A Logic Programming Approach to Scientific Workflow Provenance Querying

Yong Zhao[1], Shiyong Lu[2]

[1]Microsoft Corporation, Redmond WA
[2]Department of Computer Science, Wayne State University, Detroit, MI
yozha@microsoft.com, shiyong@wayne.edu

Abstract. Scientific workflows have become increasingly important for enabling and accelerating many scientific discoveries. More and more scientists and researchers rely on workflow systems to integrate and structure various local and remote heterogeneous data and services to perform *in silico* experiments. In order to support understanding, validation, and reproduction of scientific results, provenance querying and management has become a critical component in scientific workflows. In this paper, we propose a logic programming approach to scientific workflow provenance querying and management with the following contributions: i) We identify a set of characteristics that are desirable for a scientific workflow provenance query language; ii) Based on these requirements, we propose FLOQ, a Frame Logic based query language for scientific workflow provenance, iii) We demonstrate that our previous relational database based provenance model, *virtual data schema*, can be easily mapped to the FLOQ model; and iv) We show by examples that FLOQ is expressive enough to formulate common provenance queries, including all the provenance challenge queries proposed in the provenance challenge series.

1 Introduction

Today, scientists use scientific workflows to integrate and structure various local and remote data and service resources to perform various *in silico* experiments to produce scientific discoveries. As a result, scientific workflows have become the de facto cyberinfrastructure upper-ware for e-Science and an efficient environment for computational thinking. A scientific workflow is a formal specification of a scientific process, which represents, streamlines, and automates the steps from dataset selection and integration, computation and analysis, to final data product presentation and visualization. A Scientific Workflow Management System (SWFMS) supports the specification, execution, re-run, and monitoring of scientific processes.

Provenance management is essential for scientific workflows to support scientific discovery, reproducibility, result interpretation, and problem diagnosis, while such a facility is usually not necessary for business workflows. Provenance metadata captures the derivation history of a data product, including the original data sources, intermediate data products, and the steps that were applied to produce the data product. Although several provenance storage and query models have been developed [17], they are based on query languages designed for other data types such as relational data, XML data, and Semantic Web data that are not specifically tailored for scientific workflow provenance. While it is still not clear which provenance model is most suitable for workflow provenance query representation and processing, we argue that a provenance query language should have the following characteristics: 1) The language should be based on a well-defined semantics to represent computations and their relationships, since only with such formalism in place can we define the model and syntax for representing and querying workflow provenance; 2) The language should be able to define, query, and manipulate data structures, as they are essential components of workflows, on which various operations are performed; 3) The language should have declarative syntax for both computation and data declarations and flexible query specification; 4) The language should allow the composition of simple queries into more complex queries; and 5) The language ideally should support inference capability for provenance reasoning.

Based on these requirements, we propose FLOQ, a Logic Programming (LP) approach to workflow provenance representation and querying. We use Frame Logic (F-Logic) [16] as the theoretic foundation and base our implementation on the FLORA-2 system [24]. Moreover, we demonstrate that our previous relational database based provenance model, *virtual data schema*, can be easily mapped to the FLOQ model, and show by various examples that FLOQ is expressive enough to formulate common provenance queries, including all the provenance challenge queries proposed in the provenance challenge series. Although the logical programming approach is not innovative by itself, we hope the introduction of such an approach into the provenance community can stimulate and motivate further research in this direction.

The rest of the paper is organized as follows. In Section 2, we briefly introduce Frame Logic and the FLORA-2 system. In Section 3, we show the mapping of the virtual data schema into Frame Logic, and the logic programming representation of the sample fMRI workflow used in the provenance challenges. In Section 4, we demonstrate the expressiveness of FLOQ. In Section 5, we discuss implementation issues and related work. Finally in Section 6, we draw our conclusions.

2 Frame Logic and FLORA-2

Frame Logic [16] provides a logical foundation for frame-based and object-oriented languages. It has a model-theoretic semantics and a sound and complete resolution-based proof theory. F-Logic combines clean and declarative semantics; expressiveness and powerful reasoning provided by deductive database languages; and rich data modeling supported by its object-oriented data model.

FLORA-2 [24] is both a LP language and an application development environment. The language is a dialect of F-Logic with meta-programming extensions (HiLog) [5] and logical updates (Transaction Logic). The implementation is built on top of the powerful and efficient XSB inference engine [20]. In the following, we introduce some of the key features of F-Logic using the FLORA-2 language syntax:

Objects and properties

Zhao[name -> "Yong Zhao", affiliation -> UChicago]
Lu[name-> "Shiyong Lu", affiliation->WSU, teaches(2008)->{CS300, CS501}]
UChicago[name-> "University of Chicago", location-> Chicago]
WSU[name-> "Wayne State University", location-> Detroit]

The above examples define two people and their associated universities, where **->** denotes the value of an attribute or method.

Class or Schema information

A set of similar objects can be categorized into a class. An F-Logic program can also represent the structural information of a class and its type signature (types for method arguments and results):

*Employee[name *=> string, affiliation *=> University]*
Faculty[teaches(integer) => Course]
University[name => string, location => City]

The above examples define a few classes and their type signature, where **=>** denotes the type of the method of the class, and ***** indicates that the signature can be inherited by a subclass. The original F-Logic distinguishes between functional (=>) and set-valued (=>>) methods. In FLORA-2 it has been simplified to use only set-valued methods. However, cardinality constraints can be specified, and {0:1}=> corresponds to functional methods.

Class hierarchy and Class membership

Faculty::Employee
Zhao:Employee
Lu:Faculty
UChicago:University
WSU:University

In the example, *S*::*C* denotes that *S* is a subclass of *C*, where *O*:*C* denotes that O is a member (instance) of C. Since the signature of *Employee* is defined as inheritable, *Faculty* gets the signature from it.

Predicates

In F-Logic, predicate symbols can be used in the same way as in predicate logic (e.g. Datalog). Information expressed by predicates can usually also be represented by frames. For instance, the *location* method of *UChicago* can be alternatively represented as:

Location(UChicago, Chicago)

Rules

Rules define the deduction process. Based upon a given set of facts, rules provide the mechanism to derive new information. Rules encode generic information of the form

rule head **:-** *rule body*

The rule body specifies the precondition that must be met, and the rule head indicates the conclusion.

To give an example:

?U[offers(?Y)->?C] :- ?F[teaches(?Y)->?C],?F[affiliation->?U]

The above rule specifies that if a faculty member *F* teaches a course *C* in year *Y*, and the person is affiliated with university *U*, then we conclude that the university *U* offers the course *C* in year *Y*.

Other features

FLORA-2 programs also allow the combined and nested specification of the above definitions, for instance

Lu[affiliation->WSU[name-> "Wayne State University"]]:Faculty

It also supports path expression where

Lu.affiliation

would result in *WSU* - the value of the method *affiliation* of object *Lu*.

It also provides aggregation functions such as average, count, max, etc. For instance:

?N = count{?C[?Y]|Lu[teaches(?Y)->?C]}

The above query counts the number of courses taught by *Lu* in each year. Due to space limitation, we do not give the details of these features; interested readers can look at the online tutorials for FLORA-2.

3 Mapping Virtual Data Schema to F-Logic

To illustrate how existing provenance systems can be mapped to and thus benefit from our logical programming approach, consider the following example. The virtual data schema [26] models the various relationships that exist among *datasets*, *procedures*, *calls* (to procedures), *workflows* (a set of dependent calls) and *invocations* (the actual executions of a specific call on physical resources), as well as *annotations* that associate metadata to these entities. Originally, the virtual data schema was interpreted as a relational model and implemented as a relational database in the virtual data system (VDS) [11]. Recently, we have evolved the schema and adapted it to the XML Dataset Typing and Mapping (XDTM) model [18], and integrated it into the Swift system [25].

XDTM is a data integration model that allows logical dataset structures to be specified separately from their physical representations so that workflows can be defined to operate on cleanly typed datasets. It also provides a mapping mechanism to map these logical structures to physical data access when a workflow is scheduled to execute. Swift is a fast, scalable and reliable Grid workflow system that builds on the XDTM data model, it combines a simple scripting language called *SwiftScript* for concise, higher level specification of complex parallel computations, and an efficient workflow engine to schedule the execution of large number of parallel tasks onto distributed and parallel computing resources. In Swift, all datasets are typed, and procedures take typed inputs and produce typed outputs, workflows are represented as a set of procedure calls, and their execution sequence is determined by data dependencies.

In this section, we show that the virtual data schema can also be modeled from an object oriented (OO) perspective, and it can be represented naturally in F-Logic. We also demonstrate via the sample fMRI workflow that our SwiftScript declarations of the workflow can be easily mapped into F-Logic programs. The details about the sample fMRI workflow can be found at the first provenance challenge site [10]. We only give a brief description of the workflow here for clarity purpose. The workflow graph is shown in Fig. 1. The inputs to the workflow are a set of *brain images* (Anatomy Image 1 to 4) and a single *reference brain image* (Reference Image), and each image has an associated header file with metadata in it. The outputs are a set of atlas graphics. The stages of the workflow are as follows:

Firstly, each brain image is spatially aligned to the reference image using *align_warp*. The output is a warp parameter set defining the spatial transformation to be performed (Warp 1 to 4). For each warp parameter set, the actual transformation of the image is done by *reslice*, which creates a resliced image of the original brain image with the configuration defined in the warp parameter set. All the resliced images are then averaged using *softmean*, producing a 3D atlas image. The averaged image is sliced along a plane in x, y, and z dimensions respectively into a 2D atlas using a program called *slicer*, and lastly, each 2D atlas is converted into a graphical atlas image using (the ImageMagick utility) *convert*.

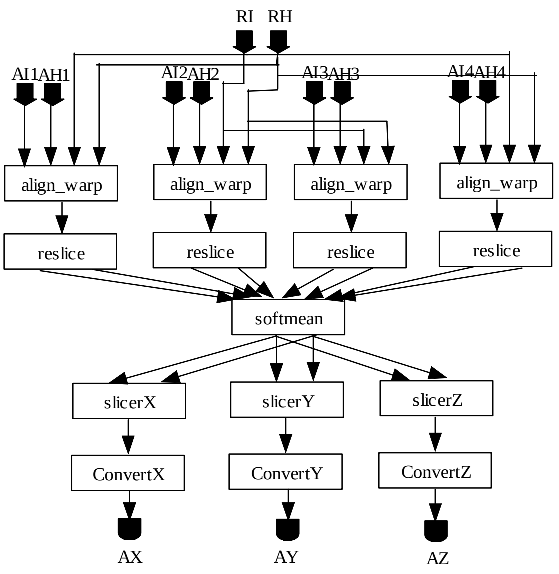

Fig. 1. Sample fMRI workflow

In the OO model, datasets all have *types*, and each dataset is an instance of its type. Procedures have typed signature, and each call is an instance of a procedure. To distinguish these different types, we firstly define a few base classes in F-Logic:

_type
_procedure
_anno

And use them to represent dataset types, procedures and annotations. SwiftScript is a typed workflow language. Datasets are typed (with structural definition), and procedures are also typed (with typed inputs and outputs). In below we show the type declarations of the fMRI workflow in SwiftScript to the left.

```
type Image;
type Header;
type Volume {
        Image img;
        Header hdr;
}
type Warp;
```

```
Image::_type.
Header::_type.
Volume::_type.
Volume[img{1:1}=>image,
        hdr{1:1}=>header].
Warp::_type.
```

This gets translated into the corresponding F-Logic statements to the right. Since FLORA-2 covers all the primitive types defined in Swift, so the mapping is a straightforward translation. Correspondingly, a SwiftScript procedure declaration can be mapped to its F-logic counterpart in a similar way:

```
(Warp w) align_warp (Volume iv, Volume reference, string overwrite)
{
}
```

```
align_warp::_procedure.
align_warp[iv=>Volume, reference=>Volume,overwrite=>string, w=>Warp].
align_warp[input->{iv,reference}, output->w].
```

The script defines a procedure called *align_warp*, which takes two volumes, a string option, and produces a warp file (which defines the spatial transformation to be performed to warp an input image to the reference image). The F-Logic program defines it as a class with its type signature. In order to capture whether a dataset is an input or output, we define two extra behavior attributes *input* and *output* for the procedure. The other procedures in the sample workflow can be mapped in the same manner and we omit the details. Now the call to a procedure (shown below in the top) is translated into F-Logic (shown in the bottom) as follows:

```
Volume vol1;
Volume std_vol;
Warp w1 = align_warp (vol1, std_vol, 'y');
```

```
vol1:Volume.
std_vol:Volume.
w1:Warp.
align_warp_uuid[iv->vol1, reference->std_vol,overwrite->'y', w->w1].
align_warp_uuid:align_warp.
```

Firstly the dataset declarations are translated into instance specifications of their corresponding types, and they are then supplied as values to the procedure inputs and outputs. Note that we generate a unique id for the call, and the call is an instance of its procedure class. For the calls, we want to find out which datasets are the inputs and outputs, instead of the parameter names. This can be achieved as follows:

?I[in->?D] **:-** *?I[?P->?D],?C[in->?P],?I:?C.*

?I[out->?D] :- ?I[?P->?D],?C[out->?P],?I:?C.

What the rules specify is that: if an instance *I* of a class *C* has a parameter *P*, and *P* is an input (or output) parameter with value *D*, then *D* is an input (or output) dataset to the instance *I*.

An invocation record captures the execution environment that a call is executed. It typically has information such as the execution host, start time, duration, exit code, memory usage, and stats of the input and output files. We model a record as an F-Logic object, and give an example below:

```
align_warp_inv_uuid[call->align_warp_uuid,
        host-> "uchost",
        arch-> "ia64",
        start_time-> "2008-02-14T09:55:33"^^_dateTime,
        duration-> "00:29:55"^^_duration ,
        exit_code->0].
```

For annotations, since we allow the association of metadata to any of the virtual data entities, including dataset types, datasets, procedures, procedure parameters, calls, invocations, the mapping is not as straightforward as the other ones. There are many ways to map an annotation into F-Logic. For instance, each annotation can be modeled as a predicate. But as annotations to an annotation should also be supported so that we can track the provenance of the annotation itself, we need to model the annotation as an object. We define a base class _anno, and each annotation is declared to be an instance of this base class. An annotation object takes the form as follows:

annotation_key [on(object, part, …)->annotation_value]:_anno.

For instance, if we want to annotate the procedure *align_warp* with *model=nonlinear*, the annotation is specified like this:

model[on(align_warp)-> 'nonlinear']:_anno.

With these simple mappings, we can already pose some interesting queries to the system. For instance, the following query can find all the types defined in the workflow.

```
Q:      ?X::_type.
A:      ?X = Image
        ?X = Header
        ?X = Volume
        …
```

Find all procedures that take Volume as a parameter:

```
Q:      ?X[?Y=>Volume]::_procedure.
A:      ?X=align_warp
        ?Y=iv

        ?X=align_warp
        ?Y=reference
```

Find procedure calls that ran on *ia64* processors:

```
Q:      ?[call->?X, arch-> "ia64"].
A:      ?X=align_warp_uuid
```

In Fig. 2 we show the specification of the sample fMRI workflow mapped to F-Logic terms, but omitting the detailed structural information about the datasets and procedures involved.

4 FLOQ Query Examples

In our virtual data query model, three major query dimensions were identified: (1) lineage information obtained by interrogating the patterns of procedure calls, argument values, and dependencies in the workflow graphs that describe the indirect nature of the production of a given data object; (2) prospective and retrospective provenance data, as provided by records of procedure definition, procedure arguments, and runtime invocation recording; and (3) metadata annotations that enrich this application-independent schema with application-specific information. In this section, we show the expressiveness of the FLOQ approach using extensive examples drawn from our single- and multi-dimensional queries and some of the core provenance challenge queries.

Lineage Queries: One of the key capabilities of a provenance system is to query the derivation history (lineage) of a data product, i.e. from which datasets this product is derived and by what procedures, and what datasets can be further derived from this product and by using what kind of procedures. These lineage relationships can be easily represented in F-Logic using the following predicates:

DirectlyDerived(?X, ?Y) :- ?Proc[in->?X, out->?Y],?Proc:_procedure.
Derived(?X,?Y) :- DirectlyDerived(?X,?Y).
Derived(?X,?Y) :- Derived(?X,?Z), Derived(?Z,?Y).

The first predicate specifies that if a procedure *Proc* takes *X* as an input, and produces *Y* as an output, then *Y* is directly derived from *X*. The second predicate defines the transitive closure of the derivation relationship, finding all *Y* that can be directly or indirectly derived from *X*. Now if we want to find what datasets can be derived from the dataset *vol1*, or what datasets are involved to derive the data product *atlas*, we can simply pose the following queries:

Q: *Derived(vol1, ?X).*
A: *?X=w1, svol1, atlas, atlas_x.ppm, atlas_x.jpg, atlas_y.ppm, …*

Q: *Deirved(?Y, atlas).*
A: *?Y=svol1, svol2, svol3, svol4, w1, w2, w3, w4, vol1, vol2, …*

Similarly, if we want to track what procedures are used to process a dataset and its derived datasets, we can define:

ConsumedBy(?X, ?P) :- ?P[in->?X], ?P:_procedure.
ConsumedBy(?X, ?P) :- DirectlyDerived(?X, ?Y),ConsumedBy(?Y, ?P).

As we can observe, these logic based derivation rules follow very closely to our natural thinking and are easy to define and understand.

```
align_warp_1[iv->vol1, reference->std_vol, overwrite->‘y’, w->w1] : align_warp.
align_warp_2[iv->vol2, reference->std_vol, overwrite->‘y’, w->w2] : align_warp.
align_warp_3[iv->vol3, reference->std_vol, overwrite->‘y’, w->w3] : align_warp.
align_warp_4[iv->vol4, reference->std_vol, overwrite->‘y’, w->w4] : align_warp.

reslice_1[w->w1, ov->svol1] : reslice.
reslice_2[w->w2, ov->svol2] : reslice.
reslice_3[w->w3, ov->svol3] : reslice.
reslice_4[w->w4, ov->svol4] : reslice.

softmean_1[iv->{svol1,svol2,svol3,svol4}, ov->atlas] : softmean.

slicer_1[iv->atlas, dimension->‘x’, ppm->‘atlas_x.ppm’] : slicer.
slicer_2[iv->atlas, dimension->‘y’, ppm->‘atlas_y.ppm’] : slicer.
slicer_3[iv->atlas, dimension->‘z’, ppm->‘atlas_z.ppm’] : slicer.

convert_1[from->‘atlas_x.ppm’, to->‘atlas_x.jpg’] : convert.
convert_2[from->‘atlas_y.ppm’, to->‘atlas_y.jpg’] : convert.
convert_3[from->‘atlas_z.ppm’, to->‘atlas_z.jpg’] : convert.
```

Fig. 2. F-Logic Program for the Sample fMRI Workflow

In contrast, a relational database based approach usually requires a more strictly defined schema such that procedures cannot be specified column-wise (they have various parameters), causing lineage queries to use expensive self-joins [23]. Since not all DBMS support recursion, such queries may have to be implemented in stored procedures that involve complex programming. Our XML based approach in VDS still required a pre-defined XML schema to write template-based queries, although the XQuery engine did provide the flexibility to join across multiple schemas and query recursively. We list the XQuery program to find all derived datasets in below for comparison purpose, and it is obvious that the query is less intuitive than the F-Logic counterpart.

```
declare namespace v='http://www.griphyn.org/chimera';
declare function v:lfn_tree($lfn as xs:string) as item()* {
        let $d := //derivation[.//lfn[@file=$lfn][@link='input']]
        return ( $lfn,
        for $out in $d//lfn[@link='output']/@file  return v:lfn_tree($out))
};
let $f := v:lfn_tree(‘vol1’);
return distinct-values($f)
```

Virtual Data Relationship Query: Virtual data relationship refers to the bindings between dataset types, procedures, calls, invocations, and the queries focus on the attributes of such entities. The query of "find all the procedures that have an input of type *Volume* and an output of type *Warp*" can be formulated as:

```
?Proc[input->?X, output->?Y]:_procedure,
?Proc[?X=>Volume, ?Y=>Warp].
```

Similarly, the query of "find all calls to procedure *align_warp*, and their runtimes**,** with argument *reference=std_vol* that ran in less than 30 minutes on non-*ia64* processors" can be formulated as:

?Inv[call->?C, duration->?d, not arch->"ia64"],
?d<= "00:30:00"^^_duration,
?C[reference->std_vol]:align_warp.

Annotation Queries: Annotation queries can be used to find any application specific information about various virtual data entities, such as procedure description and data curation, and to discover certain procedures and datasets by their associated metadata. The query of "show the values of all annotation predicates *developerName* of procedures that accept or produce an argument of type *Volume* with predicate *Model=nonlinear"* is formulated as:

developerName[on(?Proc)->?Name],
?Proc[?=>Volume]::_procedure,
Model[on(?Proc)->'nonlinear'].

Aggregation Queries: Aggregation queries can perform basic statistical mining over the provenance information, which can be useful for reporting purpose and anomaly detection. For instance, one of the following queries identifies jobs run unusually long, and the other one does a monthly tally of the total jobs run in a year. The query of "find all the align_warp invocations that ran three times longer than the average run time" is formulated as:

?X=avg{?d|?Inv[call->?C, duration->?d], ?C:align_warp},
?Inv[call->?C, duration->?d], ?C:algin_warp,
?d > 3 ?X.*

To list the total number of jobs run in each month of 2007, we use the query:

?X=count{?Inv[?M]| ?Inv[start->?T],
?T[_month->?M]@_basetype, ?T[_year->2007]@_basetype}.

Provenance Challenge Queries: Several provenance challenge queries are already discussed in previous sections, including Q1, Q6 (lineage), Q4 (relation), Q5, Q8, and Q9 (annotation). We discuss the rest of the queries below.

Q2: Find the process that led to Atlas X Graphic, excluding everything prior to the averaging of images with softmean.

This query is similar to the lineage queries, here, we are tracing back to a dataset's source. We can follow similar logic to answer this query:

ProducedBy(?X, ?P) :- ?P[out->?X], ?P:_procedure.
ProducedInBetween(?X, ?Y, ?P1, ?P2) :- ProducedBy(?X, ?P1), Derived(?Y, ?X), ConsumedBy(?Y, ?P2).

Basically the rule specifies that if a dataset *X* is the output of procedure *P1*, and *X* is somehow (directly or indirectly) derived from another dataset *Y*, which is the input of procedure *P2*, then we stop tracing back. The query can be posed as:

Q: *ProducedInBetween('atlas_x.jpg', ?Y, ?P, softmean).*
A: *?P = convert, slicer, softmean*
?Y = altas_x.ppm, atlas, svol1, svol2, svol3, svol4

Q3. Find the Stage 3, 4 and 5 details of the process that led to Atlas X Graphic.

We can track the depth of derivation by slightly modifying the definition of *ProducedBy*, and add the depth information in the rules:

ProducedBy(?X, ?P, ?D) :- ?P[out->?X], ?P:_procedure, ?D is 1.
ProducedBy(?X, ?P, ?D) :- DirectlyDerived(?X, ?Y), ProducedBy(?Y, ?P, ?D0), ?D is ?D0 + 1.

The rules specify that if a dataset *X* is directly produced by a procedure *P*, then the depth of derivation is 1; otherwise if *X* is directly derived from a dataset *Y*, and *Y* is produced by *P* at some depth *D0*, then the depth of derivation for *P* and *X* is *D0* plus 1. Now the query can be posed as:

Q:	*ProducedBy('atlas_x.jpg', ?P, ?S), ?S >=3, ?S=<5.*
A:	*?S = 3, ?P = softmean*
	?S = 4, ?P = reslice
	?S = 5, ?P = align_warp

When there are multiple paths that can be traced back from a data product, we can define the depth as the longest path to that data product. The rules would be slightly more complex, but still follow the same idea. The rules also require the workflow to be acyclic, which is the case in Swift. Otherwise, we may go into an infinite loop.

Q7. A user has run the workflow twice, in the second instance replacing each procedure (convert) in the final stage with two procedures: ppmtopnm, then pnmtojpeg. Find the differences between the two workflow runs.

Q7 turned out to be quite challenging for most of the teams. There were only one or two teams that could tackle this query. Using FLOQ, we can use the predicate for Q2 to find the answers to this query:

Q:	*ProceducedInBetween('atlas_x.jpg', ?Y, ?P, softmean).*
A1:	*?P=convert, slicer, softmean*
A2:	*?P=pnmtojpeg, ppmtopng, slicer, softmean*

The solutions to *?P* in the two cases would identify the differences exactly as they are, and also find the different intermediate datasets that have been produced.

Modification Queries: Modification queries allow the ability to couple queries with updates to define new procedures, annotations, and work requests, for instance, changing a procedure argument, replacing a procedure in a workflow, or editing a subgraph of a workflow. It is worth noting that FLORA-2 also supports updating the knowledge base, such as inserting and deleting facts and rules on-the-fly, and the updates can also be conditional, i.e. based on some rules. In our case, we can specify that for each call to *convert*, we insert two consecutive calls *ppmtopnm* and *pnmtojpeg*, and then delete the calls to *convert*, in this way, the workflow is transformed into a new one with similar functionality.

5 Discussions and Related Work

Provenance management has become an important functionality for most scientific workflow management systems [2,8]. The Kepler system implements a *provenance* *recorder* [1] to record information about a workflow run, including the context, data derivation history, workflow definition, and workflow evolution. The myGrid/Taverna system [22] uses Semantic Web technologies for representing provenance metadata at four levels: process, data, organization, and knowledge. Two levels of ontologies are used. A domain-independent schema ontology is used to describe the classes of resources and the properties between them that are needed to represent the four levels of provenance. A domain ontology is used to classify various types of resources such as data types, service types, and topic of interest for a particular domain. The VisTrails system [12] supports provenance tracking of workflow evolution in addition to data derivation history. In VisTrails, workflow evolution provenance is represented as a rooted tree, in which each node corresponds to a version of a workflow, and each edge corresponds to an update action that was applied to the parent workflow to create the child workflow. The above provenance systems are tightly coupled with their scientific workflow environments. A couple of stand-alone provenance systems have also been developed, including the PReServ system [13] and the Karma system [21]. PReServ supports the recording of interaction provenance, actor provenance, and input provenance with the Provenance Recording Protocol (PReP), which specifies the messages that actors can asynchronously exchange with the provenance store to support provenance submission.

The recent Open Provenance Model [19] effort tries to identify key relationships in data provenance systems, such as the transitive relationship of derivation, and in the mean time maintains openness about the alternative views that different users may have over the same data production process. Although the actual model and query language implementations are not yet discussed, we think a logic-based approach is natural to explore. Existing provenance systems have chosen to use different technologies for provenance querying, of which relational databases, XML, and RDF are three representative approaches [17]. For relational database based approach, persistence and indexing are the obvious advantages, however, it is difficult to define and integrate different schemas for different entities involved in the provenance space, and the queries for cross-entity joins and transitive relationships are not easy to write and extend. XML/XQuery based approaches also allow to define schemas and provide more flexibility in such definitions (can be semi-structured), yet, they lack the extensibility to define deductive rules, such that different use cases have to be implemented as different query templates. RDF/RDFS based approaches support interoperability and inference, but the triple style assertions make knowledge representation flat, requiring a lot more statements to define class and schema information. RDF/RDFS based approaches also lack the deductive rule engine, and as a matter of fact, many RDF systems choose to use an F-Logic system as its inference engine. OWL and some flavors of Description Logic are also candidates for knowledge representation, however, they focus more on the Class-level (T-Box) relationships, and do not have the expressiveness in instance level knowledge representation, and they also lack the reflective inspection capabilities (query schema, rule definitions themselves) presented in F-Logic. Again, they often engage an F-Logic system as their inference engine too [15].

Previously, F-Logic has been applied to workflow verification [9] and program query [14], they share some similarities to our approach, as all need to represent workflow and program structure to some extent. Our work can be actually extended to provide type checking (for instance, a procedure call needs to supply the right type of datasets to the procedure parameters) that is done programmatically in Swift, and if we choose to put more detailed information about Swift programs into the knowledge base, we can perform verification and program query, and even workflow scheduling within the FLORA2 engine using transaction logic.

While XSB has been shown to be highly efficient, scalability is an issue for large-scale provenance management, as the whole knowledge base may not fit into memory. We plan to address this issue by 1) developing a partitioning scheme for provenance data so that different portions of provenance information are loaded into memory dynamically as needed; 2) investigating database persistence techniques [7] for logic programs where tables can be stored externally in relational databases and loaded on-demand.

6 Conclusions and Future Work

In summary, we have described FLOQ, our Frame Logic based approach to representing and querying the virtual data provenance model, and we have demonstrated that our previously identified provenance problems and challenges can be addressed by this approach. Our Swift workflow definitions can be mapped to F-Logic programs in a straightforward manner, and provenance queries can be written and extended with great expressiveness and flexibility. For future work, we plan to 1) extend the translations of other SwiftScript program declarations to F-Logic and enable type checking and workflow structure querying. We can also apply F-Logic to type inference, so that procedure signatures can be derived instead of explicitly specified, which can further simplify SwiftScript; and 2) generalize the logical programming descriptions and apply them to more general workflow provenance problems, not limiting to the virtual data schema defined in the Swift system.

References

1. Altintas, I., Barney, O., & Jaeger-Frank, E., Provenance collection support in the Kepler Scientific Workflow System. In International Provenance and Annotation Workshop (IPAW06), LNCS 4145: 118-132, 2006.
2. Biton, O., Boulakia, S., Davidson, S., Hara, C., Querying and Managing Provenance through User Views in Scientific Workflows, ICDE 2008.
3. Bose, R., Foster, I., Moreau L., Report on the international provenance and annotation workshop (IPAW06). SIGMOD Records, September 2006.
4. Buneman, P., Khanna, S., and Tan. W.-C. Why and Where: A Characterization of Data Provenance. In International Conference on Database Theory, 2001.
5. Chen, W., Kifer, M., Warren, D.S., HiLog: A Foundation for Higher-Order Logic Programming, Journal of Logic Programming 15:3, 187-230, 1993.
6. Clifford, B., Foster, I., Voeckler, J., Wilde, M., Zhao, Y., Tracking Provenance in a Virtual Data Grid, Journal of Concurrency and Computation, Practice and Experience, 2007.
7. Costa, P., Rocha, R., Ferreira, M., Tabling Logic Programs in a Database, Proceedings of the 21st Workshop on (Constraint) Logic Programming (WLP07), 2007.

8. Davidson, S., Boulakia, S., Eyal, A., Ludäscher, B., McPhillips, T., Bowers, S., Anand, M., Freire, J., Provenance in Scientific Workflow Systems. IEEE Data Eng. Bull. 30(4): 44-50 2007.
9. Davulcu, H., Kifer, M., Ramakrishnan, C. R., and Ramakrishnan, I. V. 1998. Logic based modeling and analysis of workflows. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, United States, June 01 - 04, 1998). PODS 1998.
10. http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge, June 2006.
11. Foster, I., Voeckler, J., Wilde, M., Zhao, Y., Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation, In 14th Conference on Scientific and Statistical Database Management, 2002.
12. Freire, J., Silva, C. T., Callahan, S. P., Santos, E., Scheidegger, C. E., Vo, H. T., Managing Rapidly-Evolving Scientific Workflows, Proceedings of the International Provenance and Annotation Workshop (IPAW06), 2006.
13. Groth, P., Miles, S., Tan, V. and Moreau L. Architecture for Provenance Systems. Technical report, University of Southampton, October 2005.
14. Hajiyev, E., Verbaere, M., de Moor, O., codeQuest: Scalable Source Code Queries with Datalog, ECOOP 2006 –Object-Oriented Programming, pp. 2-27, 2006.
15. Kattenstroth, H., May, W., Schenk, F., Combining OWL with F-Logic Rules and Defaults, International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS07), 2007.
16. Kifer, M., Lausen, G., Wu, J., Logical Foundations of Object-Oriented and Frame-Based Languages, Journal of the ACM, 42:741-843, 1995.
17. Moreau, L. et al., The First Provenance Challenge, Concurrency and Computation, Practice and Experience, 2007.
18. Moreau, L., Zhao, Y., Foster, I., Voeckler, J. Wilde, M., XDTM: XML Dataset Typing and Mapping for Specifying Datasets. European Grid Conference, 2005.
19. Open Provenance Model, http://twiki.ipaw.info/bin/view/OPM, March 2008.
20. Rao, P., Sagonas, K. F., Swift, T., Warren, D. S., Freire, J., XSB: A System for Efficiently Computing Well-Founded Semantics. In J. Dix, U. Furbach, and A. Nerode, editors, Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR97), number 1265 in Lecture Notes in AI (LNAI), pages 2--17, Dagstuhl, Germany, July 1997. Springer Verlag.
21. Simmhan, Y., Plale, B., Gannon, D., A Performance Evaluation of the Karma Provenance Framework for Scientific Workflows, International Provenance and Annotation Workshop (IPAW06), LNCS 4145: 222-236, 2006.
22. Stevens, R., Zhao, J., Goble, C., Using provenance to manage knowledge of In Silico experiments. Briefings in Bioinformatics 8(3): 183-194, 2007.
23. Terracina, G., Leone, N., Lio V., Panetta, C., Experimenting with recursive queries in database and logic programming systems, Theory and Practice of Logic Programming, doi:10.1017/S1471068407003158, Cambridge University Press, 2007.
24. Yang, G., Kifer, M., and Zhao, C., FLORA-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. In Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE), Catania, Sicily, Italy, November 2003.
25. Zhao, Y., Hategan, M., Clifford, B., Foster, I., Laszewski, G.v., Raicu, I., Stef-Praun, T., Wilde, M., Swift: Fast, Reliable, Loosely Coupled Parallel Computation, IEEE International Workshop on Scientific Workflows (SWF07), Collocated with SCC 2007.
26. Zhao, Y., Wilde, M., Foster, I., Applying the Virtual Data Provenance Model, Proceedings of the International Provenance and Annotation Workshop 2006 (IPAW06), LNCS 4145, 2006.