Data Provenance Inference in Logic Programming: Reducing Effort of Instance-driven Debugging

Mohammad Rezwanul Huq
Dept. of Computer Science
University of Twente
Enschede, The Netherlands
m.r.huq@utwente.nl

Alessandra Mileo
Digital Enterprise Research Institute
National University of Ireland
Galway, Ireland
alessandra.mileo@deri.org

Andreas Wombacher
Dept. of Computer Science
University of Twente
Enschede, The Netherlands
a.wombacher@utwente.nl

ABSTRACT

Data provenance allows scientists in different domains validating their models and algorithms to find out anomalies and unexpected behaviors. In previous works, we described on-the-fly interpretation of (Python) scripts to build workflow provenance graph automatically and then infer fine-grained provenance information based on the workflow provenance graph and the availability of data. To broaden the scope of our approach and demonstrate its viability, in this paper we extend it beyond procedural languages, to be used for purely declarative languages such as logic programming under the stable model semantics. For experiments and validation, we use the Answer Set Programming solver oClingo, which makes it possible to formulate and solve stream reasoning problems in a purely declarative fashion. We demonstrate how the benefits of the provenance inference over the explicit provenance still holds in a declarative setting, and we briefly discuss the potential impact for declarative programming, in particular for instance-driven debugging of the model in declarative problem solving.

Keywords

Data provenance, Logic programming, Provenance graph

1. INTRODUCTION

Data provenance is often used for auditing and debugging of data intensive applications. Provenance-aware applications automatically maintain data provenance which refers to the derivation history of data starting from its input sources [33]. Data provenance can be defined at different levels of granularity [4]. Fine-grained data provenance is defined at the value-level documenting the relationship among the input values, the output value and the associated processes. and it turned out to be very helpful for debugging purposes [16].

Provenance-aware applications have been proposed on various abstraction levels. In this paper, we are investigating the potential of the fine-grained data provenance as a debugging tool for a declarative programming language. A characteristic of declarative languages is that the user is specifying the property of the solution rather than the algorithm that produces it. As a consequence, the user does not specify a control flow along which the data is processed, but represents direct derivation rules for data, specifies a search space for a solution, and constraints the solution based on specified properties.

A class of these languages based on the stable model semantics, namely Answer Set Programming (ASP), has been intensively applied in the last decade to solve computationally hard problems [10]. The ability to deal with incomplete knowledge, conflicting and noisy input and common sense reasoning made ASP applicable to domains such as civil engineering, home healthcare, sensor networks, planning, bioinformatics, phylogenesis, system biology, industrial applications and more [25, 22, 23, 6, 12]. Despite this wide applicability, explaining unexpected outcome in an instance-driven way, by detecting errors in the knowledge model or in the inference rules related to a particular output, is still under investigated.

In this paper, an ASP program is designed to deal with online data. In the use case, different data streams such as twitter, rss and weather information is aggregated to give an indication on the accessibility rating of a particular road segment/location. The accessibility rating of a road segment is based on the traffic delay, i.e. whether it is expected that traveling along this road will cause delays. Here the ASP is beneficial due to the huge amount of combinations on which data can be available and how they can be aggregated. An explication of the current ASP program into a control-flow based language providing the same functionality would result in a multitude of the currently 120 lines of code.

In this paper, we discuss and compare two options to make the ASP solver provenance aware. The first option is to extend the logical programs to encode the provenance information explicitly as predicates. The second option is the inference of the fine-grained data provenance based on an explication of the search space and the an provenance inference method proposed in [18, 17, 15]. The option of extending

the reasoner to document explicit provenance has been overlooked since it is then reasoner specific while the other option of inferring provenance information is easily adaptable and extensible for other reasoners. Thus, the later approach will have more impact.

The difference to our previous work discussed in [16] is that in this paper, we extend the technique of static analysis to infer provenance based on a declarative language rather than a control flow dominated one such as Python. Furthermore, we also provide some basic formalisms to encode a logic program in form of a workflow. We infer the fine-grained provenance information based on the complete search space generated by the grounding of the logic program represented as workflow provenance graph. In this paper, we will illustrate how the proposed approach is complementary to ongoing efforts in finding solutions for debugging ASP programs. This will enable us to demonstrate how the benefits of the provenance inference over the explicit provenance still holds in a declarative setting.

2. BACKGROUND

2.1 Answer Set Programming and Streams

Answer Set Programming (ASP) [1, 20, 10] is a purely declarative and non-monotonic logic programming paradigm directed at solving computationally complex search problems using the "generate and test" approach, based on the stable model semantics [13] where solutions are represented by sets of atoms (*answer sets*) for which all the rules in the program are satisfied.

Among ASP's distinguishing feature is its ability to derive multiple answer sets. This is primarily achieved through *non-monotonicity* and the use of Negation As Failure (NAF). In general, this allows ASP to perform default reasoning as well as the ability to non-deterministically derive solutions even when reasoning under incomplete knowledge.

Given the expressivity of ASP in dealing with incomplete information, conflict resolution, constraint-based reasoning, non-deterministic choices and defaults, solutions to repurpose ASP reasoning for streaming data have been proposed [5] to apply ASP to stream reasoning in scenarios like Internet of Things, social networks, financial planning, resource management, health monitoring and many more.

In this paper, we focus on solutions for complex stream reasoning capabilities based on ASP, which can offer these reasoning capabilities along with pure declarativity in the problem specification [8]. The current work by Gebser et al. [9] provides an all encompassing software solution to ASP-based stream reasoning. *Oclingo* is an Answer Set Programming reasoner designed to work similarly with the normal syntax and semantics of the language, but with the addition of new functionality to allow for the streaming input of data.

Oclingo has the ability to reason upon different *frames of reference* in addition to it's powerful ASP-based reasoning. This allows programmers to define reasoning tasks in which temporal information can be deliberated upon and solutions to frame-based problems (e.g. involving *sliding windows* of knowledge) could be encoded. The Oclingo implementation achieves this by extending the ASP language to deal with emerging and expiring knowledge, and to reason with time-decaying logic programs. In Table 1, the part relevant to this paper of Oclingo's ASP syntax for stream reasoning is shown. In the next section, we describe the the way of representing a logic program in a data-dependent manner.

Table 1: Relevant Oclingo ASP Syntax

Syntax	**Description**
$h \leftarrow a_1, ..., a_m[, not\ a_{m+1}, ..., not\ a_n].$	Logical rule
$a \leftarrow .$	Fact
$\leftarrow a_1, ..., a_m[, not\ a_{m+1}, ..., not\ a_n].$	Constraint
$l\{h_1, .., h_n\}u \leftarrow [a_1, ..., a_m[, not\ a_{m+1}, ..., not\ a_n]].$	Choice [rule] $(l, u \in \mathbb{N})$
$\#external\ a.$	Indicates Streamed Input
$\#volatile\ t : \mathbb{N}$	Indicates time-decay rules

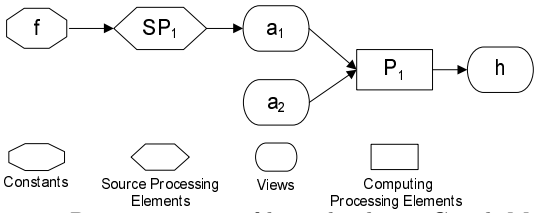

Figure 1: Representation of logical rules in Graph Model

2.2 Provenance Graph Model

The core concept of this paper is to infer provenance information in a logic program. Provenance information can be represented as a graph and referred to as *provenance graph*. Therefore, it is necessary to transform a logic program into a data-dependent provenance graph. A provenance graph G_p is a set of (V, E) where V denotes the set of vertices or nodes and E denotes the set of directed edges. We introduce a provenance graph model to distinguish different types of nodes. These are:

- **Constants**: represents a base fact mentioned in a logic program.
- **Source Processing Element**: represents an operation/rule that either assigns a constant/fact or reads data from a source.
- **Computing Processing Element**: represents an operation/rule that either computes a value/predicate based on its parameters.
- **View**: represents either a variable/predicate defined in the program or an intermediate result generated by a processing element.

A directed edge connecting two nodes represents the data dependence in the provenance graph. In the provenance model, every source and computing processing element generates a view. Further, a view node can be used as an input to multiple source and computing processing elements.

Assuming that, we have three rules in a logic program: i) $f \leftarrow .$, saying f is a fact (always true), ii) $a_1 \leftarrow f$, using fact f to derive predicate a_1 and ii) $h \leftarrow a_1, a_2$, having two (or more) predicates in the body, used to produce predicate h. Fig. 1 shows the provenance graph representing the

aforesaid rules. The first rule introduces a base fact f, represented by a constant node in the provenance graph. Later, this fact has been used to produce predicate a_1, represented by a view node and the rule itself is encoded in form of a source processing element SP_1. The next rule is represented by the computing processing element P_1 which has two body predicates a_1 and a_2. Both of them are considered as views in the provenance graph and they contribute to produce the predicate h, another view node in Fig. 1.

There are a few properties of both source and computing processing elements which are listed below:

- Windows: A window specifies a subset of data products/predicates used by an operation to produce an output data product/predicate. Therefore, a window with a predefined size is applied over the stream of input data products to limit the number of data products to be considered by the processing element.
- Trigger: In stream data, a source or computing processing element is repeatedly executed after elapsing a predefined interval, also known as trigger period.
- Input-output ratio: It refers to the ratio of the number of the data products/predicates contributed in a processing element to the number of the data products/predicates produced by the same processing element during the execution phase. As an example, an aggregate operation like *#count* considers all input data products to produce an output data product and thus, the ratio is $n : 1$ where n is the number of input data products in the window. There are a few operations where the ratio keeps changing. As an example, a *select* operation might take an n number of tuples as input and produce the tuples satisfying the condition. Therefore, The input-output ratio cannot be determined prior to the actual execution and these operations are referred to as *variable mapping* operations.

3. RELATED WORK

In the past few years some effort have been put against the design of debugging tools to promote widespread use of ASP in real world applications. Initial efforts would tackle categorization of logical errors to provide justifications [3]. [2, 11] propose the use of a debugging language to specify debugging statements, and rewrite them as an ASP program. More general approaches go beyond the propositional case, with particular attention to the complexity issues of grounding [30, 26]. Recent work proposes stepping techniques for ASP [27], relying on the intuition of a programmer on the applicability of rules. The integration of engineering tools in an IDE for ASP has also been put forward in the last few years, but works are still preliminary and mainly targeted to programmers, with graphical tools, spell checking and tracking mechanism [7, 19].

In our approach, we target both ASP programmer and domain experts in bridging the gap between practical experience in writing logic programs and domain expertise, used to go from a description of a problem to a logical model that can be validated over instances of interest.

This provides a complementary approach to debugging declarative programs which targets early stages of the modeling phase, from a problem domain to a program description. This steps are carried over incrementally, starting from easy formulation towards rules that can deal with more complex cases. In this process, the correctness of a logical model might turn out to be inaccurate when domain properties are expanded or if we omit some aspects of the domain of interest, which would require new rules and constraints. This dynamic aspect is even more important when the input series are partially unknown, like in streaming systems with unexpected chains of input and realize only later that our initial ASP formalization was not correct.

We believe a continuous graph-based view of how dynamic data provenance can explain outcome of streaming ASP formalizations at any of these stages, might help domain experts and ASP programmers spotting modeling issues earlier on and get to the intended characterization faster.

To the best of our knowledge, the use of data provenance for purely declarative languages has not been explored. Provenance aware workflow engines such as Kepler [21], Karma2 [34], Taverna [28] can help maintaining data provenance with some limitations. One of these limitations is that scientists have to learn the semantics of that particular workflow engine they facilitate and secondly, there is a possibility to have less expressiveness than in the native language, in which scientists might feel more comfortable. However, neither of these workflow engines can handle declarative languages.

In [32, 29], authors provide a complete DBMS with explicitly recorded provenance data, applicable to only subset of logic inference. In [14], authors proposed an approach that can reconstruct provenance of the manipulations done over the data in an open system like excel sheet or a programming tool like R. This approach used a library of basic transformations to infer and reconstruct provenance for a particular value. Work in [24] documents provenance by modifying the source code of a program automatically. It provides fine-grained data provenance after executing the script. In [31], authors instrumented Xlog, a logic programming language, by writing a Xlog interpreter known as PAXlog, to collect provenance. However, to collect provenance, a user guideline must be followed. In [16], we proposed an approach that can generate provenance graphs of a Python script by static code analysis.

We want to use provenance graphs over streaming ASP programs to be able to i) detect what data produced a particular fact in a solution at a particular time, and ii) visually represent the complete search space that might contribute to obtaining a particular literal in an answer set.

4. USE CASE DESCRIPTION

In this paper, we consider the problem of determining the accessibility of a particular road segment/location in a specific area (e.g London, UK). We take into account available data from various sources, namely i) the traffic conditions on major arterial roads, ii) the weather conditions being received from various stations within the area, and iii) Twitter status updates from users, specific to the geo-location and using some particular keywords or hashtags for filtering.

These sources provide streaming data, i.e. data tuples sent to the destination as soon as they are generated. To build an application that can constantly update the accessibility status of different road segments, we need to be able to process and reason about knowledge produced by the sources in a streaming fashion, based on some intelligent aggregation and reasoning over the information being received. For example, in worst case scenario, the program would infer that the location is *very inaccessible* if all three streams reports that it is difficult to enter or exit the area (e.g. slow traffic due to an accident; a snow blizzard hindering travel; and twitter status updates corresponding and/or confirming the same information). When streaming information is lacking or contradictory, the program would need to combine available knowledge in a qualitative ranking, in order to decide a suitable *accessibility rating* to aid in informing potential travelers. Contextual information or user preferences can also be used to guide the final results. The possibility of having multiple possible solutions supported by subsets of the available input streams can also be desirable when the qualitative metrics do not lead to any actual winning outcome.

In order to be able to deal with these different aspects, we declaratively encoded our problem in ASP. The logic program consists of two primary processes: a) generating the search space of plausible answer set solutions based on the input streaming data, and b) infer answer sets relevant only to the streaming data by navigating and pruning the search space. Grounding techniques for ASP generate a complete and defined search space domain and the ASP solver explores and prunes the search space (via heuristics) to find the desired solutions.

5. EXPLICIT PROVENANCE

The first approach proposed in this paper is to extend the logical program with additional logical rules to represent the fine-grained data provenance information explicitly encoded as predicates. The augmentation of the logical program with these additional logical rules can be done manually (as done for this paper) or be automated, by parsing and analyzing the logical program. For the purpose of investigating the usefulness of fine-grained provenance graphs for debugging ASPs and the pros and cons of explicit and inferred provenance information discussed in this paper, the way of augmenting the logical program is not relevant. We refer to the set of extra logical rules as *explicit provenance rules*. In this section, we describe the construction of the provenance rules per class of ASP rules, illustrate it with an example from the scenario and provide the translation of the derived provenance predicate into a graphical representation based on provenance graph model discussed in Sec. 2.2. In particular, the different rule types as indicated in Table 1 are clustered into the following classes: i) logic rule, ii) logic rule with equivalent head predicate, iii) choice rule with constraints and iv) logic rule with aggregation functions.

5.1 Logical Rule

In general, a rule R in a logical program has two parts: head and body. If the predicates in the body are satisfied then the predicate in the head of the rule can be inferred. The structure of a logical rule is given below, which extends the version in Table 1 by explicating the arguments of the involved predicates as a vector $\overrightarrow{arg}_{a_i} = (args_{1,a_i}, \ldots, args_{n_{a_i},a_i})$ respectively:

$$h(\overrightarrow{arg}_h) :- a_1(\overrightarrow{arg}_{a_1}), ..., a_M(\overrightarrow{arg}_{a_M}) \\ [, not\ a_{M+1}(\overrightarrow{arg}_{M+1}), ..., not\ a_N(\overrightarrow{arg}_{a_N})].$$

The predicate $h(\overrightarrow{arg}_h)$ in the aforesaid rule R constitutes the head of the rule, where as the body of the rule R is comprised of the predicates a_1, ... , a_N. Each predicate in both head and body of the rule may have several arguments represented by the vector $\overrightarrow{arg}_h$ and $\overrightarrow{arg}_{a_i}$ respectively. The number of arguments for each predicate may vary.

5.1.1 Provenance extension

To document explicit provenance, it is necessary to encode all predicates and argument bindings of the logical rule used to derive the predicate h into a provenance predicate. This is done by adding an extra logical rule for rule R to the original logic program. The construction of the corresponding provenance rule is quite straightforward. The provenance rule is a copy of the logical rule R with a modified head predicate.

To formulate the head predicate of the provenance rule R_{prov}, the keyword *'Prov'* is added as a suffix to the name of the predicate in the head part of rule R, thus $hProv$. The predicate $hProv$ has the following arguments:

1. arguments of the head predicate of rule R: $\overrightarrow{arg}_h$
2. a user defined rule identifier: $rule\_id$
3. for each positive predicate, a_i in the body of rule R:
   (a) name of the predicate: a_i
   (b) arguments of the predicate a_i: $\overrightarrow{arg}_{a_i}$

Negated predicates are not encoded, since in a provenance graph enumerating negated predicates may explode the graph and the negation is intended as Negation as Failure (NaF), which means that they have not been observed. As an example, for the aforesaid rule R, we add the following rule, $RProv$, which is used to capture the explicit provenance information.

$$hProv(\overrightarrow{arg}_h, rule\_id, a_1, \overrightarrow{arg}_{a_1}, \ldots, a_M, \overrightarrow{arg}_{a_M}) \\ :- a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}), \\ [not\ a_{M+1}(\overrightarrow{arg}_{a_{M+1}}), \ldots, not\ a_N(\overrightarrow{arg}_{a_N})].$$

5.1.2 Example

As an example Listing 1, lines 1 and 6 show two rules which are excerpted from the logic program of the use case (see Sec. 4). Both of these rules follow the aforesaid basic structure of a logical rule and therefore, an extra provenance rule per logical rule will be added to document provenance explicitly. Based on the provenance rule formulation discussed above, Listing 1, lines 4 and 9 show the provenance rules.

Listing 1: An example of a logical rule with it's corresponding provenance rule

```
1 riskvalue(rss, high, LOCATION) :- rss(
      STATUSTYPE, LOCATION, SEVERITY, TIME),
      negative(STATUSTYPE), SEVERITY>1.
```

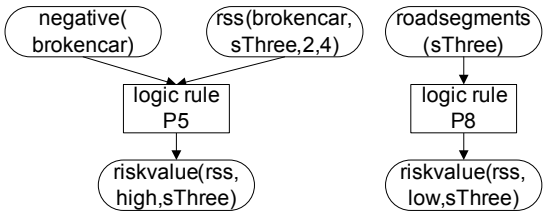

Figure 2: Example provenance graph for logical rule

```
2
3 % explicit provenance
4 riskvalueProv(rss, high, LOCATION, p5, rss,
      STATUSTYPE, LOCATION, SEVERITY, TIME,
      negative, STATUSTYPE) :- rss(STATUSTYPE,
      LOCATION, SEVERITY, TIME), negative(
      STATUSTYPE), SEVERITY>1.
5
6 riskvalue(rss, low, LOCATION) :- roadsegments(
      LOCATION), not riskvalue(rss, high,
      LOCATION).
7
8 % explicit provenance
9 riskvalueProv(rss, low, LOCATION, p8,
      roadsegments, LOCATION, null, null, null,
      null, null) :- roadsegments(LOCATION), not
      riskvalue(rss, high, LOCATION).
```

The *riskvalueProv* predicate in line 4 and 9 have different arity. Since a predicate in a logical program must always have the same arity, the remaining arguments of the predicate in line 9 are filled with *null* values. When constructing the provenance graph the *null* values are ignored.

5.1.3 Provenance Graph

To construct the provenance graph, we interpret the provenance predicates as found in the answer set of the logical program. Inverse to its construction the head and the positive body predicates can be derived knowing the arity of each predicate. The arity can either be derived from the remaining content of the answer set or be specified in a configuration file. The relation between input predicates and output predicate of the processing element is derived from the fact that the head predicate of the rule is defined by a single logical rule respectively. Thus the processing element is a logical rule.

Examples of the provenance predicates as derived from the use case are given below resulting in the provenance graph depicted in Fig. 2. Squares represent processing elements and ovals represent views. The logical rule processing element has a many to one input-output ratio (see Sec. 2.2), which means that all input views must contain a tuple to actually derive the output view.

```
riskvalueProv(rss,high,sThree,p5,
   rss,brokencar,sThree,2,4,negative,brokencar)
riskvalueProv(rss,low,sThree,p8,
   roadsegments,sThree,null,null,null,null,null)
```

The logical rule discussed in the previous section is the basic building block of a logical program. In the following some more complex constructions are discussed.

5.2 Projection Rule

A projection rule is a logical rule where the body of the rule contains a predicate which has at least one argument which is neither constrained by another predicate nor used in the head predicate. The schema of a projection rule is given below. Please note that for readability, we do not add the optional negated predicates in the body of the production rule as in the previous section.

$$h(\overrightarrow{arg}_h) :- a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}).$$
$$\text{where there exists an unbound } arg_{j,k} \in \overrightarrow{arg}_k$$
$$\text{with } k \in \{a_1, \ldots a_M\} \text{ and } 1 \leq j \leq n_k$$

The predicate h in the aforesaid rule R constitutes the head of the rule, where as the body of the rule R is comprised of the predicates a_1, ... , a_M. There must be at least one argument $arg_{j,k}$ which is unbounded, i.e. not used by any other predicate in this rule. The determination of argument $arg_{j,k}$ in a logical rule is simply checking all variables for their occurrences in the head and the remaining body predicates. If there is no match for an argument, the condition for a projection rule is fulfilled.

5.2.1 Provenance extension

The provenance rule follows the construction of the production rule. Please note that if there is a predicate h in the answer set for the head of the projection rule, there may be multiple instances of the provenance predicate $hProv$ - one for each instance of argument arg_j^k.

$$hProv(\overrightarrow{arg}_h, rule\_id, a_1, \overrightarrow{arg}_{a_1}, \ldots, a_M, \overrightarrow{arg}_{a_M})$$
$$:- a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}).$$

5.2.2 Example

As an example, for the aforesaid rule R, listing 2 line 1 shows an example of a projection rule taken from our scenario. The variables R and N are unbounded in this example. Listing 2, line 4 shows the explicit provenance rule related to the projection rule. Please be aware that the provenance rules for projection and logical rules do not deviate with regard to the provenance rule, but they potentially deviate in the number of instances in the answer set and therefore in the provenance graph.

Listing 2: Example of a projection rule with explicit provenance.

```
1 has_support(LOC) :- support(RISC,LOC,N).
2
3 % explicit provenance
4 has_supportProv(LOC, p11, support, RISC, LOC, N
      ) :- support(RISC,LOC,N).
```

5.2.3 Provenance Graph

Examples of the provenance predicates as derived from the use case are given below resulting in the provenance graph as depicted in Fig. 3. The projection processing element differs from the logical rule processing element descried in Sec. 5.1.3 by having a single input only. Here the input-output ratio of the projection processing element means that each tuple in the input view is consumed into a single tuple in the output view. The union processing element deviates

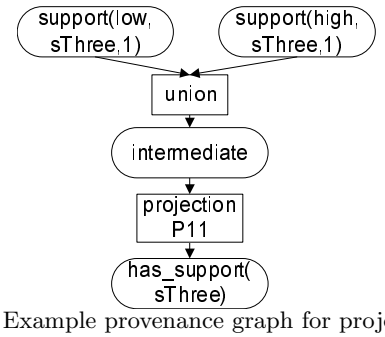

Figure 3: Example provenance graph for projection rule

from a logical rule processing element since it has an input-output ratio of one to one, thus every tuple in an input view is directly transformed into a tuple in the output view without correlating it with tuples from the remaining input views. As a consequence, the union in Fig. 3 introduces two tuples in the intermediate view, which are then projected into a single tuple in the has_support predicate.

has_supportProv(sThree,p11,support,low,sThree,1)
has_supportProv(sThree,p11,support,high,sThree,1)

5.3 Choice Rule with Constraints

A logic program might also include choice rules where the non deterministic choice in the head of the rule generates the search space and associated constraints reduce the search space again. The structure of a choice rule is given in Table 1 which is a short-hand notation of the following:

$$l\{h(\overrightarrow{arg}_h^1),\ldots,h(\overrightarrow{arg}_h^n)\}u :- a_1(\overrightarrow{arg}_{a_1}),\ldots a_M(\overrightarrow{arg}_{a_M}).$$

where $l, u \in \mathbb{N}$ and the choice rule says that the solver must choose at least l predicates from the set of $h(\overrightarrow{arg}_h^1),\ldots,h(\overrightarrow{arg}_h^n)$ predicates but not more than u predicates. A choice rule is often associated with constraints. Constraints are used to minimize the search space by excluding certain combinations of predicates that do not satisfy a particular condition. The basic structure of a constraint in relation with a choice rule is given below:

$$:- h(\overrightarrow{arg}_h^i),\ b_1(\overrightarrow{arg}_{b_1}),\ldots b_M(\overrightarrow{arg}_{b_M}), not\ b_{M+1}(\overrightarrow{arg}_{b_{M+1}}).$$

It is important to understand that the core information is the $h(\overrightarrow{arg}_i)$ predicate and the negated predicate b_{M+1}. Since the body of a constraint must always evaluate to false, it actually means that the negated predicate b_{M+1} must always evaluate to true if the predicate $h(\overrightarrow{arg}_h^i)$ evaluates to true. This information must be added to the provenance graph.

5.3.1 Provenance extension

To document explicit provenance for choice rules with constraints, we add one explicit provenance rule for each predicate in the head of the corresponding choice rule. The formulation of the provenance rule $hProv$ follows the same procedure as discussed in Sec. 5.1.3. However, the body of the provenance rule contains now an additional predicate $h(\overrightarrow{arg}_h^i)$ to ensure that the right instances of the provenance predicate $hProv$ are available in the answer set. To represent the constraints an additional provenance rule is added $hProvConstr$ which contains as arguments the arguments of the chosen head predicate $h(\overrightarrow{arg}_h^i)$ as well as the name and the arguments of the negated predicate b_{M+1}. The body of this additional rule consists out of the head predicate of the choice rule $h(\overrightarrow{arg}_h^i)$ and the negated predicate b_{M+1}. Please be aware that there can be multiple instances of the $hProfConstr$ provenance rules for a single head predicate $h(\overrightarrow{arg}_h^i)$. The formulation of the provenance rule for the aforementioned choice rule with constraints is given below where $i \in [1,n]$:

$$hProv(\overrightarrow{arg}_h^i, rule\_id, a_1, \overrightarrow{arg}_{a_1}, ..., a_M, \overrightarrow{arg}_{a_M})$$
$$:- h(\overrightarrow{arg}_h^i), a_1(\overrightarrow{arg}_{a_1}),\ldots,a_M(\overrightarrow{arg}_{a_M}).$$
$$hProvConstr(\overrightarrow{arg}_h^i, rule\_id, b_{M+1}, \overrightarrow{arg}_{b_{M+1}})$$
$$:- h(\overrightarrow{arg}_h^i), b_{M+1}(\overrightarrow{arg}_{b_{M+1}}).$$

5.3.2 Example

Listing 3, lines 1 and 2 show a choice rule with constraint from the use case. Please be aware that the enumeration of all choices is done by providing the domain of variable $RISK$ which is given by the predicate $accessibilityrisk(RISK)$. For the provenance rules in line 5 and 6 of Listing 3, the enumeration of all possible arguments of *choicerisk* can also be avoided by using variables $RISK$ and LOC. Please note that the answer set may contain multiple instances of the $choiceriskProvConstr$ constraint - one for each constraint related to the choice rule.

Listing 3: Example of a choice rule with constraints with it's corresponding provenance rule

```
1 1{choicerisk(RISK,LOC): accessibilityrisk(RISK)
      }1:- roadsegments(LOC), has_support(LOC).
2 :-choicerisk(RISK,LOC),not aux(RISK,LOC).
3
4 % explicit provenance
5 choiceriskProv(RISK, LOC, p13,
      accessibilityrisk, RISK, roadsegments, LOC,
       has_support, LOC) :- choicerisk(RISK,LOC),
       roadsegments(LOC), has_support(LOC).
6 choiceriskProvConstr(RISK, LOC, p13, aux, RISK,
       LOC) :- choicerisk(RISK,LOC), aux(RISK,LOC
      ).
```

5.3.3 Provenance Graph

Examples of the provenance predicates as derived from the use case are given below resulting the provenance graph depicted in Fig. 4. The logical rule processing element has been explained in Sec. 5.1.3. The constraint processing element has similar to the logical rule processing element a many to one input-output ratio and therefore they provide actually the same semantics. We use anyway different names in the provenance graph to increase the usability for the user, thus, ease the mapping back to the source code.

choiceriskProv(low,sThree,p13, accessibilityrisk,low,
  roadsegments,sThree, has_support,sThree).
choiceriskProvConstr(low,sThree,p13,aux,low,sThree).

5.4 Logic Rule with Function

In logical programming languages, there are basic built-in support for operations on data types like e.g. the numerical

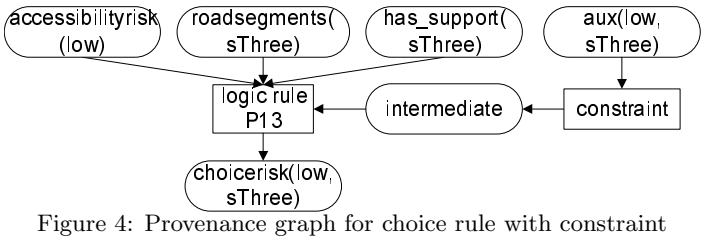

Figure 4: Provenance graph for choice rule with constraint

functions $\#count$, $\#min$, $\#max$ etc. A logic rule, R_F, using built-in functions has the following form:

$$h(\overrightarrow{arg}_h,CMP) :- a_1(\overrightarrow{arg}_{a_1}),\ldots,a_M(\overrightarrow{arg}_{a_M}),\\ CMP = \#function\{a_{M+1}(\overrightarrow{arg}_{a_{M+1}}),\ldots,a_N(\overrightarrow{arg}_{a_N})\}.$$

The difference between a basic logic rule, R (see Sec. 5.1) and a logical rule with a function, R_F is that in R_F, there is one extra argument in the head of the rule, i.e. CMP, which value has been calculated using $\#function$ over a set of predicates $a_{M+1},\ldots,a_N$.

5.4.1 Provenance extension

R_F is executed in two steps – first applying $\#function$ over the set of predicates $a_{M+1},\ldots a_N$, and second executing the entire logical rule. Since in the first step a set of predicates is used, a representation of the set as arguments in the provenance predicate results in an arbitrary number of arguments. Therefore, the provenance rule is split into two pieces. The first one addresses the entire logical rule, R_FProv, which is formulated following the same procedure discussed in Sec. 5.1.1. However, the head of the R_FProv does not contain the predicates used by the function, i.e. $a_{M+1},\ldots,a_N$, as it's arguments. The structure of the rule R_FProv is given below:

$$hProv(\overrightarrow{arg}_h,CMP,rule\_id,a_1,\overrightarrow{arg}_{a_1},\ldots,a_M,\overrightarrow{arg}_{a_M})\\ :- a_1(\overrightarrow{arg}_{a_1}),\ldots,a_M(\overrightarrow{arg}_{a_M}),\\ CMP = \#function\{a_{M+1}(\overrightarrow{arg}_{a_{M+1}})\}.$$

The second one addresses the set of predicates used to evaluate the function, where one predicate is created for each element of the set to capture explicit provenance information. The head predicate $hProv\#function$ of this provenance rule is named after the original rule appended with the key word '*Prov*' and as the suffix the name of function $\#function$. Adding the name of the function in the predicate encodes the type of function used and therefore gives an indication on how to translate this information into a provenance graph later on. The head predicate $hProv\#function$ of this rule also contains the predicates $a_{M+1},\ldots a_N$ used as input to the function. The body of the rule is formulated following the procedure discussed in Sec. 5.1.1. Furthermore, it contains one extra predicate which is the head of the original rule which ensures that the exact relationship is maintained between predicates once the rule is inferred. Finally, the body of the rule contains the predicates used inside the function, but does not apply the function to these predicates. The structure of the rule is given below:

$$hProv\#function(\overrightarrow{arg}_h,CMP,rule\_id,a_1,\overrightarrow{arg}_{a_1},\ldots,\\ a_M,\overrightarrow{arg}_{a_M},a_{M+1},\overrightarrow{arg}_{a_{M+1}},\ldots,a_N,\overrightarrow{arg}_{a_N})\\ :- h(\overrightarrow{arg}_h,CMP),a_1(\overrightarrow{arg}_{a_1}),\ldots\\ a_M(\overrightarrow{arg}_{a_M}),a_{M+1}(\overrightarrow{arg}_{a_{M+1}}),\ldots,a_N(\overrightarrow{arg}_{a_N}).$$

5.4.2 Example

Listing 4, line 1 shows a logic rule with the numerical function #count taken from our scenario. Based on the procedure to formulate provenance rules discussed above, Listing 4, lines 4 and 5 show the extended program containing the provenance rules. Please note that the answer set may contain multiple instances of the *supportProvCount* predicate - one for each predicate used for evaluating the function #count.

Listing 4: Example of a logic rule with aggregation and it's corresponding provenance rules

```
1 support(RISK, LOCATION, N) :- accessibilityrisk
      (RISK), roadsegments(LOCATION), N=#count{
      riskvalue(STREAMTYPE, RISK, LOCATION):
      streamtype(STREAMTYPE)}, N>0.
2
3 % explicit provenance
4 supportProv(RISK, LOCATION, N, p10,
      accessibilityrisk, RISK, roadsegments,
      LOCATION) :- accessibilityrisk(RISK),
      roadsegments(LOCATION), N=#count{riskvalue(
      STREAMTYPE, RISK, LOCATION): streamtype(
      STREAMTYPE)}, N>0.
5 supportProvCount(RISK, LOCATION, N, p10,
      accessibilityrisk, RISK, roadsegments,
      LOCATION, riskvalue, STREAMTYPE, RISK,
      LOCATION, streamtype, STREAMTYPE) :-
      support(RISK, LOCATION, N),
      accessibilityrisk(RISK), roadsegments(
      LOCATION), riskvalue(STREAMTYPE, RISK,
      LOCATION), streamtype(STREAMTYPE).
```

5.4.3 Provenance Graph

Examples of the provenance predicates as derived from the use case are given below resulting in the provenance graph depicted in Fig 5. The logical rule processing element has been explained in Sec. 5.1.3. The graph consists of using all related predicates for the evaluation of the function as input for the count processing element, which counts the number of tuples accessible in the input view and outputs this number in the output view, i.e., the intermediate view. The processing element uses a many to one input-output ratio. The intermediate view in addition with other predicates in the body of the logical rule are used as input for the logical rule processing element resulting in the intermediate2 output view. The intermediate2 view contains now implicitly the counted value, but it is not explicated in the name of a view. To explicate this information, the content of the view must be analyzed by the select processing element. This is the first processing element, which actually considers the value of the tuples in a view. Therefore we characterize the select processing element as a variable mapping processing element, while all others have been classified as constant mapping processing elements. While the latter one allow easy inference, the variable mapping processing elements require a special inference per processing element,

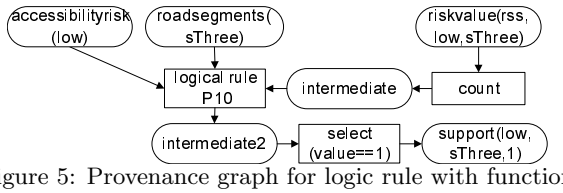

Figure 5: Provenance graph for logic rule with function

thus is less generic. The output view of the select predicate explicates the count result again in the name of the view, thus corresponds to the head predicate of the logical rule with functions.

```
supportProv(low,sThree,1,p10,
    accessibilityrisk,low,roadsegments,sThree).
supportProvCount(low,sThree,1,p10,
    riskvalue,rss,low,sThree).
```

6. INFERRED PROVENANCE

An alternative approach to obtain provenance information is to infer this information rather than explicating it as discussed in Sec. 5. The inference based approach consists out of a static code analysis part producing a workflow provenance graph, which is pruned on being requested to infer the fine-grained provenance information. As discussed in Sec. 1, workflow provenance explicates the relationship among the operations within a scientific model. In a logic program, it is used to capture the relationship among the predicates constituting the complete search space. Fine-grained data provenance documents the relationship among input data products, output data products and the associated operations in a model. In a logic program, fine-grained data provenance represents the *causality* between the predicates through the logic rules for a particular answer in a answer set.

To realize the inference mechanism, first, workflow provenance graph of a logic program has to be built. Afterward, based on the provenance finding request for a particular answer in an answer set, we can infer fine-grained provenance graph facilitating the existing workflow provenance graph of that logic program.

6.1 Workflow Provenance

The inference of the workflow provenance is based on a static analysis of the logical program. The oClingo tool explicates the search space for streaming logical programs, called grounding, where all possible instances of logical rules and constraints are explicated. The static inferences in the logical program are optimized and therefore are no longer explicated.

The inference of the workflow provenance analyzes the grounding of the logical program, clusters the rules according to the four classes mentioned in Sec. 5: logical rules, projection rules, choice rules with constraints, and logical rules with function. Based on these clusters a workflow provenance graph can be inferred consisting out of the basic building blocks described in Sec. 5. The mapping of the cluster to the provenance graphs is not repeated in this section.

In this sub section we discuss on how these cluster can be derived from the information provided by the grounding of the logical program. The grounding of the logical program is the explication of the search space, which can not be limited due to the usage of volatile predicates, i.e., the handling of streaming information. In the grounding logical rules and constraints containing non volatile predicates are not represented but only the result of the logical rule since it is static. However, the volatile part of logical rules and constraints remain as rules in the grounding. In the following we cluster rules and constraints contained in the grounding, thus logical rules and constraints with volatile predicates.

6.1.1 Logical Rule

The basic building block of a logical program is the logical rule which has been described in detail in Sec. 5.1. Listing 5 provides an example of logical rules contained in the grounding of the logical program which is related to the example given in Sec. 5.1.3.

Listing 5: An example of the grounding of a logical rule

```
1 riskvalue(rss,high,sThree):-rss(accident,sThree
      ,2,4).
2 riskvalue(rss,high,sThree):-rss(brokencar,
      sThree,2,4).
3 riskvalue(rss,high,sThree):-rss(accident,sThree
      ,2,5).
4 riskvalue(rss,high,sThree):-rss(brokencar,
      sThree,2,5).
5 riskvalue(rss,low,sThree):-not riskvalue(rss,
      high,sThree).
```

Each of these rules contained in the listing is transformed into a provenance graph. The provenance graph for line 2 is comparable to the left explicit provenance graph in Fig 2, except that the static predicate $negative(brokencar)$ is not contained in the grounding and therefore not contained in the inferred provenance graph (see Fig 6).

The right side of the explicit provenance graph in Fig 2 contains as input predicate only a static predicate. However, the grounding in line 5 of Listing 5 contains the volatile expression of not having a high riskvalue. Negated predicates are excluded in the explicit provenance but are modeled in the inferred provenance as a 'not' processing element. This processing element has variable mapping and therefore requires for the inference special consideration. The resulting provenance graph is depicted on the right hand side of Fig 6.

6.1.2 Projection Rule

The second class of rules are projection rules as discussed in detail in Sec. 5.2. Listing 6 provides an example of reduction rules contained in the grounding of the logical program which is related to the example given in Sec. 5.2.3.

Listing 6: An example of the grounding of a reduction rule

```
1 has_support(sThree):-support(low,sThree,3).
2 has_support(sThree):-support(low,sThree,2).
3 has_support(sThree):-support(low,sThree,1).
4 has_support(sThree):-support(high,sThree,3).
5 has_support(sThree):-support(high,sThree,2).
6 has_support(sThree):-support(high,sThree,1).
```

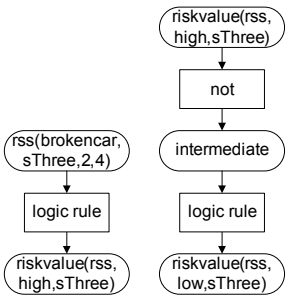

Figure 6: Example inferred provenance graph for logical rule

Line 3 and 6 in Listing 6 describe the information visualized in the explicit provenance graph in Fig 3. The actual inferred provenance graph extends the explicit one by adding more input views, in particular, one view per predicate in the body of the logical rules in Listing 6.

6.1.3 Choice Rule with Constraints

Another class of rules are logical rules with a non deterministic choice in the head of the rule and associated constraints to reduce the generated space of options. Choice rules have been discussed in detail in Sec. 5.3. Listing 7 provides an example of choice rules with related constraints as they are contained in the grounding of the logical program. The examples are related to the example given in Sec. 5.3.3. In particular, a single logical rule is associated with a set of constraints, where the negated predicates in the constraints are added to the provenance graph via a constraint processing element.

Listing 7: An example of the grounding of a choice rule with constraints

```
1 1{choicerisk(high,sThree),choicerisk(low,sThree
      )}1:-has_support(sThree).
2 :-choicerisk(low,sThree),not aux(low,sThree).
3 :-choicerisk(high,sThree),not aux(high,sThree).
```

The explicit provenance graph depicted in Fig 4 is very similar to the inferred provenance graph for line 1 and 3 of Listing 7. The inferred provenance graph does not contain the static predicates *accessibilityrisk* and *roadsegments*.

6.1.4 Logical Rule with Function

The last class are logical rule with functions which has been described in detail in Sec. 5.4. Listing 8 provides an example of logical rules contained in the grounding of the logical program which is related to the example given in Sec. 5.4.3. In the grounding functions are represented based on their semantics, thus no generic statement can be made. For the #count function as used in the example, the aggregates are represented as choices without additional constraints except a lower and upper bound which corresponds to the result of the count argument in the head predicate of the rule. In particular, there is one logical rule per count value. The lack of additional constraints can be explained by the oClingo solver applying a specialized macro on this part of the grounding to prune the search space. However, this special behavior is not represented in the grounding.

To detect the count function in the grounding during the static analysis, it is necessary to know that the support predicate has as the third argument a numerical value derived by a count operation. This information can be either configured or derived from the original logical program. In our current implementation we have used a configuration. The inferred provenance graph equals the explicit provenance graph as depicted in Fig 5.

Listing 8: An example of the grounding of a logical rule with function

```
1 support(low,sThree,1):-1{riskvalue(weather,low,
      sThree),riskvalue(rss,low,sThree),riskvalue
      (tweet,low,sThree)}1.
2 support(low,sThree,2):-2{riskvalue(weather,low,
      sThree),riskvalue(rss,low,sThree),riskvalue
      (tweet,low,sThree)}2.
3 support(low,sThree,3):-3{riskvalue(weather,low,
      sThree),riskvalue(rss,low,sThree),riskvalue
      (tweet,low,sThree)}3.
```

6.2 Fine-grained Provenance Inference

Fine-grained provenance inference facilitates the generated workflow provenance of a given model, i.e. a logic program, and the available input and output data tuples, i.e. input and output predicates in the answer set. Previously, we proposed several methods to infer fine-grained provenance data [18, 17, 15]. In [18], we proposed an approach that can infer fine-grained provenance in an environment with a constant delay for the processing. Later, we extended this approach and proposed a probabilistic inference mechanism that can address variable delays in processing in [17]. However, this approach can work for a single processing step. To address this issue, recently we have proposed probabilistic inference for multiple processing steps, i.e. a complete workflow [15]. In this paper, we apply this technique on the scenario described in Sec. 4 since there are multiple rules in a logic program which are transformed into a set of processing steps.

In this sub section, we discuss briefly the working mechanism of the multi-step inference approach. The inference technique has two phases: i)*backward computation* and ii)*forward computation*. To retrieve actual data tuples, the algorithm requires to have only the output data products, i.e. *choicerisk* predicates, and the input data products such as *input_weather*, *input_tweet* and *input_rss* predicates persistent. The other views generated by the intermediate processing steps are considered as transient views. We create a SQLite[1] database that contains tables for each persistent view found in the workflow provenance graph and then populate these tables during the execution of the logic program.

6.2.1 Backward Computation

This phase is executed once a user requests provenance of an answer/predicate from the available answer sets. This predicate is known as chosen predicate/tuple. The timestamp associated with this predicate is referred to as *reference point*. Based on this *reference point*, the inference mechanism calculates the window boundary which is fixed over all

[1] http://www.sqlite.org/

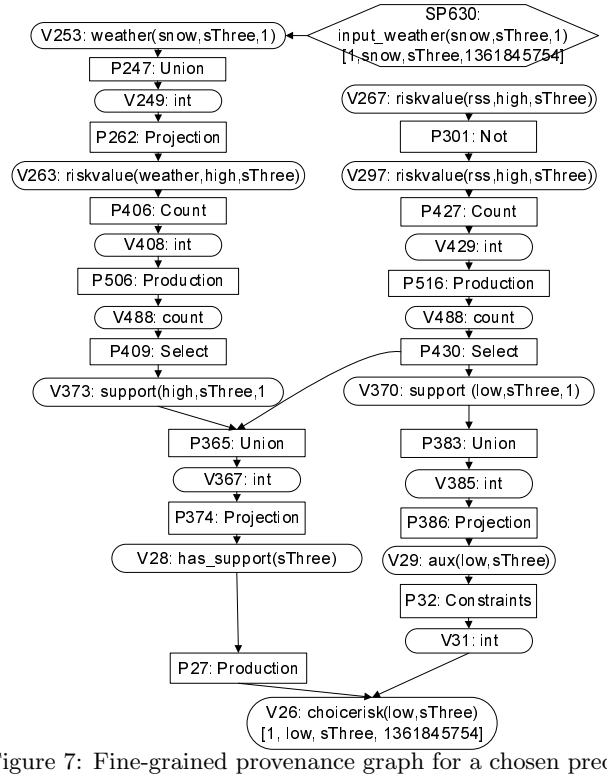

Figure 7: Fine-grained provenance graph for a chosen predicate

the input views. The equations to calculate the *upper bound* and the *lower bound* of a window boundary are given below:

$$upperBound = reference\ point \tag{8.1}$$

$$lowerBound = reference\ point - window\ size \tag{8.2}$$

These formulas are slightly adjusted from the original formulas what have been proposed in [15] because of the fact that, the entire workflow is executed at once and therefore, there is no accumulated execution delay in this case.

6.2.2 Forward Computation

After defining the window boundary, the inference algorithm reconstructs the windows in the forward computation phase. In this phase, the inference mechanism establishes the relationship between the data products in the input views to the subsequent view until it reaches the chosen predicate in the output view. It also fetches the stored tuples corresponding to the input views only from the database. While executing this phase, a few of the branches which are not directing towards the chosen predicate are pruned from the graph.

6.2.3 Post-processing

It is possible that a group of *selection* operators would appear in the provenance graph indicating the different possibilities of the `count` value in the *support* predicates as discussed in Sec. 5.4. Therefore, in the post processing phase, the inference mechanism determines exactly which of the *selection* operation satisfies the condition and thus contributing to the generation of the chosen predicate. The other branches having non-satisfiable *selection* operators are pruned from the graph. Furthermore, there exists an assumption in the logic program indicating that if there is no *riskvalue* with *high* for streamtype *rss*, the program assumes that there is a *riskvalue* with *low* for streamtype *rss* contributing to the output predicate. It is realized via *not* construct. This *not* construct is also explicated in the fine-grained provenance graph by checking the existence of any *not* processing element in the provenance trace generated during the backward phase.

Fig. 7 shows an example of a fine-grained provenance graph for the chosen predicate *choicerisk(low,sThree)*. The provenance graph explains the complete derivation history for the chosen output predicate. The source processing elements shows the actual input tuple which contributes to produce the chosen predicate represented by the view node $V_2$6. It also shows that there exists no input predicate for *riskvalue* with *high* for streamtype *rss* and hence the program assumed the existence of the predicate *riskvalue(rss,low,sThree)*.

7. EVALUATION

7.1 Evaluation Criteria

The aim of the evaluation is to see whether the fine-grained provenance graph is useful for debugging logic programs. Further the explicit and inferred provenance approach (see Sec. 5 and 6.1 respectively) are compared using the following evaluation criteria: i) accuracy of the inferred provenance graph, ii) execution time of the logic program extended with provenance information, and iii) storage space consumption. We execute the logic program designed for our scenario described in Sec. 4 for 100, 200, 300 and 500 steps with randomly generated synthesized data sets.

7.2 Accuracy of Provenance Graphs

The accuracy of the inferred provenance graph is measured against the explicit provenance graph which represents the ground truth. The explicit provenance graph contains the base facts, that is constants based on the provenance graph model, which are not contained in the inferred provenance graph since the grounding mechanism does not explicate any base facts. Constants could be added to the inferred provenance based on a static analysis of the logical program. Due to lack of time, we have not done this step. Therefore, we do not consider constants for determining the accuracy of the inferred provenance graph.

The accuracy of the inferred provenance graph for a single output predicate acc_i is considered one if the set of source processing elements of the explicit and the inferred provenance graph are equivalent. The average accuracy of the experiment is the sum of the accuracy of a single output predicate over the number of output predicates investigated. Thus, $average\ accuracy = (\frac{\sum_{i=1}^{n} acc_i}{n} \times 100)\%$.

In all test cases, the output predicates of the inferred provenance graph has an accuracy of 100%, thus all inferred provenance graphs match exactly the corresponding explicit provenance graph. This accuracy value was expected since there are no options for errors during the inference due to the fact that the logical program is executed at once and therefore all predicates are available at the same time. Thus, no processing delay can be accumulated to cause a subsequent processing element to shift window boundaries producing errors.

Table 2: Comparison of Execution Time (in seconds)

Number of steps	Program without provenance rules (Inference)	Program with provenance rules (Explicit)	Ratio
100	18.2	58.1	1:3.2
200	57.4	166.9	1:2.9
300	115.1	329.6	1:2.9
500	280.3	out of memory	-

Table 3: Comparison of Storage space consumption (in KB)

Number of steps	Inferred Provenance	Explicit Provenance	Ratio
100	14	181	1:13
200	23	354	1:15.3
300	29	533	1:18
500	57	out of memory	-

7.3 Execution Time

Execution time of a logic program depends on the number and complexity of the rules, thus the complexity of the search space. The inferred provenance mechanism does not require to include extra provenance rules like the explicit provenance approach. Hence, the later incurs overhead in terms of execution time due to the augmentation of extra provenance rules. Table 2 reflects this statement showing the execution times of the logic program with and without explicit provenance rules, which are used by explicit technique and inference technique respectively.

For the test cases with 100, 200 and 300 logical time steps, the execution time of the program with the provenance rules takes around 3 times more time than the one without the provenance rules. For the last test case with 500 steps, only the logic program without the provenance rules can finish the execution. Thus, the execution of the logic program without provenance rules as used by the proposed inference mechanism is much faster and for a higher number of steps it is simply not possible to document explicit provenance information.

7.4 Storage Space Consumption

The storage space consumption is measured by comparing the size of the SQLite databases that hold the predicates of the answer set of the logical program in form of relational data tuples. Both approaches require to materialize all input and output predicates. Furthermore, the explicit provenance approach materializes all provenance predicates as tuples in the database. As a consequence, the storage overhead of the explicit approach is dependent on the required provenance predicates.Table 3 shows the disk space consumed by these two methods.

From Table 3, it is evident that the explicit provenance has several magnitudes of storage overhead compared to the inferred provenance. The explicit method takes at least 13 times more storage than the proposed inferred provenance approach. The ratio comparing storage consumed by these two methods keeps increasing with the increase in number of steps because of the accumulated data which is the nature of stream data processing.

7.5 Applicability to Debugging

The opinions presented in this section are provided by two ASP domain experts. We are aware that this is not a representative usability evaluation, but it gives an indication on the applicability of provenance graphs for debugging. In future work, we will conduct an extensive usability study.

The workflow and fine-grained provenance graphs provide useful insights to both the logic programmer and the domain expert. The workflow provenance graph captures the way data is related in the search space and facilitates the understanding of these connections. It also helps identifying how constraints should be added or removed to reduce or expand the set of correct solutions. The fine-grained provenance graph is a subset of the workflow provenance graph that explains the complete derivation history of a chosen fact in an answer set. It allows to verify the correctness of the rules for modeling a particular domain. All these insights facilitate early-stage *instance-driven* debugging of an ASP program.

The presented approach addresses many core elements of the logical programs, however, Negation as Failure (NAF) is not completely addressed in explicit provenance graphs. Handling NAF in logical rules for explicit provenance can be done by introducing a new provenance predicate $hProvNAF$ documenting the NAF provenance information. Based on this additional information, the provenance graph can be extended. We have not done this due to a lack of time.

Fine-grained provenance provides the complete derivation history of an out-coming fact. Therefore, this graph can be useful to achieve reproducible results at the instance-level. Another interesting possibility is to make the provenance graphs more compact by grouping several steps together. Currently, we group any intermediate step with its successor to have a more compact representation of the graph as well as to provide transparency of the transformation from logical rules to a set of nodes in the graph. However, it would be worth investigating options to customize this compacting process based on the semantics of the program.

8. CONCLUSION AND FUTURE WORK

In this paper, we introduce the way of collecting provenance information in the context of a logic program. Further, we discuss two approaches - explicit and inference method and their pros and cons. Our evaluation shows that the provenance inference approach is better suited to extract provenance information of a logic program. The use we make of data provenance in this paper should be considered as an additional tool for debugging ASP program in the initial modeling phase, when domain experts and ASP programmers sit together to formulate a problem description. The mutual understanding of how and why certain inputs generate a certain output can produce better formulation faster, in this crucial phase. However, we acknowledge the benefits of ongoing work on ASP debugging from a semantic perspective and the added value of IDE tools, and we will be exploring how data provenance can be embedded in those for the next phases of ASP development.

9. REFERENCES

[1] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[2] M. Brain et al. Debugging ASP programs by means of asp. In *LPNMR*, pages 31–43, 2007.

[3] M. Brain and M. D. Vos. Debugging logic programs under the answer set semantics. In *Answer Set Programming*, 2005.

[4] P. Buneman and W. C. Tan. Provenance in databases. In *SIGMOD*, pages 1171–1173, 2007. ACM.

[5] T. Do, S. Loke, and F. Liu. Answer set programming for stream reasoning. *Advances in Artificial Intelligence*, pages 104–109, 2011.

[6] E. Erdem, Y. Erdem, H. Erdogan, and U. Öztok. Finding answers and generating explanations for complex biomedical queries. In *AAAI*, 2011.

[7] O. Febbraro, K. Reale, and F. Ricca. Aspide: Integrated development environment for answer set programming. In *LPNMR*, pages 317–330, 2011.

[8] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Stream reasoning with answer set programming: Extended version. *Unpublished draft.*, 2012.

[9] M. Gebser et al. Stream reasoning with answer set programming: Preliminary report. In *Proc. Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR 2012)*, 2012.

[10] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answerset solving in practice. 2012.

[11] M. Gebser, J. Pührer, T. Schaub, and H. Tompits. A meta-programming technique for debugging answer-set programs. In *AAAI*, pages 448–453, 2008.

[12] M. Gebser, T. Schaub, S. Thiele, and P. Veber. Detecting inconsistencies in large biological networks with answer set programming. *TPLP*, 11(2-3):323–360, 2011.

[13] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of Int'l Conf. on Logic programming*, volume 161, 1988.

[14] P. Groth et al. Automatic metadata annotation through reconstructing provenance. In *Semantic Web in Provenance Management*, CEUR Workshop, 2012.

[15] M. R. Huq et al.. Fine-grained provenance inference for a large processing chain with non-materialized intermediate views. In *SSDBM*, LNCS volume 7338, pages 397–405. Springer, 2012.

[16] M. R. Huq et al. From scripts towards provenance inference. In *IEEE Conference on eScience*, pages 1–8, 2012.

[17] M. R. Huq, P. M. G. Apers, and A. Wombacher. Probabilistic inference of fine-grained data provenance. In *DEXA (1)*, LNCS volume 7338, pages 296–310, 2012.

[18] M. R. Huq, A. Wombacher, and P. M. G. Apers. Inferring fine-grained data provenance in stream data processing: Reduced storage cost, high accuracy. In *DEXA (2)*, LNCS volume 6861, pages 118–127, 2011.

[19] C. Kloimüllner et al. Kara: A system for visualising and visual editing of interpretations for answer-set programs. *CoRR*, abs/1109.4095, 2011.

[20] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1):39–54, 2002.

[21] B. Ludascher et al. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[22] A. Mileo, D. Merico, and R. Bisiani. Reasoning support for risk prediction and prevention in independent living. *TPLP*, 11(2-3):361–395, 2011.

[23] A. Mileo, T. Schaub, D. Merico, and R. Bisiani. Knowledge-based multi-criteria optimization to support indoor positioning. *Ann. Math. Artif. Intell.*, 62(3-4):345–370, 2011.

[24] S. Miles. Automatically adapting source code to document provenance. In *Provenance and Annotation of Data and Processes*, LNCS volume 6378, pages 102–110. 2010.

[25] V. Novelli et al. Log-ideah: Asp for architectonic asset preservation. In *ICLP (Technical Communications)*, pages 393–403, 2012.

[26] J. Oetsch, J. Pührer, and H. Tompits. Catching the ouroboros: On debugging non-ground answer-set programs. *TPLP*, 10(4-6):513–529, 2010.

[27] J. Oetsch, J. Pührer, and H. Tompits. Stepping through an answer-set program. In *LPNMR*, pages 134–147, 2011.

[28] T. Oinn et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, June 2004.

[29] U. Park and J. Heidemann. Provenance in sensornet republishing. *Provenance and Annotation of Data and Processes*, pages 280–292, 2008.

[30] E. Pontelli, T. C. Son, and O. El-Khatib. Justifications for logic programs under answer set semantics. *TPLP*, 9(1):1–56, 2009.

[31] C. F. Reilly et al. Instrumenting a logic programming language to gather provenance from an information extraction application. In *WWW '12 Companion*, pages 589–590, 2012. ACM.

[32] A. Sarma et al. LIVE: A Lineage-Supported Versioned DBMS. In *SSDBM*, pages 416–433, 2010.

[33] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.

[34] Y. L. Simmhan et al. Karma2: Provenance management for data driven workflows. *International Journal of Web Services Research*, 5:1–23, 2008.