Tracking Provenance in Swift

Ben Clifford[1] Luiz M. R. Gadelha Jr.[2] Marta Mattoso[2]

[1]Computation Institute
University of Chicago
Chicago, USA
benc@hawaga.org.uk

[2]Computer and Systems Engineering Program
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
{gadelha, marta}@cos.ufrj.br

Abstract

The objective of this technical report is to describe the provenance recording and analysis capabilities of Swift. Swift allows the specification, management, execution and analysis of large-scale scientific workflows on parallel and distributed environments. To demonstrate these capabilities, we show the results of the Swift Team participation at the Third Provenance Challenge (PC3) in this tutorial style report. PC3 organizers chose a workflow from the Pan-STARRS (The Panoramic Survey Telescope and Rapid Response System) project that should be implemented by the participating teams, and proposed a series of provenance activities. This reports presents the data model used to record and query provenance, the Pan-STARRS implementation in SwiftScript, and an SQL implementation of the PC3 queries. Finally, improvements to the Open Provenance Model and to Swift's provenance recording are proposed.

1 Introduction

The management of large scale computational scientific experiments is considerably facilitated by the use of scientific workflow management systems. They allow the definition of which processes compose an experiment as well as the data and control dependencies that exist between them. With the automation of the specification and execution of an experiment, little effort is required to re-execute it with different input parameters or slightly different tasks. The analysis of these experiments can be aided by provenance systems, which record the derivation history of each dataset manipulated by a workflow. The Open Provenance Model (OPM) [1] goal is to standardize the way in which provenance information is represented. It defines the entities *artifact*, *process*, and *agent* and the relations *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), *wasDerivedFrom* (between two artifacts). One of the goals of OPM is to allow the interoperability between different provenance systems.

The Swift scientific workflow management system [2] is a successor of the Virtual Data System (VDS) [3] and 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 workflows, it has features such as data types, data mappers, conditional and repetition flow-controls, and sub-workflow composition. Its data model and type system is derived from XDTM [4], which allows the abstract definition of data types and objects without refering to their physical representation. If some dataset does not reside in main memory, its materialization is done through the use of data mappers. Procedures perform logical operations on input data, without modifying them. Swiftscript also allows these procedures to be composed in order to form complex workflows. By analyzing the inputs and outputs of the procedures, the system determines data dependencies between procedures. This information is used to execute procedures that have no mutual data dependencies in parallel. Karajan [5] is used to submit the computational tasks of a workflow to various computational resources, it supports common execution managers such as PBS [6] and Globus GRAM [7]. The system can also use Falkon [8] as execution manager, which provides high job execution throughput. Swift logs a variety of information about each workflow execution. This information can be exported to a relational database that uses a data model that is very similar to OPM. Kickstart [9] can be used in order to gather provenance information about remote process executions.

The objective of this technical report is to describe the provenance recording and analysis capabilities of Swift. To demonstrate these capabilities, we show the results of the Swift Team participation at the Third Provenance Challenge (PC3) in detail. PC3 organizers chose a workflow from the Pan-STARRS (The Panoramic Survey Telescope and Rapid Response System) project that should be implemented by the participating teams, and proposed a series of provenance related tasks to be performed after the execution of the workflow.

The remaining sections of this report are organized as follows. The data model used to record and query provenance information in Swift is described in section 2. In section 3, we describe LoadWorkflow, the workflow selected for PC3 that is used in the Pan-STARRS project. In section 4, we describe the implementation of LoadWorkflow in SwiftScript. In section 5, we describe how we answered the queries proposed in PC3. Finally, in section 5, we describe other tasks performed as part of PC3 and make some concluding remarks.

2 Data Model

In the Swift runtime, data is represented by a DSHandle Java object: this includes data mapped to files, primitive types such as ints and strings, arrays and structures. DSHandles are produced and consumed by processes in Swift: invocations of external programs, and invocations of internal procedures, functions and operators. In the persistent Swift provenance model DSHandles and processes are recorded, as are the relations between them (either a process consuming a DSHandle as input, or a process producing a DSHandle as output). Each DSHandle and processes is uniquely identified in time and space by a URI. This provenance is persistently stored in an SQL database. In addition to SQL, other database layouts were experimented with, as detailed in appendix A. The two key tables of the database that store the structure of the provenance graph, defined in prov-init.sql, are `processes`, that stores brief information about processesdescribed in tables, and `dataset_usage`, that stores produced and consumed relationships between processes and DSHandles. Both are described in tables 2 and 2 respectively. Further tables exist to record details about each process, dataset and other relationships such as containment. Details of these can be found in `prov-init.sql`, in appendix B.

Consider this Swift program fragment:

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

When this program is run, then processes will be recorded for the evaluation of the following expressions:

(A) `s(f)` at the root level, representing the application execution;

(B) `@i` inside `s`, representing the evaluation of the `@filename` function;

(C) `@o` inside `s`, again representing the evaluation of the `@filename` function.

DSHandles will be recorded for:

(Q) the string `"inputfile"`;

(R) the string `"outputfile"`;

(S) the file variable `f`;

(T) the file variable `g`;

Row	Definition
`id`	the URI identifying the process
`type`	the type of the process (execution, compound procedure, function, operator)

Table 1: Database table `processes`.

Row	Definition
`process_id`	a URI identifying the process end of the relationship.
`dataset_id`	a URI identifying the DSHandle end of the relationship.
`direction`	whether the process is consuming or producing the DSHandle.
`param_name`	the parameter name of this relation.

Table 2: Database table `dataset_usage`.

(U) the filename of `i`;

(V) the filename of `o`.

Input/output relations will be recorded as:

- (A) takes (S) as an input;
- (A) produces (T) as an output;
- (B) takes (S) as an input;
- (C) takes (T) as an input;
- (B) produces (U) as an output.

One of the main concerns with using an SQL model for representing provenance is the need for querying over the transitive relation expressed in the `dataset_usage` table. For example, after executing the fragment:

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

It might be desirable to find all DSHandles that lead to `c` - that being `a` and `b`. However simple SQL queries over the used relation can only go back one step, leading to the answer `b` but not to the answer `a`. To address this problem, a transitive closure table is generated by an incremental evaluation system as described in [10]. This allows straightforward query over transitive relations using natural SQL syntax, at the expense of larger database size and longer import time.

2.1 Comparison of model to OPM

The Swift model is very close to OPM, but there are some differences.

2.1.1 DSHandles are almost OPM artifacts

DSHandles correspond closely with OPM artifacts as immutable representations of data. However they do not correspond exactly. An OPM artifact has unique provenance. However, a DSHandle can have multiple provenance descriptions. Consider this SwiftScript program:

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

Then consider the expression `c[0]`. This evaluates to a DSHandle, that is the DSHandle corresponding to the variable `a`. That DSHandle has a provenance trace indicating it was assigned from the constant value `7`. However, that DSHandle now 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 [11].

3 PC3 Workflow

Pan-STARRS is an astronomical survey of visible stars in the northern hemisphere, the Solar System, and potential collisions of astronomical objects with the Earth. The images are captured with a telescope operated by the University of Hawaii. Astronomical data is stored in an object data management framework, where scientists may query the information generated by the survey. The workflow proposed for PC3, LoadWorkflow, is an intermediate step between data acquisition with the telescope and its storage in the object data management framework. It receives as input a set of CSV files containing astronomical information, performs a series of validation steps, and stores the astronomical information in a relational database.

4 Workflow Implementation

In this section we describe how the PC3 workflow was implemented in Swift. A Java implementation of the workflow's activities was provided by Simmhan and is available at the PC3 web site [12]. Our implementation is available at the Swift team entry at the PC3 web site [13]. Each activity can be run using the Execute class in this implementation, which is called with the activity name and the activity inputs as parameters. We implemented a series of simple shell scripts to call the Execute class passing the appropriate parameters to it in order to execute the workflow activities from Swift. These scripts were listed in Swift's application catalog. Next we describe in detail the LoadWorkflow implementation in SwiftScript.

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 {
```

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 3: LoadWorkflow Activities - Pre-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 CreateEmpty-LoadDBOutput.
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 LoadCSVFileIntoTable-Output, 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 UpdateComputedColumn-sOutput, which is true if the columns were successfully updated, or false otherwise.

Table 4: LoadWorkflow Activities - 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 IsMatchTableRowCountOut-put, 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 IsMatchTableColumnRange-sOutput, 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.

Table 5: LoadWorkflow Activities - Post-Load Section

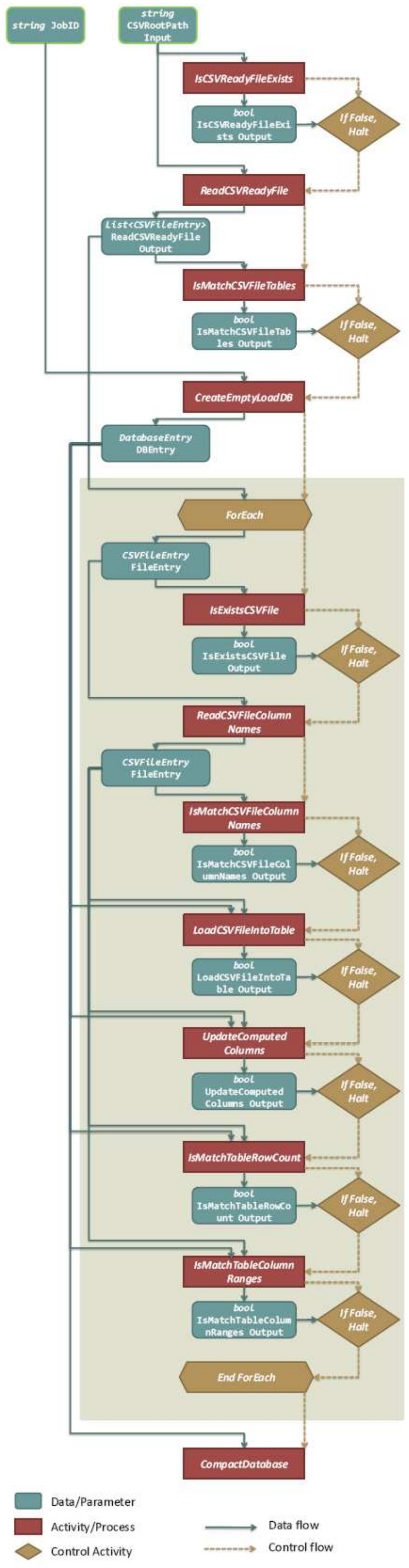

Figure 1: LoadWorkflow. Source: PC3 web site [12].

```
    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;
  }
}

(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();
}
```

4.1 Implementation Issues

In the first attempt to implement LoadWorkflow in Swift, the use of the foreach loop was problematic because the database routines are internal to the Java implementation and, therefore, Swift has no control over them. Since Swift tries to parallelize the `foreach` iterations it ended up incorrectly parallelizing the database operations of the workflow. It was necessary to move the `if()` and `stop()` statements into a separate procedure, called `checkvalid`, which output is of type external, making sure database accesses happen in sequence. That means also that there is a dataset object for each ”version” of the database, over time - previously there was no dataset representing the database.

5 PC3 Queries

Most of the PC3 queries are for row-level database provenance. As observed previously, the database operations are internal to the workflow activities and therefore Swift has no control over them. Also, Swift has no native support for making database connections. A workaround for this problem was implemented by modifying the application database so that for every row inserted or modified, an entry containing the execution identifier of the Swift process that performed the corresponding database operation is also inserted. In this section we show how we answered the queries proposed in the PC3 workshop, a task that was enabled by the workaround implemented. Details on creating the database and importing provenance data from Swift’s log files can found in [14].

5.1 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
```

5.2 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.

5.3 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.

5.4 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
```

5.5 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.

6 Additional PC3 Comments and Concluding Remarks

One of the main goals of PC3 was to evaluate OPM, therefore each team was asked to export its provenance data about LoadWorkflow in the OPM format, to import the OPM graph generated by the other teams, and to perform the proposed queries on the imported data. The OPM output for the LoadWorkflow run in Swift is available at the the web page. Few teams were able to import OPM graph from the other teams. Since OPM and the Swift provenance database use similar data models it is fairly straightforward to build a tool to import data from an OPM graph into the Swift provenace 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 specifically 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 (datasets).

In order to address the divergence between OPM and Swift provenance database data models the DSHandle implementation could be modified so that it supported DSHandles being aliases to other DSHandles, and so that any provenance creating aliasing behavior made such alias DSHandles instead of returning the original DSHandle. The alias DSHandle would behave identically to the DSHandle that it aliases, except that it would have different provenance reflecting both the provenance of the original DSHandle, and subsequent operations made to retrieve it. In the above example, then, `c[0]` would return a newly created DSHandle that aliased the original DSHandle for `a`.

6.1 Other Issues

6.1.1 Naming

OPM does not specify a naming mechanism for globally identifying artifacts outside of an OPM graph. DSHandles are given a URI. That fits in with a proposed OPM modification to use a Dublin Core identifier to identify artifacts [15].

6.1.2 Collections

The Swift provenance implemenation has two models of representing containment for DSHandles contained inside other DSHandles (arrays and complex types):

1. constructor/accessor model: in this model, there are 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 DSHandles as discussed in the alias section elsewhere;
2. container/contained model: relations are stored directly between DSHandles indicating that one is contained inside the other, without intervening processes. These relations can always be inferred from the constructor/accessor model.

6.1.3 Other OPM proposals

The Swift entry made a minor proposal to change the XML schema to better reflect the perceived intentions of the OPM authors [16]. 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. An OPM modification proposal for this is forthcoming. In Swift, this information is often available through the karajan thread ID which closely maps to the Swift process execution hierarchy: a Swift process contains another Swift process if its Karajan thread ID is a prefix of the second processes Karajan thread ID. The Swift provenance database stores values of artifacts/DSHandles when those values exist in-core (for example, when a DSHandle represents and int or a string). There was some desire in the PC3 workshop for a standard way to represent this, and a modification proposal may be forthcoming.

6.1.4 Missing Swift provenance

Some control-flow based provenance is not collected. A short contrived example illustrates this, by laundering the provenance of a boolean value using the `if()` language construct:

```
boolean b = ...input...;
boolean cleanb;
if(b) {
  cleanb = true;
} else {
  cleanb = false;
}
```

In the present implementation, the provenance of `cleanb` is recorded as an assignment from a constant, with link to the value of `b` and its associated provenance. Whilst the above example is contrived, similar situations may affect real applications. Two proposals for dealing with this are:

1. More functional language constructs, for example like the C `?:` trinary operator or the Haskell `if` expression, which require expressions to more explicitly include control-flow influences on their value. In such a model, the above example would be written as `cleanb = b ?  true :  false` and the `if` construct is represented as an operator process.

2. Make DSHandles that influence control flow be recorded as inputs to everything that runs inside the influenced scope. This would not need syntax or semantic changes to SwiftScript.

6.1.5 Voluminous amounts of data

The present model stores very large amounts of data. It may be desirable for Swift to optionally store less data, leading to a smaller provenance database with reduced accuracy. Some thought would need to be given to the meaningful options to be made available here.

References

[1] L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson. The Open Provenance Model. Technical report, University of Southampton, December 2007.

[2] 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 *Proceedings of the First IEEE International Workshop on Scientific Workflows (SWF 2007)*, pages 199–206, 2007.

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

[4] 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.

[5] 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.

[6] 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.

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

[9] J. Voeckler, G. Mehta, Y. Zhao, Ewa Deelman, and M. Wilde. Kickstarting Remote Applications. In *Second International Workshop on Grid Computing Environments*, 2006.

[10] G. Dong, L. Libkin, Jianwen Su, and Limsoon Wong. Maintaining transitive closure of graphs in sql. *International Journal of Information Technology*, 5, 1999. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.53.

[11] Support for Collections. Proposal distributed during the Third Provenance Challenge, 2009.

[12] Third Provenance Challenge. http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenance-Challenge, 2009.

[13] B. Clifford and L. Gadelha. Swift Team Entry at the Third Provenance Challenge. http://twiki.ipaw.info/bin/view/Challenge/SwiftPc3, 2009.

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

[15] B. Clifford. Dublin Core identifier naming proposal, Open Provenance Model Wiki. http://twiki.ipaw.info/bin/view/OPM/ChangeProposalDCNaming, 2009.

[16] B. Clifford. Change Proposal: Date Time in XML Schema, Open Provenance Model Wiki. http://twiki.ipaw.info/bin/view/OPM/ChangeProposalDateTime, 2009.

A Other databases experimented with

The present Swift implementation of provenance uses an SQL database. A number of other forms were briefly experimented with during development. The two most developed and interesting models were XML and Prolog. XML provides a semi-structured tree form for data. A benefit of this is that new data can be added into 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 Xpath's // operator which has specific benefits to provenance queries. Representing the data as Prolog tuples is a very different representation than a traditional database, but provides a very different query interface which can express interesting queries flexibly.

B `prov-init.sql`

```
-- this is the schema definition used for the main relational provenance
-- implementation (in both sqlite3 and postgres)

DROP TABLE processes;
DROP TABLE executes;
DROP TABLE execute2s;
DROP TABLE dataset_usage;
DROP TABLE invocation_procedure_names;
DROP TABLE dataset_containment;
DROP TABLE dataset_filenames;
DROP TABLE executes_in_workflows;
DROP TABLE dataset_values;
DROP TABLE known_workflows;
DROP TABLE workflow_events;
DROP TABLE extrainfo;


-- 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

-- TODO: no primary key here. should probably index both on execute_id and on
-- dataset_id for common queries? maybe add arbitrary ID for sake of it?

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

-- TODO probably desirable that this is part of executes table
-- but for now this is the easiest to pull data from logs.

-- TODO primary key should be execute_id
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

-- TODO this should perhaps be replaced with a more OPM-like model of
-- constructors and accessors, rather than, or in addition to,
-- a containment hierarchy. The relationship (such as array index or
-- structure member name) should also be stored in this table.
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

-- TODO dataset_id should be primary key
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)
    );
```