WebLab PROV: Computing fine-grained provenance links for XML artifacts

Bernd Amann  
LIP6 - UPMC, Paris

Camelia Constantin  
LIP6 - UPMC, Paris

Clément Caron  
EADS-Cassidian, Val de Reuil

Patrick Giroux  
EADS-Cassidian, Val de Reuil

ABSTRACT
We present a new provenance model for generating fine-grained data and service dependencies within XML data processing workflows. Our approach follows the widely used black box transformation semantics [15] in which service components produce new outputs from their inputs (without transformation). The heart of the model are data dependency rules which are evaluated on XML documents assembling all data produced by some workflow execution (similar to nested collections [5]). Dependency rules are defined in XPath extended with variables and can directly be compiled into XQuery expressions for generating provenance information in RDF-PROV [8]. We also present an implementation of our model, using the WebLab platform [19], showing step-by-step how our model works in a typical media mining use-case.

1. INTRODUCTION
The exponential growth of structured and unstructured information on the web has led to the development of powerful data and knowledge processing services for transforming this information into valuable data and knowledge. The WebLab platform [19] aims to provide an open environment for integrating such services into complex media mining workflows. A key feature of the WebLab environment is a domain specific web service ontology defining the function and interfaces of different classes of media mining services. This ontology and the adoption of web standards like WSDL-/SOAP, XML/XQuery and RDF/SPARQL makes it possible to build generic media mining workflows which can be instantiated by integrating different software components into the platform.

Capturing and analyzing the quality and validity of data and knowledge produced by media mining workflows is a challenging task and requires access to fine-grained provenance information about workflow executions. We argue that standard techniques for generating provenance information are insufficient to obtain this kind of fine-grained provenance dependencies in open web service composition environments like WebLab.

We can essentially distinguish between two main approaches for generating provenance information [14]. Database provenance is based on a “white-boxes” approach which consists in inferring fine-grained provenance links from high-level (declarative) component specifications [3] or database queries [10, 25]. Workflow provenance is based on a “black-boxes” approach where dependency relationships are declared at run-time or generated by implicit assumptions concerning dependency patterns. In this setting, generated provenance links are generally more coarse-grained. We show in this article how it is possible to obtain fine-grained provenance links between XML data by extending “black-boxes” workflows with explicit data dependency mappings. These are schema mapping rules which have been found to be important for lineage tracing [15]. In this sense, our work is also similar to [9], that also uses a rule-based approach.

The contributions of this article are (1) a provenance model for artifact-based XML document workflows including, (2) a declarative provenance mapping language for defining explicit data dependencies, (3) evaluation strategies for inferring provenance links from final workflow results and (4) a first prototype implemented on top of XML/XQuery and the PROV ontology [8] for building provenance graphs.

By these contributions we try to solve the issue of efficiently inferring provenance links in XML artifact-based workflow systems like WebLab. We propose a non invasive declarative provenance model that is generic enough to be applied to many XML data processing workflows, independently of their internal implementation. The possibility to translate provenance mapping rules into standard XQuery expressions also simplifies the implementation by taking advantage of existing query optimization techniques for generating data and service dependency links.

The article is organized as follows. Section 2 illustrates the problem and our solution by an extended example. Section 3 formally defines the WebLab data and provenance model. The main contribution about provenance generation is presented in Section 4. Possible extensions to our model are discussed in Section 5. In Section 6 we discuss a first implementation followed by the presentation of related work in Section 7. We conclude in Section 8.

2. MOTIVATING EXAMPLE
We illustrate our provenance framework in the context of
the WebLab platform [19] whose purpose is to build complex media mining (text, image, audio and video) workflows in a web service oriented architecture. All resources (documents, videos, images, texts, etc.) that are useful for the final user or for the processing tasks are identified XML fragments in a unique WebLab document. WebLab documents are generated by service workflows where each service call receives a WebLab document as input and extends it with new resources. This “append” semantics guarantees that the information contained in WebLab documents increases at each step. No data is ever deleted and the result of the workflow formation contained in WebLab documents increases at each step. No data is ever deleted and the result of the workflow execution is an XML document containing all information used or produced by a service. Without loss of generality, we assume that service call parameters are part of the document and can be extracted by the service orchestrator before each service call. In this sense, the WebLab model is in line with the notion of nested data collections [5].

Figure 1(a) shows a simple workflow composing three services Normaliser, LanguageExtractor and Translator. Figure 1(b) shows the evolution of some initial document $d_0$ during an execution of this workflow. Each $d_i$ corresponds to a different state of the document obtained by a service call at time instant $t_i$. The execution of service Normaliser at time instant $t_1$ transforms Native Content resource $d_0$ of $d_0$ and produces a normalized version, TextMediaUnit resource $d_1$, which is added as a child of the initial Resource node $d_0$. Service LanguageExtractor is called at time instant $t_2$ for identifying the language of the normalized text $d_1$ which is stored in a new resource $d_2$ of type Annotation. The final Translator service call at time instant $t_3$ uses TextMediaUnit $d_2$ to create a new TextMediaUnit resource $d_3$ containing its translation into some target language. Observe that the final document state $d_3$ contains the initial document state $d_0$ and all intermediate states.

The dashed arrows in figure 1(b) show the dependencies from new resources generated by a service call to resources used by each call. For example, service call (LanguageExtractor, $t_2$) used XML resource $d_1$ to produce resource $d_2$. We assume that resources are labeled by the service call that produced them (service dependency). These labels can be generated after each service call by the service itself or by the workflow engine. We can then define the provenance graph of some workflow execution as a labeled directed acyclic graph, connecting each labeled resource to the other labeled resources which have been used for its generation (data dependency). Figure 2 shows the provenance graph for our example. Table Source contains the resource labels and shows, for example, that resources $d_1$ and $d_2$ have been generated by service Normaliser at time instant $t_1$. Resource node $d_2$ which existed in the initial document state is annotated by service Source (for example the URL of a web page) and time instant $t_0$ (for example the acquisition date). This table is generated during the workflow execution and represents, together with the final XML document, the workflow execution trace.

Our goal is to automatically generate data dependency links between labeled resources from the workflow execution trace. These links are shown in table Provenance for all labeled resources in the previous workflow execution trace. We can see that resource $d_3$ depends on resource $d_0$ generated by service Normaliser and on resource $d_2$ which is a descendant of $d_1$ and has been generated by service LanguageExtractor.

![Figure 1: A simple workflow](image1.png)

![Figure 2: Provenance Graph](image2.png)
mation generated by service call (LanguageExtractor, t₂).

Observe that all explicit provenance links (⃝ → ⚫) between labeled resources can be propagated to all XML descendants and ancestors of ⚫ and all descendants of ⚫. For example, nodes ⚫, ⚫ and ⚫ “inherit” all provenance information from resource ⚫. Also, node ⚫ depends on ⚫, which is an ancestor of ⚫.

WebLab service orchestration considers web services as black boxes transforming a single XML document by adding their result fragments. Different approaches can be used for building provenance graphs depending on the available information (e.g. execution logs, workflow specifications) and the granularity of the provenance links (see related work section). In our case, we want to be able to infer fine-grained provenance links between identified XML fragments (resources) after the workflow execution, from the final document where resources are labeled by the service calls that produced them. We are confronted in this case with two issues. The first issue consists in soundly identifying in the final document the input and output resources for each service call, whereas the second issue consists in inferring all data dependency links between these resources.

In order to specify and generate fine-grained provenance links, we apply non-intrusive techniques without modifying the service definition. We propose to use a mapping-based approach [15] for identifying data dependencies between service input and output resources. We will define provenance mapping rules based on XPath patterns with variables to specify data dependency links between its input and output resources. For example, for documents generated by the workflow in Figure 1(a), a user might define the following provenance mappings shown in Figure 3.Mapping M₁,

\[
M₁ : \frac{/
Resource//\text{NativeContent}}{/
TextMediaUnit[@id]}
\]

\[
M₂ : \frac{/
TextMediaUnit[/$s := @id]/\text{TextContent}}{/
TextMediaUnit[/$s := @id]/\text{Annotation}[\text{Language}]}
\]

\[
M₃ : \frac{/
TextMediaUnit[\text{Annotation}]/\text{Language} = \text{\textquoteleft fr\textquoteright]}{/
TextMediaUnit[\text{Annotation}]/\text{Language} = \text{\textquoteleft en\textquoteright]}
\]

Figure 3: Example Provenance Mappings

tells that the first TextMediaUnit resource depends on all previously generated Native Content resources. Similarly, M₂ signifies that all Annotation resources containing an element Language depend on their siblings of type Text Content in the same TextMediaUnit resource identified by XML attribute id. Finally, rule M₃ generates data dependency links from TextMediaUnit resources in French to TextMediaUnit resources in English.

Provenance mappings are similar to schema mapping rules for relational databases [17] or XML documents[2]. Also, the work of [21] deduces data provenance from relational mappings in a distributed data exchange setting. However, there are some differences between our provenance generation problem and the general schema mapping problem:

- Our goal is not to define mappings between any kind of XML documents, but to infer causal provenance links between document fragments generated by a workflow execution.

- The semantics of dependency rules includes context-specific implicit logical and temporal constraints for inferring provenance links from the final document (see section 4):

  - The use of node creation timestamps permit to identify the set of resources which existed before a service call and obviously a resource created at timestamp d can not depend on a resource created at some timestamp d’ > d.

  - Starting from the workflow definition, we can exploit service orchestration constraints like service s is always executed before service s’, to eliminate provenance links from data produced by service s to data produced by s’.

3. DATA AND PROVENANCE MODEL

WebLab Documents.

The WebLab platform features XML documents which can be represented as node-labeled ordered ranked trees. XML trees τ are defined over the tree domain D which is built over an infinite domain of labeled tree nodes N. Trees can be compared by using the containment relationship ≤; we say that τ ≤ τ’ iff all nodes and structural relationships of τ are preserved in τ’. Equivalently, we can say that τ is obtained from τ’ by removing a bag of sub-trees δ = τ \ δ’. WebLab resource nodes are distinguished nodes r ∈ N which are uniquely identified by a URI (Uniform Resource Identifier) assigned by some partial function uri : N → URI.

**Definition 1 (WebLab Document).** A WebLab document is a couple d = (τ, uri) where τ is an XML tree and uri assigns a unique URI to a subset of nodes in τ including its root node.

Observe that uri might identify any subset of nodes in τ. Documents can also be compared, by extending the tree containment relationship: for two documents d₁ = (τ₁, uri₁) and d₂ = (τ₂, uri₂), we can say that d₁ ≤d₂ iff τ₁ ≤ τ₂ and for all common resource nodes n in τ ∩ τ’, either uri₁(n) = uri₂(n) (the same identifiers in both resources) or uri₁(n) is undefined. In other words, for d₁ to be contained in d₂, the uri₂ function should preserve the identifiers of all resources in d₁, i.e. it can add identifiers but it cannot modify nor delete them.

If d₁ ≤d₂, by definition, d₁ and d₂ share the same root resource and we consider that d₁ and d₂ are two different states of the same document. The difference d’ \ d always returns a bag of resources, i.e XML fragments with identified roots.

**Example 1.** Figure 4 shows four different document states rooted at the same resource node ⚫, that correspond to the document states shown in Figure 1(b). Resource nodes are shaded in the figure, the other nodes of the tree are unidentified. We can observe the inclusion relationships d₀ ≤uri d₁ ≤uri d₂ ≤uri d₃ and d₁ \ d₀ is a set of two XML fragments which are rooted at resources ⚫ and ⚫. Note that node ⚫ in d₀ has been “promoted” to a resource node ⚫ in document state d₁.

**Workflow executions.**

We consider in the following WebLab workflows where each service call takes a single WebLab resource d = (t, uri) as input and increases it with one or several new resources.
Let $S$ be a possibly infinite domain of services and $T$ be an infinite ordered domain of timestamps. Let $C = S \times T$ be the infinite domain of service calls. Each service call $c = (s, t)$ which is applied to a document $d$ returns a new document $d'$ such that $d \leq_{ur} d'$, as defined previously.

**Definition 2 (Workflow Execution).** A workflow execution is an alternating sequence $e = d_0.c_1.d_1.c_2...c_n.d_n$ where $d_0, d_1...d_n$ is a sequence of $n + 1$ documents (states) (called the data flow) such that $d_i \leq_{ur i} d_{i+1}$ for all $i \in [0, n-1]$ and $c_1, c_2...c_n$ is a sequence of $n$ service calls called the control flow of $e$.

Data flow $d_0.d_1...d_n$ is produced by the control flow $c_1.c_2...c_n$ applied on document $d_0$. We will denote by $in(c_i) = d_{i-1}$ the input document state of $c_i$ and by $out(c_i) = d_i/d_{i-1}$ the bag of resources generated by $c_i$. Observe that, for the sake of simplicity, we consider only sequential workflows in our model.

**Provenance Graphs.**

Each service call $c_i$ of the workflow uses as input a set of XML fragments in $d_{i-1}$ rooted at resource nodes $in(c_i)$ and produces as result a set of new XML fragments whose roots are resource nodes in $out(c_i)$. The provenance graph associated to a workflow execution materializes provenance dependencies from the results of service calls to their input resources. We can define a provenance graph associated to a workflow execution as follows:

**Definition 3. (Provenance Graph)** A provenance graph of a workflow execution $e = d_0.c_1.d_1...c_n.d_n$ is a labeled directed acyclic graph $G(e) = (R, E, \lambda)$ where $R \subseteq N$ is the set of resource nodes in the result document $d_n$, labeling function $\lambda : R \rightarrow C$ assigns to each resource node $\circ \in R$ the service call $c_i$ that produced it and $E \subseteq R \times R$ is a set of edges $(r, r')$ where resource $r$ has been generated by service call $\lambda(r)$ by using resource $r'$ in $d_{i-1}$.

**Example 2.** Document $d_3$ has been produced by the sequential execution of three service calls $c_1.c_2.c_3$ in Figure 1(a). The provenance graph is represented by the two tables in Figure 2. Table Source corresponds to the labeling function $\lambda$ and table Provenance contains all provenance links $E$.

Note that this provenance model is fine-grained since provenance graphs might contain links between any document fragments. Our goal is to enable the generation of this kind of links without inspecting or changing the web service code, similarly to the solution proposed in [9].

### 4. PROVENANCE MAPPING RULES

**XPath Patterns and Mapping Rules.**

XPath patterns are based on Core XPath [20] enriched with predicates using variables to collect and match attribute values. A similar approach was proposed by [6] for defining XML mappings. Compared to full XPath, Core XPath is restricted to the usage of child and descendant axes without functions. This subclass covers a large set of useful queries and reduces the polynomial complexity of full XPath to linear complexity.

**Definition 4 (XPath Patterns).** An XPath pattern is sequence of steps $step_1/.../step_n$ where $step_i$ is a Core XPath step

$$step_i = axis :: filter[predicate]^* [\alpha_i]^{0,1}$$

and $\alpha_i$ is an optional sequence of variable assignments

$$_{x_1} := \@a_1, ..., _{x_j} := \@a_j, ..., _{x_k} := \@a_k$$

Expressions $\@a_j$ denote XML attributes called binding attributes and $$_{x_i}$$ are binding variables satisfying the following conditions: (1) each $$_{x_j}$$ appears exactly once, but the same attribute $\@a_j$ might be assigned to different variables; (2) the filter node must contain all the attributes in $\alpha_i$ (this is equivalent to adding an implicit constraint $\@a_1 and... and \@a_n$ to $step_i$); (3) there exists an implicit assignment expression $$_{x} := \@id in the last step where $$_{x}$ is a new variable and $\@id$ is the identifier assigned by the urifunction to the node of the last step.

In the following we will use the notation $\varphi(\bar{x})$ for XPath patterns where tuple $\bar{x} = ([_{x_1}, ..., _{x_j}])$ denotes the set of binding variables of the pattern (the default variable for the final result set is not explicitly mentioned).

**Example 3. Four XPath patterns:**

$$\varphi_1(\$x) = //T[@id=\$id]/C$$

$$\varphi_2(\$x) = //T[@id=\$id]/C[@r=\$id]/T[@l=\$id]$$

$$\varphi_3(\$x) = //T[@id=\$id]/A[@l]$$

$$\varphi_4(\$x) = //T[@id=\$id]/A[@l]$$

Observe that $\varphi_2(\$x)$ is an equivalent rewriting of $\varphi_1(\$x)$ according to condition (3) in the previous definition.

**Definition 5 (Mapping Rule).** A mapping rule is an expression

$$\varphi_2(\bar{x}) \rightarrow \varphi_1(\bar{x})$$

where $\varphi_2(\bar{x})$ and $\varphi_1(\bar{x})$ are XPath patterns.

Target path expression $\varphi_T$ only can contain attributes that also appear in $\varphi_S$. This constraint might be relaxed by using Skolem functions, as explained later.

**Example 4.** Figure 3 shows three mapping rules.
Formal Semantics.

Evaluating an XPath pattern $\varphi(\bar{x})$ over a document $d$ consists in finding (variable) embeddings of the associated tree pattern into the XML tree of $d$.

**Definition 6 (Embedding).** Given a pattern $\varphi(\bar{x})$ and a document state $d = (\tau, \text{uri})$, an embedding $\varepsilon$ of $\varphi(\bar{x})$ into $d$ is a tree homomorphism from the tree pattern associated with $\varphi(\bar{x})$ to the document tree $\tau$ of $d$ that maps (1) all nodes of $\varphi(\bar{x})$ to nodes of $\tau$ by preserving the predicates and the structural constraints (parent/child and ancestor/descendant) and (2) all variables in $\bar{x}$ to the corresponding attribute values.

Each embedding $\varepsilon$ of $\varphi(\bar{x})$ into $d$ defines a binding tuple $\bar{x}/\varepsilon = (id, v_1, \ldots, v_n)$ for all variables $\bar{x} = (\$r, \$x_1, \ldots, \$x_n)$.

**Definition 7 (Pattern Result).** Given a pattern $\varphi(\bar{x})$ and a document state $d \in D_{\infty}$, the pattern result $R_{\varphi}(d)$ is the set of all binding tuples $\bar{x}/\varepsilon$ obtained by an embedding $\varepsilon$ of $\varphi(\bar{x})$ into $d$.

**Example 5.** Tables $R_{\varphi_1}(d_1)$ contain all binding tuples generated by embeddings of the corresponding patterns of Example 3 into the documents of Figure 4.

<table>
<thead>
<tr>
<th>$R_{\varphi_1}(d_1)$</th>
<th>$R_{\varphi_2}(d_2)$</th>
<th>$R_{\varphi_3}(d_3)$</th>
<th>$R_{\varphi_4}(d_4)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$r</td>
<td>$$x</td>
<td>$$r</td>
<td>$$x</td>
</tr>
<tr>
<td>(3)</td>
<td>(3)</td>
<td>(3)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

For defining the application a mapping rule to a service call, we will first define the result of applying a mapping rule to two document states $d$ and $d'$:

**Definition 8 (Document State Provenance).** Let $M : \varphi_S(\bar{x}) \rightarrow \varphi_T(\bar{x})$ be a mapping rule. The application of $M$ to two document states $d$ and $d'$ is denoted $M(d, d')$ and returns a set of provenance links from resource nodes in $d'$ to resource nodes in $d$. $M(d, d')$ is defined by the following algebraic expression on the corresponding embedding tables:

$$M(d, d') = \pi_{\text{in}, \text{out}}(p_{\text{in}}) = R_{\varphi_3}(d) \bowtie p_{\text{out}} = R_{\varphi_2}(d')$$

Operations $\pi$, $\bowtie$ and $\rho$ correspond to the standard relational operators project, join and rename respectively.

Observe that $d$ and $d'$ are not necessarily different, and one can define mappings between resources of the same document.

**Example 6.** The following tables illustrate the applications $M_1(d_1, d_2)$ and $M_2(d_2, d_3)$ of mapping rules $M_1 : \varphi_1 \rightarrow \varphi_3$ and $M_2 : \varphi_4 \rightarrow \varphi_1$ from document state $d_2$ into $d_3$.

<table>
<thead>
<tr>
<th>$R_{\varphi_1}(d_1)$</th>
<th>$R_{\varphi_3}(d_3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$r</td>
<td>$$x</td>
</tr>
<tr>
<td>(3)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

The first mapping creates the provenance link $\odot \rightarrow \odot$ whereas the second mapping creates two additional links $\odot \rightarrow \odot$ and $\odot \rightarrow \odot$.

The data dependencies of each service $s \in S$ are described by a set of mapping rules $M(s)$ which allow to infer for each service call $c_i = (s, t)$ the provenance sub-graph (nodes and links) from resource nodes $out(c_i) = \sigma_{\text{service} = c_i, \text{Service}}$ generated by $c_i$ to resource nodes in the input document state $in(c_i) = d_{i-1}$.

Then, $M(d_{i-1}, d_i)$ returns a set of direct provenance links between the input document state $d_{i-1}$ and the output document state $d_i$ of $c_i$. Since we are only interested by provenance links for resources generated by $c_i$, we can filter out all other provenance links by joining $M(d_{i-1}, d_i)$ with $out(c_i)$.

**Definition 9 (Service Call Provenance).** Let $M : \varphi_S \rightarrow \varphi_T \in M(s)$ be a mapping rule for $s$ and $c_i$ be a service call of $s$. Then we can define the set of direct provenance links of $c_i$ generated by $M$ for $c_i$ as the subset of $M(d_{i-1}, d_i)$ generated between the input document state $d_{i-1}$ and the output document state $d_i$ of $c_i$: $M(c_i) = M(d_{i-1}, d_i) \bowtie out(c_i)$

**Example 7.** The direct provenance links for $c_4 = (\text{Translator}, \text{t3})$ are generated by joining $M(d_2, d_3)$ from the previous example with $out(c_3)$ which only keeps provenance link $\odot \rightarrow \odot$.

Mapping Rule Evaluation.

The formal semantics of a mapping rule for a given service call $c$ is defined over the document states $d$ and $d'$ before and after the call. This semantics can be directly implemented by generating provenance links during the workflow execution where the service orchestrator maintains a copy of the input document and directly applies the rules to the newly generated fragments. This solution has several drawbacks: (i) it is intrusive since it needs the modification of the service orchestrator; (ii) it is inefficient since it might slow down the workflow execution; (iii) it allows for limited optimization since each provenance computation is executed independently which reduces database optimization opportunities like factorization and indexing.

In the following we will discuss the issue of evaluating all service mapping rules after the workflow execution on the whole result document. The main goal is to find for each service call $c = (s, t)$ and each mapping $M : \varphi_S \rightarrow \varphi_T \in M(S)$ the embeddings between the correct document states. A simple, but also inefficient solution is to compute for each service call the document state before and after the call. A second solution is to rewrite the pattern expressions and exploit generated meta-data for computing all embeddings directly on the final document state.

Our solution exploits information about the service call attached to each resource node. This information is generated by the majority of provenance-aware workflow systems [16] and sufficient if we consider sequential workflows where each service call $c_i$ has access to all resources produced by previous calls.

Inferring Direct Provenance: For a workflow execution that produced a final document state $d$ we compute the direct provenance links between the input and the output nodes of each service call $c = (s, t)$ of the workflow. We assume that each resource node has two attributes $\text{@timestamp} = t$ (timestamp) and $\text{@service} = s'$ (service) defining its service call label. We evaluate the XPath expressions for all mappings $M : \varphi_S \rightarrow \varphi_T \in M(S)$ as follows:

1. Rewrite patterns $\varphi_S$ and $\varphi_T$ by adding the following temporal constraints:
• Pattern $\varphi_S$ is extended by adding the condition $[\exists t < t_i]$ to all steps in the expression.
• Pattern $\varphi_T$ is extended by adding XPath condition $[\exists s = s$ and $\exists t = t_i]$ to the final step and condition $[\exists t <= t_i]$ to all the other steps.

2. Apply the new mapping $M' : \varphi_S' \rightarrow \varphi_T'$ to the final document $d_n$ as defined in Definition 8: $M'(d_n, d_n)$.

Observe that since each service call only can add new resources (XML fragments), for all nodes in the final document $d_n$, its creation timestamp is greater or equal to that of all its ancestors. We therefore can remove the temporal tests on the intermediate steps.

Inferring inherited provenance: The previous algorithm generates provenance links for mapping rules encoding explicit data dependencies between resources. However these resources might contain other resources connected by implicit provenance links. For example, in Figure 1(b) there exists an explicit provenance links between resource ② and resource ③ and an implicit link ① → ②. This can easily be achieved by adding to all XPath patterns and additional step descendant-or-self: * before applying the previous rewriting rules.

5. EXTENSIONS

Adding node position constraints.

Our mapping model can naturally be extended by including the XPath node position function $\text{position}(\cdot)$. For example, the following mapping rule maps all B nodes of an A at position i in the input document to the $i^{th}$ node C:

$\text{//A}[B][\exists p := \text{position}(\cdot)]/B \rightarrow /\text{C}[^p = \text{position}(\cdot)]$

Observe that this rule is different from the following rule which takes the position of A independently of the fact that it has a child B:

$\text{//A}[^p := \text{position}(\cdot)]/B \rightarrow /\text{C}[^p = \text{position}(\cdot)]$

Note also that the position attribute is relative to the document state on which the corresponding XPath is evaluated and that it can change from one document state to another.

Adding Skolem functions.

Skolem functions are used in predicate logic for replacing existentially quantified variables by function symbols. They can be used in our mapping model for defining different kinds of aggregation as described in [15]. In the following examples $\exists a$ denotes non identifier attributes and $\exists id$ are identifier attributes.

One-to-many: The following rule states that all C nodes with the same value for some attribute $\exists a$ assigned to variable $\exists y$ are generated by a single A node with some unique identifier assigned to $\exists x$:

$\text{//A}[\exists x := \exists id] \rightarrow \exists y//\text{C}[\exists y := \exists a]$

This rule can be replaced by rule

$\text{//A}[\exists x := \exists id] \rightarrow /\text{C}[f(\exists x) = \exists a]$

Many-to-one: The following mapping links a unique C to all A nodes with the same value for some attribute $\exists a$:

$\text{//A}[\exists x := \exists a] \rightarrow /\text{C}[f(\exists x) = \exists id]$

6. IMPLEMENTATION

The goal of our first implementation was to integrate our provenance model into the existing WebLab platform. Following the core architecture of the platform, all components are implemented in Java and deployed within an application server (Apache Tomcat) and a service-oriented processing middleware (PETALS). The workflow execution engine is based on XML for representing data and on RDF for encoding meta-data and ontologies. Pursuing this policy, we decided to adopt PROV-RDF and separate WebLab documents (stored in an XML repository) from their meta-data stored in a Sesame RDF repository. Using an RDF triple-store allows us to use SPARQL endpoints for querying generated provenance graphs.

Our first goal was to illustrate the feasibility of our model without dealing with scalability issues. WebLab applications feature workflow executions of different sizes and we plan to evaluate the performance of our current solution in the near future. We also believe that the data-centric and declarative nature of our provenance model and mapping language will enable the future use of existing query processing optimization techniques for obtaining good performance at a large scale.

The overall architecture shown in Figure 5 is divided into three parts:

1. The Recording part, where the Recorder module receives all input and output messages of a service call, in addition to the generated execution meta-data (timestamp and service). Resources generated by a given service call are identified by the workflow engine by comparing the call’s input and output document states,

http://tomcat.apache.org/

http://petals.ow2.org/

http://www.openrdf.org/
using a standard XML-diff service. The Recorder finally replaces the updated WebLab document in the Resource Repository (1) and transmits all generated meta-data (service, timestamp, generated_nodes) to the Execution Trace triple-store for future use.

2. The Graph Construction part contains a Service Catalog with meta-data about services including the service endpoints and signatures as well as the provenance mapping rules. It also includes a Provenance triple-store storing the provenance graphs according to the RDF-PROV ontology. These two components are linked to the Mapper which is the main component of our architecture.

3. The Request Manager receives provenance queries for specific workflow executions. It first checks in the Provenance triple-store if the graph has already been materialized by a previous query (3). If not, the Mapper materializes the request by applying the corresponding mapping rules on the execution trace (4). It collects all execution trace triples corresponding to the workflow specified by some user query (5). It then calls the Resource Repository for obtaining the final resource of the workflow execution (6). The mapping rules corresponding to the executed service executed are obtained from the Service Catalog (7). All these data and rules are then combined to construct an XQuery expression for building the provenance graph. The basic idea is to translate mapping rules into XQuery expressions which compute the embeddings of the source and the target XPath expression.

We consider the following two solutions for computing data dependency links. The first solution uses timestamps in order to identify the input and output resources of a given service call. The Mapper then rewrites the mapping rules into two XQuery expressions applied on the input resource and on the output resource respectively. Both results (set of embeddings) are then joined, returning a set of provenance and on the output resource respectively. Both results (set of executed information of which service has produced which data independently of the workflow control flow. Using dynamically generated information of which service has produced which data (execution trace), the second step connects these rules to services. This combination of statically defined data dependencies and dynamically generated service dependencies facilitates the work of workflow designers.

Other work on fine-grained provenance[7, 3] consider a white-box approach exploiting fine-grained descriptions of

7. RELATED WORK

Provenance is an important notion in the context of scientific workflows. The Taverna scientific workflow system implements the Janus provenance model [22]. The provenance graph is generated during the workflow execution (using an event monitor detecting the activation and termination components, as well as data accesses) and stored in a relational database (mySQL). Provenance graphs can be queried via an API and exported in the OPM (Open Provenance Model [24]) or the Janus RDF format. The Kepler provenance add-on [1] is similar to Janus. It uses its own model, and records provenance data in a relational database during execution time. Finally, Vistrails [11] behaves similarly to both previous systems by generating and storing XML provenance data.

In the previous systems, data dependency mappings are part of the workflow description, while in WebLab PROV provenance links are generated in two separate steps. The first steps consists in defining mapping rules by only considering data dependencies in the expected result independently of the workflow control flow. Using dynamically generated information of which service has produced which data (execution trace), the second step connects these rules to services. This combination of statically defined data dependency specifications and dynamically generated service dependencies facilitates the work of workflow designers.
workflows and services. We consider black-box services for which we infer provenance links by defining fine-grained data dependency mappings between input and output XML fragments. [7] study the problem of efficiently answering reachability queries over views of provenance graphs. Such provenance views include composite modules (for focusing on relevant or hiding private provenance information) and model fine-grained dependencies between module inputs and outputs. View specifications are labeled statically (i.e. as they are created) and data items are labeled dynamically as they are produced during a workflow execution. This work is complementary to ours, the statically defined provenance mapping rules could also be used to generate different provenance views over the same workflow execution.

The solution proposed by [9] is probably the most similar to our approach. Nevertheless, it also differs in some points. The main difference relies in the way execution traces are represented and used for computing provenance links. In [9], an execution trace contains the set of ordered actor steps and their corresponding parameter updates. This trace is not part of the data. In WebLab, all modifications or updates have to be translated into insertions. This follows the idea of artifact-based workflow approach which enables provenance link inference by using standard query languages like XPath and XQuery.

[13] uses inference rules specified as SPARQL graph patterns to build the RDF data dependency graph of a scientific workflow run. These rules specify the type of links that should be added between given input and output data objects of a task run. In our case, XPath mappings are used to find the specific data objects (in our case XML fragments) that should be linked together.

In the case of [23], data processors are also black-boxes and fine-grained data dependencies are specified between individual elements of input and output lists. The work of [5] also considers XML provenance for transformations with update and add-only semantics. The system generates data nodes usage annotations (delete, insert) during the workflow execution. In our case, we do not interfere with the workflow execution, but only deduce provenance links by evaluating XPath mappings on the final document.

8. CONCLUSION

This article proposes a method for capturing fine-grained provenance information from an open service composition environment, where services are considered as black-boxes in the sense that their implementation cannot be inspected or modified.

Our provenance model is non-invasive and is generic enough to be applied to any provenance-agnostic XML transformations, independently of their internal implementation. We used explicit data dependency mappings, inspired from the existing works on schema mapping, in order to create the provenance graph of a workflow execution. The separation between data dependency mapping rules and their association with service calls makes our model also applicable for settings where the concrete input and output interface of a service is unknown. Our prototype allows to evaluate XPath mappings on the final resources and to register RDF provenance graphs (some kind of retrospective provenance information as described in [18]). By using the PROV ontology, the RDF representation of provenance meta-data can easily replaced other formats like PROV-XML.

In our model we only assumed sequential workflow executions. However it can be extended to more complex execution patterns including nesting and parallel service executions. The main issue is to capture enough information during the execution for inferring the document state before and after each service call. This can be done by adding additional meta-data for identifying different control flow “channels” during provenance link inference like in [9]. We intend to thoroughly analyze our generated provenance information, in order to conceive efficient provenance storage and querying methods [12, 5, 4].

For now, our provenance system is integrated into the existing Cassidian WebLab project that features many workflow executions of different sizes. Whereas our first implementation only tries to illustrate the feasibility of our approach without dealing with scalability issues. We plan to evaluate the performance of our current solution in the near future. We also believe that the data-centric and declarative nature of our provenance model and mapping language will enable the future use of existing query processing optimization techniques for obtaining good performance at a large scale.

9. REFERENCES


