", thread);
}
```

The subsequent piece of Swiftscript code is used for variable declarations. The workflow receives two files as input arguments, one containing the path to the CSV root directory and another one containing a job identifier. These values are received by the `csv_root_path_input_arg` and `job_id_arg` variables. The `csv_root_path_input` and `job_id` mapped type variables are declared and their values are declared to be contained in the files given as input arguments. The remaining variables declared in this piece of code are used to hold outputs of workflow procedures.

```
string csv_root_path_input_arg = @arg("csvpath");
string job_id_arg = @arg("jobid");
xmlfile csv_root_path_input <single_file_mapper;file=csv_root_path_input_arg>;
xmlfile job_id <single_file_mapper;file=job_id_arg>;
boolean  is_csv_ready_file_exists_output;
xmlfile  read_csv_ready_file_output;
boolean is_match_csv_file_tables_output;
xmlfile create_empty_load_db_output;
textfile count_entries_output;
int entries;
xmlfile split_list_output[];
```

The final part of the SwiftScript code is the actual procedural portion of the LoadWorkflow implementation in Swift. It closely follows the LoadWorkflow logic since Swift has native support for decision and loop controls. The `split_list_output` array holds the CSV file entries that will be processed in the workflow, they are extracted from the XML file generated by the `read_csv_ready_file` procedure.

```
is_csv_ready_file_exists_output = is_csv_ready_file_exists(csv_root_path_input);
if(!is_csv_ready_file_exists_output) { stop(); }
read_csv_ready_file_output = read_csv_ready_file(csv_root_path_input);
is_match_csv_file_tables_output =  is_match_csv_file_tables(read_csv_ready_file_output);
if(is_match_csv_file_tables_output) {
  external db_over_time[];
  external dbinit; // some bug in analysis means can't use db_over_time for initial one
  (create_empty_load_db_output, dbinit) = create_empty_load_db(job_id);
  count_entries_output = count_entries(read_csv_ready_file_output);
  entries = readData(count_entries_output);
  int entries_seq[] = [1:entries];
  foreach i in entries_seq {
    split_list_output[i] = extract_entry(read_csv_ready_file_output, i);
  }
  foreach i in entries_seq {
    boolean is_exists_csv_file_output;
    xmlfile read_csv_file_column_names_output;
    boolean is_match_csv_file_column_names_output;
    boolean load_csv_file_into_table_output;
    boolean update_computed_columns_output;
    boolean is_match_table_row_count_output;
    boolean is_match_table_column_ranges_output;

    is_exists_csv_file_output = is_exists_csv_file(split_list_output[i]);
    external thread6 = checkvalid(is_exists_csv_file_output);
    read_csv_file_column_names_output = read_csv_file_column_names(split_list_output[i],
      thread6);
    is_match_csv_file_column_names_output =
      is_match_csv_file_column_names(read_csv_file_column_names_output);
    external thread2 = checkvalid(is_match_csv_file_column_names_output);

    if(i==1) { // first element...
     load_csv_file_into_table_output =  load_csv_file_into_table(create_empty_load_db_output,
       read_csv_file_column_names_output, dbinit);
    } else {
     load_csv_file_into_table_output =  load_csv_file_into_table(create_empty_load_db_output,
       read_csv_file_column_names_output, db_over_time[i]);
    }
    external thread3=checkvalid(load_csv_file_into_table_output);
    update_computed_columns_output =  update_computed_columns(create_empty_load_db_output,
      read_csv_file_column_names_output, thread3);
    external thread4 = checkvalid(update_computed_columns_output);
    is_match_table_row_count_output = is_match_table_row_count(create_empty_load_db_output,
      read_csv_file_column_names_output, thread4);
    external thread1 = checkvalid(is_match_table_row_count_output);
    is_match_table_column_ranges_output =
      is_match_table_column_ranges(create_empty_load_db_output,
      read_csv_file_column_names_output, thread1);
    db_over_time[i+1] = checkvalid(is_match_table_column_ranges_output);
  }
  compact_database(create_empty_load_db_output, db_over_time[entries+1]);
}
else {
  stop();
}
```

Core Query 1. The first query asks, for a given detection, which CSV files contributed to it. The strategy used to answer this query is to determine input CSV files that preceed, in the transitivity table, the process that inserted the detection. Suppose we want to determine the provenance of the detection that has the identifier 261887481030000003, the first query can be answered by first obtaining the Swift process identifier of the process that inserted the detection from the annotations included in the application database:

```
> select
    provenanceid
  from
    ipaw.p2detectionprov
  where
    detectid = 261887481030000003;

> tag:benc@ci.uchicago.edu,2008:swiftlogs:execute2:pc3-20090507-1008-q4dpcm28
  :ps_load_executable_db_app-b2bclgaj
```

The identifier returned is an `execute2` identifier, which means in this case that it refers to a successful execution attempt. In order to obtain the predecessors of this process in the transitivity table we need the actual execute identifier of the process, which can we can get with the following SQL query:

```
> select
    execute_id
  from
    execute2s
  where
    id = 'tag:benc@ci.uchicago.edu,2008:swiftlogs:execute2:pc3-20090507-1140-z7ebbrz0
         :ps_load_executable_db_app-8d52pgaj';

> tag:benc@ci.uchicago.edu,2008:swiftlogs:execute:pc3-20090507-1140-z7ebbrz0:0-5-5-1-5-1-2-0
```

Finally, we determine the filenames of datasets that contain CSV inputs in the set of predecessors of the process that inserted the detection:

```
> select
    filename
```

```
  from
    trans, dataset_filenames
  where
    after='tag:benc@ci.uchicago.edu,2008:swiftlogs:execute
           :pc3-20090507-1140-z7ebbrz0:0-5-5-1-5-1-2-0'
  and
    before=dataset_id and filename like '%split%';

> file://split_list_output-65fe229c-2da2-4054-997e-fb167b8c30ed--array/elt-3
  file://split_list_output-65fe229c-2da2-4054-997e-fb167b8c30ed--array/elt-2
  file://split_list_output-65fe229c-2da2-4054-997e-fb167b8c30ed--array/elt-1
```

These files contain the filenames of the CSV files that were given as input to the workflow, and that will result in the detection row insertion:

```
P2_J062941_B001_P2fits0_20081115_P2Detection.csv,
P2_J062941_B001_P2fits0_20081115_P2ImageMeta.csv,
P2_J062941_B001_P2fits0_20081115_P2FrameMeta.csv
```

Core Query 2. The second query asks if the range check (IsMatchColumnRanges) was performed in a particular table, given that a user found values that were not expected in it. This is implemented in the `q2.sh` script in the Swift SVN repository with the following SQL query:

```
> select
    dataset_values.value
  from
    processes, invocation_procedure_names, dataset_usage, dataset_values
  where
    type='compound' and
    procedure_name='is_match_table_column_ranges' and
    dataset_usage.direction='O' and
    dataset_usage.param_name='inputcontent' and
    processes.id = invocation_procedure_names.execute_id and
    dataset_usage.process_id = processes.id and
    dataset_usage.dataset_id = dataset_values.dataset_id;
```

This returns the input parameter XML for all IsMatchColumnRanges calls. These are XML values, and it is necessary to examine the resulting XML to determine if it was invoked for the specific table. There is unpleasant cross-format joining necessary here to get an actual yes/no result properly, although probably could use a `LIKE` clause to peek inside the value.

Core Query 3. The third core query asks which operation executions were strictly necessary for the Image table to contain a particular (non-computed) value. This uses the additional annotations made, that only store which process originally inserted a row, not which processes have modified a row. So to some extent, rows are regarded a bit like artifacts (though not first order artifacts in the provenance database); and we can only answer questions about the provenance of rows, not the individual fields within those rows. That is sufficient for this query, though. First find the row that contains the interesting value and extract its `IMAGEID`. Then find the process that created the `IMAGEID` by querying the Derby database table `P2IMAGEPROV`:

```
> select * from ipaw.p2imageprov where imageid=6294301;

  IMAGEID | PROVENANCEID
  ----------------------------------------------------------------------
  6294301 | tag:benc@ci.uchicago.edu,2008:swiftlogs:execute2:pc3-20090519
          | -2057d8dyi9o9:ps_load_executable_db_app-dpc8q1bj
```

Now we have a process ID for the process that created the row. Now query the transitive closure table for all predecessors for that process (as in the first core query). This will produce all processes and artifacts that preceeded this row creation. Our answer differs from the sample answer because we have sequenced access to the database, rather than regarding each row as a proper first-order artifact. The entire database state at a particular time is a successor to all previous database accessing operations, so any process which led to any database access before the row in question is regarded as a necessary operations. This is undesirable in some respects, but desirable in others. For example, a row insert only works because previous database operations which inserted other rows did not insert a conflicting primary key - so there is data dependency between the different operations even though they operate on different rows.

Optional Query 1. The workflow halts due to failing an IsMatchTableColumnRanges check. How many tables successfully loaded before the workflow halted due to a failed check? This counts how many load processes are known to the database (over all recorded workflows):

```
> select
    count(*)
  from
    invocation_procedure_names
  where
    procedure_name='load_csv_file_into_table';
```

This can be restricted to a particular workflow run like this:

```
> select
    count(process_id)
  from
    invocation_procedure_names, processes_in_workflows
  where
    procedure_name='load_csv_file_into_table'
  and
    workflow_id='tag:benc@ci.uchicago.edu,2008:swiftlogs:execute:pc3-20090519-1659-jqc5od2f
                 :run'
  and
    invocation_procedure_names.execute_id = processes_in_workflows.process_id;

> 3
```

Optional Query 2. Which pairs of procedures in the workflow could be swapped and the same result still be obtained (given the particular data input)? In our Swift representation of the workflow, we control dataflow dependencies. So many of the activities that could be commuted are in our implementation run in parallel. One significant thing one cannot describe in SwiftScript (and so cannot answer from the provenance database using this method) is commuting operations on the database. From a Swift perspective, this is a limitation of our SwiftScript language rather than in the provenance implementation. The query lists which pairs Unix process executions (of which there are 50x50) have no data dependencies on each other. There are 2082 rows. The base SQL query is this:

```
> select
    L.id, R.id
  from
    processes as L, processes as R
  where
    L.type='execute'
  and
    R.type='execute'
  and
    NOT EXISTS (select * from trans where before=L.id and after=R.id);
```

This answer is deficient in a few ways. We do not take into account non-execute procedures (such as compound procedures, function invocations, and operator executions) - there are 253 processes in total, 50 being executes and the remaineder being the other kinds of process. If we did that naively, we would not take into account compound procedures which contain other procedures (due to lack of decent support for nested processes - something like OPM accounts) and would come up with commutations which do not make sense.

In our initial attempts to implement LoadWorkflow, we found the use of the parallel `foreach` loop problematic because the database routines executed by the external application procedures are opaque to Swift. Due to dependencies between iterations of the loop, these routines were being incorrectly executed in parallel. It was necessary to serialize the loop execution to keep the database consistent. For the same reason, since most of the PC3 queries are for row-level database provenance, we had to implement a workaround for gathering this provenance by modifying the application database so that for every row inserted, an entry containing the execution identifier of the Swift process that performed this insertion is recorded on a separate annotation table.

The OPM output for a LoadWorkflow run in Swift was generated by a script that maps Swift's provenance data model to OPM's XML schema. Since OPM and Swift's provenance database use similar data models, Listing 3: Multiple provenance descriptions for a dataset.

```
int a = 7;
int b = 10;
int c[] = [a, b];
```

it is fairly straightforward to build a tool to import data from an OPM graph into the Swift provenance database. However we observed that the OPM outputs from the various participating teams, including Swift, carry many details of the LoadWorkflow implementation that are system specific, such as auxiliary tasks that are not necessarily related to the workflow. To answer the same queries, it would be necessary to perform some manual interpretation of the imported OPM graph in order to identify the relevant processes and artifacts.

4. PC3: Evaluation

PC3 provided an opportunity to use OPM in practice. This also enabled us to evaluate OPM and compare it to Swift's provenance data model. OPM originally did not specify a naming mechanism for globally identifying artifacts outside of an OPM graph. In Swift, dataset handles are given an URI, now OPM has an annotation for this purpose [18].

Swift's provenance implementation has two models of representing containment for dataset handles contained inside other dataset handles (arrays and complex types). A constructor/accessor model has special processes called accessors and constructors corresponding to the `[]` array accessor and `[1,2,3]` explicit construction syntax in SwiftScript. This model is proposed in OPM. In the Swift implementation, this is a cause of multiple provenances for dataset handles. For example, consider the SwiftScript program displayed in listing 3, the expression `c[0]` evaluates to the dataset handle corresponding to the variable `a`. That dataset handle has a provenance trace indicating it was assigned from the constant value `7`. However, that dataset handle has additional provenance indicating that it was output by applying the array access operator `[]` to the array `c` and the numerical value `0`. In OPM, the artifact resulting from evaluating `c[0]` is distinct from the artifact resulting from evaluating `a`, although they may be annotated with an *isIdenticalTo* arc [15]. In order to address the divergence between OPM and Swift's provenance data model, the dataset handle implementation could be modified so that it supported dataset handles being aliases to other dataset handles. The alias dataset handle would behave identically to the dataset handle that it aliases, except that it would have different provenance reflecting both the provenance of the original dataset handle, and subsequent operations made to retrieve it. In listing 3, then, `c[0]` would return a newly created dataset handle that aliased the original dataset handle for `a`. There is also a container/contained model, where relations are stored directly between dataset handles indicating that one is contained inside the other, without intervening processes. These relations can be inferred from the constructor/accessor model. The *Contained* relation between two artifacts, defined in [15], indicates that one is contained within another. This maps relatively cleanly to Swift's in-memory model of dataset handles containing other dataset handles. Swift collections may be hierarchical. In [15], it is not specified if the *Contained* relation holds only one level deep or to all elements contained in a collection.

The Swift team [3] made a proposal [2] for a minor change to the XML schema to better reflect the perceived intentions of the OPM authors. It was apparent that the present representation of hierarchical processes in OPM is insufficiently rich for some groups and that it would be useful to represent hierarchy of individual processes and their containing processes more directly. In Swift this is given by two categories: at the highest level, SwiftScript language constructs, such as procedures and functions; below that, the mechanics of Swift's execution, such as moving files to and from computational resources, and interactions with job execution. Swift provenance work to date has concentrated on the high-level representation, treating all of the low-level behavior as opaque and exposing neither processes nor artifacts. An OPM modification proposal for this is forthcoming. In Swift, this information is often available through the Karajan [17] execution engine thread identifier which closely maps to the Swift process execution hierarchy: a Swift process contains another Swift process if its Karajan thread identifier is a prefix of the second processes Karajan thread identifier. The Swift provenance database stores values of dataset handles when those values exist in-memory (for example, when a dataset handle represents and integer or a string). During PC3, interest in a standard way to represent this was expressed.

5. Related Work

As pointed out by some provenance surveys [22] [8], provenance systems are diverse regarding what the subject of the recorded provenance information is, what its level of granularity is, and how it is gathered, stored, and queried. This is also noticeable in PC3, where a variety of approaches were used to perform its activities. These provenance systems use a variety of data models, including semantic, relational and semistructured ones. Nevertheless, most of them were able to map their data models to OPM. In this section we build on these surveys and PC3 to compare Swift to other provenance systems.

Some provenance systems are not dependent on a particular workflow management system and work as provenance stores accessible, for example, as web services or through API calls. Karma [4], for instance, is implemented as a web service and stores provenance information in a relational database. Another example is Tupelo [20], which is a data and metadata management middleware that uses semantic web techniques, it has an API for enabling the storage of provenance information. Unlike Swift, systems of this type often require the instrumentation of the applications that compose a scientific workflow, which was also observed during PC3. This may prove difficult when the users of these applications are not also their developers, or if these applications are given by undocumented legacy code.

Another category is given by workflow systems that have integrated provenance management support. Vistrails [14], for instance, has a specialized provenance query language and uses both XML and relational databases to store provenance about data, processes and workflow evolution. This category includes Swift, which has a provenance system that is tightly coupled to it, that gathers information both about processes and data involved in the execution of a parallel script. It has comprehensive support for execution engines on high-performance parallel and distributed computing environments. Also, by having an integrated provenance system, Swift enables its users to readily generate and query provenance records of their experiments. In Swift, prospective provenance [26] is given by SwiftScript code, workflow evolution is not recorded. One alternative for recording this, not yet implemented, would be to couple Swift's provenance database with a source code version control system.

6. Concluding Remarks

Swift was able to perform the activities proposed for PC3. This success illustrates its capability to support provenance collection and analysis. Its provenance model is close to OPM, which enables interoperability with other provenance systems. One important aspect of Swift is its support for scalable execution of large-scale computations on parallel and distributed environments, along with the collection of provenance information. Several examples of parallel scripting applications are mentioned in [23], which include protein structure prediction, identification of drug targets using computational docking, and computational economics. In a recent example, Swift executed a neural imaging analysis workflow that involved 500,000 jobs. Swift development continues with the objective of improving its provenance capabilities and future work will concentrate on the following aspects:

Provenance system scalability. Swift provenance tracking model results in the generation and storage of large amounts of data. For smaller computations, such as LoadWorkflow, this is not a problem, but for larger computations (which is precisely the sort of computations for which Swift is particularly well suited) the provenance can become extremely large, and provenance queries can take a long time to execute. It may be desirable to provide options that can allow the programmer to request that Swift store less data albeit (presumably) with reduced accuracy. It may also be interesting to use distributed data management techniques to enable better scalability.

Provenance query system. It was clear from PC3 that although it is possible to express the provenance queries in SQL it is not always straightforward to do so, due to its poor transitivity support. One future objective is to make the provenance query system, which should include a specialized provenance query language, capable of being readily queried by scientists to let them do better science through validation, collaboration, and discovery.

Acknowledgments

This work was supported in part by CAPES Grant 5205-09-3, CNPq, and the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

A. Database Schema

This is the schema definition used for the main relational provenance implementation (in both SQLite3 and PostgreSQL):

```
-- executes_in_workflow is unused at the moment, but is intended to associate
-- each execute with its containing workflow
CREATE TABLE executes_in_workflows
    (workflow_id char(128),
     execute_id char(128)
    );

-- processes gives information about each process (in the OPM sense)
-- it is augmented by information in other tables
CREATE TABLE processes
    (id char(128) PRIMARY KEY, -- a uri
     type char(16) -- specifies the type of process. for any type, it
                   -- must be the case that the specific type table
                   -- has an entry for this process.
                   -- Having this type here seems poor normalisation, though?
    );



-- this gives information about each execute.
-- each execute is identified by a unique URI. other information from
-- swift logs is also stored here. an execute is an OPM process.
CREATE TABLE executes
    (id char(128) PRIMARY KEY, -- actually foreign key to processes
     starttime numeric,
     duration numeric,
     finalstate char(128),
     app char(128),
     scratch char(128)
    );

-- this gives information about each execute2, which is an attempt to
-- perform an execution. the execute2 id is tied to per-execution-attempt
-- information such as wrapper logs
CREATE TABLE execute2s
    (id char(128) PRIMARY KEY,
     execute_id, -- secondary key to executes and processes tables
     starttime numeric,
     duration numeric,
     finalstate char(128),
     site char(128)
    );

-- dataset_usage records usage relationships between processes and datasets;
-- in SwiftScript terms, the input and output parameters for each
-- application procedure invocation; in OPM terms, the artificts which are
-- input to and output from each process that is a Swift execution
CREATE TABLE dataset_usage
    (process_id char(128), -- foreign key but not enforced because maybe process
                           -- doesn't exist at time. same type as processes.id
     direction char(1), -- I or O for input or output
     dataset_id char(128), -- this will perhaps key against dataset table
     param_name char(128) -- the name of the parameter in this execute that
                          -- this dataset was bound to. sometimes this must
                          -- be contrived (for example, in positional varargs)
    );
```

```
-- invocation_procedure_name maps each execute ID to the name of its
-- SwiftScript procedure
CREATE TABLE invocation_procedure_names
    (execute_id char(128),
     procedure_name char(128)
    );

-- dataset_containment stores the containment hierarchy between
-- container datasets (arrays and structs) and their contents.
-- outer_dataset_id contains inner_dataset_id
CREATE TABLE dataset_containment
    ( outer_dataset_id char(128),
      inner_dataset_id char(128)
    );

-- dataset_filenames stores the filename mapped to each dataset. As some
-- datasets do not have filenames, it should not be expected that
-- every dataset will have a row in this table
CREATE TABLE dataset_filenames
    ( dataset_id char(128),
      filename char(128)
    );

-- dataset_values stores the value for each dataset which is known to have
-- a value (which is all assigned primitive types). No attempt is made here
-- to expose that value as an SQL type other than a string, and so (for
-- example) SQL numerical operations should not be expected to work, even
-- though the user knows that a particular dataset stores a numeric value.
CREATE TABLE dataset_values
    ( dataset_id char(128), -- should be primary key
      value char(128)
    );
```

```
-- known_workflows stores some information about each workflow log that has
-- been seen by the importer: the log filename, swift version and import
-- status.
CREATE TABLE known_workflows
    (
      workflow_id char(128),
      workflow_log_filename char(128),
      version char(128),
      importstatus char(128)
    );

-- workflow_events stores the start time and duration for each workflow
-- that has been successfully imported.
CREATE TABLE workflow_events
    ( workflow_id char(128),
      starttime numeric,
      duration numeric
    );

-- extrainfo stores lines generated by the SWIFT_EXTRA_INFO feature
CREATE TABLE extrainfo
    ( execute2id char(128),
      extrainfo char(1024)
    );
```

References

[1] Third Provenance Challenge. http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge.

[2] Provenance Challenge Wiki. http://twiki.ipaw.info, 2009.

[3] Swift Team Entry at the Third Provenance Challenge. http://twiki.ipaw.info/bin/view/Challenge/SwiftPc3, 2009.

[4] B. Cao, B. Plale, G. Subramanian, E. Robertson, and Y. Simmhan. Provenance Information Model of Karma Version 3. In *Proc. IEEE Congress on Services*, pages 348–351, 2009.

[5] B. Clifford. Provenance Working Notes. http://www.ci.uchicago.edu/~benc/provenance.html, 2009.

[6] B. Clifford, I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Tracking provenance in a virtual data grid. *Concurrency and Computation: Practice and Experience*, 20(5):565–575, 2008.

[7] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Job Scheduling Strategies for Parallel Processing - IPPS/SPDP '98 Workshop*, volume 1459 of *LNCS*, pages 62–82. Springer, 1998.

[8] S. da Cruz, M. Campos, and M. Mattoso. Towards a Taxonomy of Provenance in Scientific Workflow Management Systems. In *Proc. IEEE Congress on Services, Part I, (SERVICES I 2009)*, pages 259–266, 2009.

[9] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows in e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

[10] G. Dong, L. Libkin, J. Su, and L. Wong. Maintaining Transitive Closure of Graphs in SQL. *Intl. Journal of Information Technology*, 5, 1999.

[11] D. Epemaa, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12(1):53–65, 1996.

[12] N. Kaiser et al. Pan-STARRS: A Large Synoptic Survey Telescope Array. *Proc. SPIE*, 4836:154–164, 2002.

[13] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying and Automating Data Derivation. In *Proc. 14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*, pages 37–46, 2002.

[14] J. Freire, C. Silva, S. Callahan, E. Santos, C. Scheidegger, and H. Vo. Managing Rapidly-Evolving Scientific Workflows. In *International Provenance and Annotation Workshop (IPAW 2006)*, volume 4145 of *LNCS*, pages 10–18, 2006.

[15] P. Groth, S. Miles, P. Missier, and L. Moreau. A Proposal for Handling Collections in the Open Provenance Model. http://mailman.ecs.soton.ac.uk/pipermail/provenance-challenge-ipaw-info/2009-June/000120.html, 2009.

[16] R. Henderson. Job Scheduling Under the Portable Batch System. In *Job Scheduling Strategies for Parallel Processing - IPPS '95 Workshop*, volume 949 of *LNCS*, pages 279–294. Springer, 1995.

[17] G. Laszewski, M. Hategan, and D. Kodeboyina. Java CoG Kit Workflow. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 340–356. Springer, 2007.

[18] L. Moreau, B. Clifford, J. Freire, Y. Gil, P. Groth, J. Futrelle, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, Y. Simmhan, E. Stephan, and J. Van den Bussche. The Open Provenance Model - Core Specification (v1.1). *Future Generation Computer Systems*, doi:10.1016/j.future.2010.07.005, 2010.

[19] L. Moreau, Y. Zhao, I. Foster, J. Voeckler, and M. Wilde. XDTM: XML Dataset Typing and Mapping for Specifying Datasets. European Grid Conference (EGC 2005), 2005.

[20] J. Myers, J. Futrelle, J. Plutchak, P. Bajcsy, J. Kastner, L. Marini, R. Kooper, R. McGrath, T. McLaren, A. Rodríguez, and Y. Liu. Embedding Data within Knowledge Spaces. *CoRR*, abs/0902.0744, 2009.

[21] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: A Fast and Lightweight Task Execution Framework. In *Proc. ACM/IEEE Conference on High Performance Networking and Computing (Supercomputing 2007)*, 2007.

[22] Y. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in e-Science. *SIGMOD Record*, 34(3):31–36, 2005.

[23] M. Wilde, I. Foster, K. Iskra, P. Beckman, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel Scripting for Applications at the Petascale and Beyond. *IEEE Computer*, 42(11):50–60, November 2009.

[24] T. Ylönen. SSH - Secure Login Connections over the Internet. In *Proc. of the Sixth USENIX Security Symposium*, pages 37–42, 1996.

[25] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *Proc. 1st IEEE International Workshop on Scientific Workflows (SWF 2007)*, pages 199–206, 2007.

[26] Y. Zhao, M. Wilde, and I. Foster. Applying the Virtual Data Provenance Model. In *International Provenance and Annotation Workshop (IPAW 2006)*, volume 4145 of *LNCS*, pages 148–161. Springer, 2006.