# UP & DOWN: Improving Provenance Precision by Combining Workflow- and Trace-Level Information

*Saumen Dey*[*] *Khalid Belhajjame*[†] *David Koop*[‡] *Tianhong Song*[*]

*Paolo Missier*[§] *Bertram Ludäscher*[*]

## Abstract

Workflow-level provenance declarations can improve the precision of coarse provenance traces by reducing the number of "false" dependencies (not every output of a step depends on every input). Conversely, fine-grained execution provenance can be used to improve the precision of input-output dependencies of workflow actors. We present a new logic-based approach for improving provenance precision by *combining downward and upward inference*, i.e., from workflows to traces and vice versa.

## 1 Introduction

Many scientific workflow systems have been instrumented to capture workflow execution events as provenance. Typically, such events record the computational steps that were invoked as part of a workflow execution as well as the data that were input to and output by each step. These recorded events can be used to trace data provenance by identifying the input values that contributed to an output data value generated as a result of the workflow execution. In practice, however, this application is limited by the *black box* nature of the actors—the computational modules that implement the steps of the workflow.

To illustrate this, consider the actor $A$ shown in Fig. 1. Assume that an execution of $A$ uses the input values $v_{I_1}$ and $v_{I_2}$ for the input ports $I_1$ and $I_2$, respectively, and output ports $O_1$ and $O_2$ produce values $v_{O_1}$ and $v_{O_2}$, respectively. Because the actor is a *black box* we cannot assert whether the value $v_{O_1}$ was derived from (i) $v_{I_1}$, (ii) $v_{I_2}$, (iii) both $v_{I_1}$ and $v_{I_2}$, or (iv) none of the inputs. Similarly, we can not assert how the value $v_{O_2}$ was derived from $v_{I_1}$ and $v_{I_2}$. By derivation, we refer to the transformation of a data value into another, an update of a data value resulting in a new one, or the construction of a new data value based on a pre-existing one [13]. All we can safely conclude is that the invocation of the actor used the input values $v_{I_1}$ and $v_{I_2}$ and generated the values $v_{O_1}$ and $v_{O_2}$. While useful, this information is not sufficient to trace fine grained dependencies of data values produced by workflow executions. For the actor $A$ shown

[*]University of California, {scdey,thsong,ludaesch}@ucdavis.edu
[†]Université Paris-Dauphine, Khalid.Belhajjame@dauphine.fr
[‡]New York University, dakoop@nyu.edu
[§]Newcastle University, Paolo.Missier@ncl.ac.uk

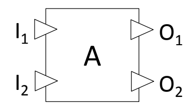
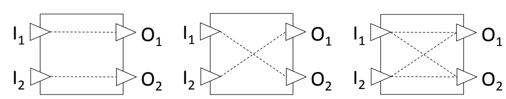



Figure 1: An actor $A$ with two input ports $I_1$, and $I_2$ and two output ports $O_1$ and $O_2$. Do the outputs depend on none, one, or both of the inputs?




(a) (b) (c)

Figure 2: Three examples of internal port dependencies of the actor $A$ shown in Fig. 1. There could be 13 more such dependencies.

in Fig. 1, there could be 16 possible ways output ports are dependent on the input ports, three of which are shown in Fig. 2. If we determine that one of the input values was incorrect, e.g., due to a malfunctioning sensor, we would potentially need to invalidate many useful results than if we knew that the input was only used for an insignificant output. This is one of many reasons showing the importance of finding fine-grained dependencies.

In general, if there are $n$ input ports and $m$ output ports for an actor, then there are $2^{mn}$ possible internal port dependencies. With $k$ such actors in a workflow, there are $2^{kmn}$ possible internal port dependency models of the workflow, i.e., possible models of the workflow.

In this paper, we propose a framework that takes a workflow specification and a set of provenance information to generate all possible data dependency models. By dependency models, we mean a graph in which the nodes represent the input and output ports of the actors, and the edges specify derivation dependencies between the ports. Notice that the framework generates multiple dependency models, all of which are possible. Fig. 2 shows three possible dependency models of the workflow with a single actor $A$ shown in Fig. 1. However, given the execution trace of a workflow execution, only one of the possible data dependency models reflects the true dependencies between the ports, whereas the remaining data dependency models are false positives. Figure 3 depicts the overall framework of our proposal towards reducing the number of false positive data dependency models.

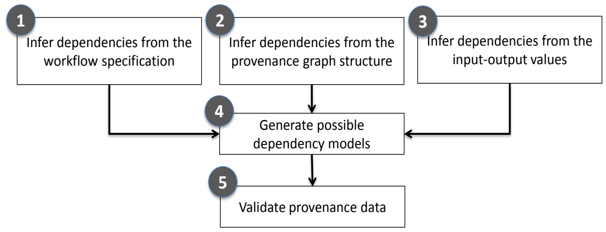



Figure 3: Components of our solution for inferring dependencies.

During the design of a workflow, the designer knows some internal details of the actors, e.g., (i) data on an output port can only be produced only after using data from a specific subset of input ports, (ii) two outputs are generated using exact set of inputs, etc. In step ①, we assume that our workflow model captures these additional design level information and analyze them to generate additional dependencies. In ②, we analyze the provenance graph structure and infer further dependencies by computing the transitive closure of data dependencies. Given a set of provenances either by (i) collecting actual workflow execution traces, or (ii) probing the input values, executing the workflow, and collecting provenances from the execution traces, in ③ we analyze input and output data values towards understanding the input-output dependencies. Steps ①, ②, and ③ generate additional dependency information, which this framework encodes as constraints and uses them while generating all the possible models of the workflow in step ④. Each of these models is an "improved" version of the given workflow, i.e., they have more dependency information, and these models are in turn used in step ⑤ to validate a provenance graph of an unknown origin (i.e., workflow) and improve it by injecting more dependencies. This improved provenance then can be used to have better result analysis, debugging, etc.

**Running Example.** We use the *Climate Data Collector* workflow to showcase the features of the proposed framework. We assume a set of sensors that collect temperature, pressure, and other climate observations including various flags. Given a sensor id, the `ReadSensor` actor reads from the sensors and returns *temperature*, *pressure*, and three flags, *flagA*, *flagB*, and *flagC*. The second actor, `SensorLogic`, takes these three flags and returns two flags *weatherCode* and *temperatureCode*. The flag *weatherCode* specifies if the weather was normal or not and *temperatureCode* indicates if the temperature was read in Celsius or in Fahrenheit. Another actor, `ConvertToKelvin`, uses the *temperatureCode* flag and the temperature reading to convert the temperature to Kelvin. A final actor `RangeCalculation` accepts the

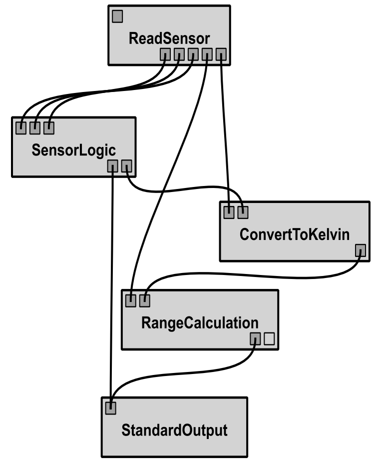



Figure 4: *Climate Data Collector* workflow, an example workflow that processes climate data read from a sensor.

pressure and temperature and computes a range. This workflow, defined in VisTrails, is shown in Figure 4.

If we consider all these actors as black boxes, `ReadSensor`, `SensorLogic`, `ConvertToKelvin`, and `RangeCalculation` actors individually have 32, 64, 4, and 4 input-output dependency models and thus, the workflow as a whole has **32768** possible dependency models. These many models makes it impossible to make any specific claims about dependencies, but if we use provenance information, we can reduce the number of possibilities.

For example, all the outputs of `ReadSensor` depend on a single input and there is no output if the sensor is not available. These information reduces 32 possible dependency models to exactly one which makes result analysis easier. Unfortunately, neither the workflow specification nor the provenance clearly capture these dependencies. We use this example workflow and show how we can capture additional dependency information, how we can use this information, and how we can improve the possible dependency models in Section 4.

The paper is organized as follows. We start by stating the assumptions needed for workflow model and provenance model used in our approach in Section 2. We then explain our proposed framework and the logical architecture in Section 3. In Section 4, we discuss our implementation using the example workflow and then describe related work in Section 5. We conclude the paper and discuss ongoing work in Section 6.

## 2 Models

**Workflow Model.** We base our workflow model on [3, 4]. A workflow $W = (V_w, E_w)$ is a directed graph whose nodes $V_w = A \cup P \cup C$ are *actors* $A$, *ports* $P$, and *channels*

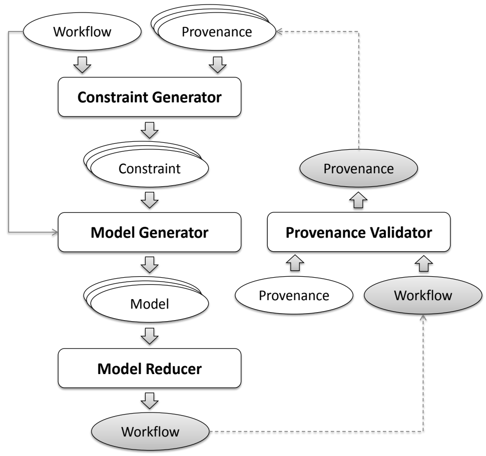



Figure 5: Logical architecture of how components interact to improve a workflow specification.

$C$. Actors (a.k.a. processes) are computational entities. An invocation of an actor reads data from *channels* and writes data into *channels*. The edges $E_w = in \cup out \cup pdep$ are either **input** edges $in \subseteq C \times P$, **output** edges $out \subseteq P \times C$, or **port dependency** edges $pdep \subseteq P \times P$. We also maintain an additional relation $port(P,A,P_t)$ where $P$ is a port in actor $A$ and $P_t$ specifies the type of port (input or output).

**Provenance Model.** The starting point for our provenance model is [12]. A provenance graph is a directed acyclic graph $G = (V_g, E_g)$ where the nodes $V_g = D \cup I$ represent either *data* tokens $D$ or *invocations* $I$. The edges $E_g = used \cup genBy \cup derBy$ are either **used** edges $used \subseteq I \times D$, **generated-by** edges $genBy \subseteq D \times I$, or **derived-by** edges $derBy \subseteq D \times D$. Here, a **used** edge $(\mathtt{i},\mathtt{d}) \in E_g$ means that invocation `i` consumes `d` as input; a **generated-by** edge $(\mathtt{d},\mathtt{i}) \in E_g$ means that `d` is an *output* token, generated by invocation `i`; and a **derived-by** edge $(\mathtt{d_i},\mathtt{d_j}) \in E_g$ means that data $\mathtt{d_i}$ was derived using data $\mathtt{d_j}$. We use two more relations *data* and *invoc*. The relation $data(D,P,A,R,V)$ specifies that data $D$ appeared at port $P$ of actor $A$ during the run $R$ with a value $V$. The relation $invoc(I,A)$ maps an invocation $I$ to its actor $A$. In addition, we use the auxiliary relation $ddep(X,Y)$ to specify that data $Y$ depends on data $X$ and $ddep^*(X,Y)$ is the transitive closure of $ddep(X,Y)$.

```
(1) ddep(X,Y) :- used(I,X),genBy(Y,I).
(2) ddep*(X,Y) :- ddep(X,Y).
(3) ddep*(X,Y) :- ddep(X,Z),ddep*(Z,Y).
```

## 3 Proposed Framework and Architecture

This framework takes a workflow specification $W$ and a set of provenance graphs $Gs$ and infers the internal (i.e., internal to an actor) port dependencies by analyzing $W$ and $Gs$. It then encodes all these inferred dependencies as constraints, which are used to reduce the number of possible workflow models. It uses the *Generate-and-Test* pattern from *Answer Set Programming (ASP)* [11], i.e., we use sets of *stable models* [9] to represent all possible models that are consistent with the generated constraints.

Let us consider the snippet of ASP program we have used in this framework.

```
pd(X,Y) v pnd(X,Y):- in(X,A), out(A,Y).
```

Here, $pd(X,Y)$ relation means that $Y$ depends on $X$ and $npd(X,Y)$ relation means that $Y$ does not depend on $X$. If there are **n** $\mathtt{in}(X,A)$ relations and **m** $\mathtt{out}(A,Y)$ relations in $W$, then there would be $2^{mn}$ dependency models for the actor $A$. In one extreme possible model, we would have all the $pd(X,Y)$ as "true" and in another extreme possible model, we would have all the $npd(X,Y)$ as "true". In all possible models, we would have some of $pd(X,Y)$ as "true" and some $npd(X,Y)$ as "true". Now, this framework applies the constraints in the following way: if a possible model has one $npd(X,Y)$ as "true" and for the same $(X,Y)$ pair a *constraint* has been observed to be "true" as discussed in Section 1 then its a contradiction as the *constraint* specifies that node $Y$ depends on node $X$. The framework applies the *negation as failure* principle and excludes this model.

The proposed framework, as shown in Fig. 5, has several components including the *Constraint Generator*, *Model Generator*, *Model Reducer*, and *Trace Validator*. The *Constraint Generator* takes a workflow $W$ and a set of provenance traces $Gs$, and performs three analysis tasks to generate constraints using the techniques we present next.

We analyze provenance graphs and capture such inferred data dependencies and use them to reduce the number of models. More specifically, in our provenance model, we capture $used(I,D)$, $genBy(D,I)$, and $derBy(D_1,D_2)$ edges. The $used(I_2,D)$, and $genBy(D,I_1)$ edges along with the mappings $invoc(I_1,A_1)$, and $invoc(I_2,A_2)$ unambiguously states that the actor $A_2$ depends on actor $A_1$. However, edges $used(I,D_1)$, and $genBy(D_2,I)$ are unable to unambiguously state that data artifact $D_2$ depends on data artifact $D_1$. The $derBy(D_1,D_2)$ edges describe dependencies among data artifacts and thus these edges are able to infer internal port dependencies of an actor. This framework analyzes these $derBy/2$ (i.e., $derBy$ is a two arity relation) relations and infers further data dependencies.

We use the provenance graph shown in Fig. 6 to describe how this framework analyzes these $derBy/2$) relations. This provenance graph has seven data artifacts $T$ though $Z$ and two $derBy/2$ edges, $derBy(Z,Y)$ and $derBy(Z,T)$ . Considering the $derBy(Z,T)$ edge we infer that $Z$ depends on $W$. Thus, given the $derBy(Z,Y)$

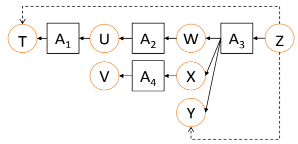



Figure 6: A provenance graph with two **derived-by** edges *derBy(Z,Y)* and *derBy(Z,T)*. Circles and rectangles represent data artifacts and processes, respectively.

and $derBy(Z,T)$ edges, we conclude that output port associated with $Z$ depends on input ports associated with $W$ and $Y$, which reduces the number of possible dependency models.

More generally, given provenance traces $Gs$, the $derBy/2$ relations in all the $Gs$ are analyzed. A $derBy$ edge could specify the dependency of (i) an output and an input of some actor or (ii) an output of one actor and an input of another actor. The framework infers these constraints using an algorithm, which is represented by the datalog rules 4 and 5 as shown below. Based on these two inferences, we get that port $P2$ depends on $P1$, which is a workflow specification level information by analyzing the $derBy/2$ relation from the provenance graph. Note that (i) provide exact dependency, but (ii) provides an over estimate.

```
(4) pdep(P1,P2) :-
      derBy(Y1,X1),data(X1,P1,A,R,_),
      data(Y1,P2,A,R,_).
(5) pdep(P1,P2) :-
      derBy(Y1,X1),data(X1,P1,A,R,_),
      data(D,P2,A,R,_),ddep*(D,Y1),
      ddep*(X1,D).
```

Input port dependencies can also be inferred by analyzing the input and output data values*. The framework computes the constraints using an algorithm, which is represented using rule 6 shown below.

```
(6) pdep(IP,OP) :-
      data(D1,IP,A,R1,IV1),data(D1,IP,A,R2,IV2),
      data(D2,OP,A,R1,OV1),data(D2,OP,A,R2,OV2),
      ¬R1=R2, ¬IV1=IV2, ¬OV1=OV2.
```

The *Model Generator* computes all possible dependency models in which output ports may depend on input ports for an actor and removes the models that contradict the constraints generated by the *Constraint Generator* as discussed in the beginning of this section. This component returns only those models that have no contradictions. The *Model Reducer* takes all the possible workflow models generated by *Model Generator* and combines them into one workflow model. This workflow model is an improved specification of the input workflow

*In this work, we assumed a simple model for input and output data value analysis. In this model, an output port of an actor depends on an input port if the values on the output port changes while changing values of an input and keeping the values for all other inputs same. In our future work, we plan to investigate further complexities of an actor.

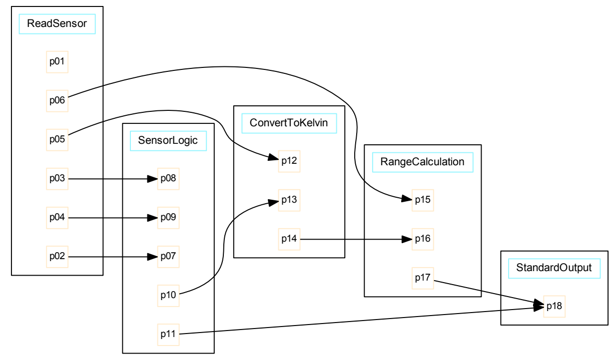



Figure 7: *Climate Data Collector* workflow represented by this framework. There are **32768** possible dependency models for this workflow.

model $W$, in which some of overestimates of the internal port dependencies have been removed. In *Provenance Validator*, the framework uses this improved workflow specification (i) to validate the provenance graph of unknown origin, (ii) to improve provenance graph by removing the inconsistencies.

# 4 Prototypical Implementation

In this section, we describe a prototypical implementation of the proposed framework.

Fig. 7 shows the *Climate Data Collector* workflow represented using this framework; actors are represented as rectangles and have ports “p01” through “p18”. The edges represent channels which connect output ports of one actor to input ports of another actor.

Fig. 8 shows that there are 32 possible dependency models for the *SensorLogic* actor. Without additional information, the framework will generate all 32 models. Similarly, other actors have many possible dependency models (as discussed in Section 1), and are not displayed because they mirror the *SensorLogic* actor.

The framework analyzes the workflow specification, provenance traces, and user-specified constraints as discussed in Section 1, and generates the constraints as shown in Fig. 9 which the framework applies while generating the possible dependency models. Fig. 9(a) shows the constraints generated based on the additional specifications† provided by the workflow designer. Fig. 9(b) shows all the constraints generated by the framework by analyzing the input-output value dependencies. Fig. 9(c) and Fig. 9(d) show all the constraints generated by the framework by analyzing the $derBy/2$ edges from the provenance graphs.

Fig. 10 shows the improved workflow specification based on the given workflow, provenance

†The definition of this specification is available in [5].

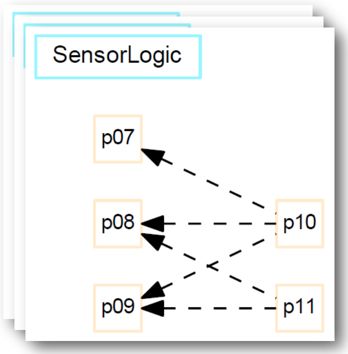



Figure 8: Possible dependency models for the *Sensor-Logic* actor. There are 32 possible dependency models for this actor alone.

graphs, and constraints. Based on the constraints generated by the framework, there is only one model for actors `ReadSensor`, `ConvertToKelvin`, and `RangeCalculation`. There are still 6 models for the `SensorLogic` actor. Thus, there are total 6 possible models for the *Climate Data Collector* workflow, and the framework generates all of them.

Using this example, we have demonstrated that the proposed framework can improve the workflow specification, and this information can be used to later validate provenance graphs with unknown origin.

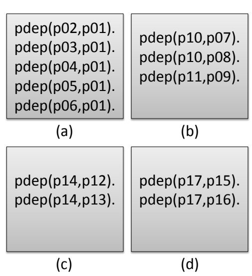



Figure 9: All of the constraints that are generated by the framework.

## 5 Related Work

The problem of defining or inferring data dependencies in scientific workflows has been investigated by a handful of researchers. For example, Bowers *et al.* proposed a declarative language for specifying fine-grained dependencies at the level of the workflow definition, which are propagated and applied to workflow trace events produced as a result of the workflow execution [2]. Their approach is complementary to ours, and we adopt a similar language to encode the dependencies that are dervied from workflow specifications and their corresponding

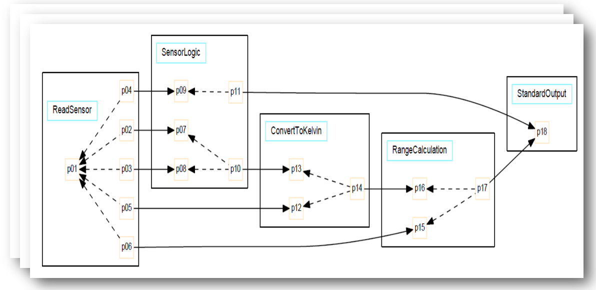



Figure 10: Improved workflow specification of the *Climate Data Collector* workflow after including one possible dependency model. There are be 6 such possible models, and this framework generates all of them.

trace events. However, that approach did not tackle the problem of inferring data dependencies.

Ghoshal *et al.* investigated the use of static analysis techniques to derive dependencies (or what they term mappings) between the inputs and outputs of an actor [10]. However, while they assume that the source code of the program implementing an actor is available, we tackle the problem of deriving data dependencies when the source code for the actors is not avaiable.

Garijo *et al.* proposed a framework where they identified data transformation and manipulation patterns (which they term motifs) that are commonly found in scientific workflows [8]. They distinguish two types of motifs: data intensive activities that are observed in workflows (data-oriented motifs), and different manners in which activities are implemented within workflows (workflow-oriented motifs). The first type is relevant to our work. However, in our framework, we are particularly interested in identifying if there is a dependency between an input and an output of an actor rather than the kind of data manipulation performed by the actor.

The above work and others, e.g., [1], [14], are related because they aim to understanding the kind of manipulation carried out by workflows. However, none tackle the problem of identifying fine-grained dependencies between input and output port for black box actors. In our prior work [6], we used the same *Generate-and-test ASP* based approach towards finding the possible orders of events. However, in this work we focus on finding the fine-grained dependencies using the same *Generate-and-test ASP* based approach.

## 6 Discussion

In this paper, we have presented a framework for inferring fine grained dependencies between the inputs and output ports of actors. We have described a probing based methods for inferring such dependencies. In doing so, we made the assumption that dependencies are static, in the sense that they hold across all invocations of the actors. In practice, however, dependencies may be dynamic, in the sense they may change across invocations of the actor. For example, the VisTrails system [7] provides a conditional actor (`If`) that uses modified dataflow logic to execute only one of two upstream workflows to generate an output based on a boolean input. There are also instances where two ports are provided to make the specification of an input possible in different formats, meaning only one port's value will eventually be used but is dependent on which are provided.

The framework that we presented in this paper needs to be refined in order to cater for the identification of dynamic dependencies. In particular, the user should be able to understand the cases (conditions under which) a given output is likely to depend or not on a given input. Note also, that so far, we treated input-output port equally. A classification of dependencies is needed to provide the user (workflow designer or provenance user) with better understading on the kind of relationship between the input and output ports. For example, distinguishing control-flow dependencies from data-flow dependencies. The later can be further classified to specify the kind of contribution a given input value had in the construction of the output value.

**Acknowledgments.** Supported in part by NSF ACI-0830944 and IIS-1118088.

## References

[1] P. Alper, K. Belhajjame, C. A. Goble, and P. Karagoz. Enhancing and abstracting scientific workflow provenance for data publishing. In *EDBT/ICDT Workshops*, pages 313–318. ACM, 2013.

[2] S. Bowers, T. M. McPhillips, and B. Ludäscher. Declarative rules for inferring fine-grained data provenance from scientific workflow execution traces. In *Proc. IPAW '12*, pages 82–96, 2012.

[3] V. Cuevas-Vicenttín, S. Dey, B. Ludäscher, M. Li Yuan Wang, and T. Song. Modeling and querying scientific workflow provenance in the d-opm. In *7th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2012.

[4] V. Cuevas-Vicenttín, P. Kianmajd, B. Ludäscher, P. Missier, F. Chirigati, Y. Wei, D. Koop, and S. Dey. The PBase scientific workflow provenance repository. *IDCC*, 2014.

[5] S. Dey, S. Köhler, S. Bowers, and B. Ludäscher. Datalog as a lingua franca for provenance querying and reasoning. In *TaPP '12*, 2012.

[6] S. Dey, S. Riddle, and B. Ludäscher. Provenance analyzer: exploring provenance semantics with logic rules. In *TaPP '13*. USENIX, 2013.

[7] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. Silva, and H. T. Vo. *The Architecture of Open Source Applications*, chapter VisTrails. Lulu.com, 2011.

[8] D. Garijo, P. Alper, K. Belhajjame, Ó. Corcho, Y. Gil, and C. A. Goble. Common motifs in scientific workflows: An empirical analysis. In *eScience*, pages 1–8. IEEE Computer Society, 2012.

[9] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th Int'l Conf. Logic Prog.*, volume 161, 1988.

[10] D. Ghoshal, A. Chauhan, and B. Plale. Static compiler analysis for workflow provenance. In *Proc. 8th Work. on Workflows in Support of Large-Scale Science*, WORKS '13, pages 17–27. ACM, 2013.

[11] V. Lifschitz. What is answer set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597, 2008.

[12] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *FCGS*, 27(6):743–756, 2011.

[13] L. Moreau and P. Missier. PROV-DM: The PROV data model, 2013.

[14] R. J. Sethi, H. Jo, and Y. Gil. Re-using workflow fragments across multiple data domains. In *Proc. SCC '12*, pages 90–99, Washington, DC, USA, 2012. IEEE Computer Society.