

# Mapping the NRC Dataflow Model to the Open Provenance Model

Natalia Kwasnikowska and Jan Van den Bussche

Hasselt University and Transnational University of Limburg, Belgium

**Abstract.** The Open Provenance Model (OPM) has recently been proposed as an exchange framework for workflow provenance information. In this paper we show how the NRC data model for workflow repositories can be mapped to the OPM. Our mapping includes such features as complex data flow in an execution of a workflow; different workflows in the repository that call each other; and the tracking of subvalues of complex data structures in the provenance information. Because the NRC dataflow model has been formally specified, also our mapping can be formally specified; in particular, it can be automated. To facilitate this specification, we present an adapted set-theoretic formalization of the basic OPM.

## 1 Introduction

The Open Provenance Model (OPM) has recently been proposed as an exchange framework for workflow provenance information [1]. In order to validate this new framework, it is important to investigate how existing models and systems for provenance can be mapped to the OPM. In this paper, we do this exercise for a data model for workflow repositories which we recently introduced, called the NRC dataflow model [2].

The NRC dataflow model is a formally specified data model for workflows which emphasize data manipulation and data management. Hence we usually refer to such workflows as *dataflows*. The NRC dataflow model incorporates important aspects such as complex-data flow governed by expressions of the Nested Relational Calculus (NRC [3]); use of external services; formal representation of past executions; tracking of subvalues of a complex data structure in a past execution; and different dataflows in a repository that call each other. We will propose a representation in the OPM of all these features of our model. For example, to model the execution of one dataflow, called as a subdataflow in another dataflow, we use the interesting “accounts” feature provided by the OPM.

In this paper we assume familiarity with the OPM [1]. We will, however, give a set-theoretical formal definition of the OPM, adapted from the original set-theoretical formalization. We will use this definition to specify our NRC-to-OPM mapping formally.

This paper is organized as follows. In Section 2, we recall the basics of the NRC dataflow model. In Section 3, we give our formal definition of the OPM. In

Section 4, we describe the mapping from the NRC model to the OPM. We also show how an OPM description of an NRC dataflow execution can be augmented with information to track the provenance of a subvalue occurring in the final result of this execution.

## 2 The NRC Dataflow Model

In this section we present aspects of the NRC dataflow model that are relevant to workflow provenance information. For a more detailed description of the NRC dataflow model and repository we refer to the paper [2].

### 2.1 Specification of Dataflows in NRC

Consider the following computation, based on a real proteomics protocol [4]: “Given a set of raw data produced by a mass spectrometer in a proteomics experiment, and all its associated parameters, generate a list of proteins possibly identified in this experiment”. We can express this computation in the following NRC dataflows:

```
dataflow identify(data: TMSdata, p: Parameters): ProteinCandidateList is
  let list := for x in data return
    ⟨id: x.id, spectra: extract(x.file)⟩
  in validation(search1(list, p), search2(list, p), search3(list, p))
```

```
dataflow search(data: TMSextracted, p: Parameters): AAList is
  for x in data return
    ⟨id: x.id, aalist: for y in x.spectra return dbSearch(y, p)⟩
```

Each dataflow states its signature, i.e., the types for its input parameters and for its result. Dataflow *identify* expects one parameter of type **Parameters** and one of type TMSdata. Type **Parameters** is a *base type*. Values of base types, called *base values*, are considered to be “atomic” to the operations in the dataflow. In our example, a value of type **Parameters** is an XML document conforming to a specific DTD. Type TMSdata is a *complex type*. Complex values are constructed using record and set constructions from base values, conforming to their type. For example, type TMSdata equals  $\{\langle id: \mathbf{Number}, file: \mathbf{RawFile} \rangle\}$ , that is, a set of records with first component labeled *id*, and second component labeled *file*. We use value *tmsInput* of this type, illustrated in Fig. 1, as first input parameter to dataflow *identify*.

Both dataflows are composed of NRC expressions and *service calls*. NRC is a simple functional programming language [3], built around the basic operations on sets and records, with for-loops, if-then-else and let-statements as the only programming constructs. Service calls model all other actions in the dataflows. These can be calls to external services, such as NCBI BLAST or MASCOT search, but also calls to library functions provided by the underlying system, such as addition for numbers or concatenation for strings. Moreover, one dataflow

$tmsInput =$	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px 5px;"><i>id</i></th> <th style="padding: 2px 5px;"><i>file</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">rawVial10</td> </tr> <tr> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">rawVial11</td> </tr> <tr> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> </tr> <tr> <td style="padding: 2px 5px;">55</td> <td style="padding: 2px 5px;">rawVial64</td> </tr> </tbody> </table>	<i>id</i>	<i>file</i>	1	rawVial10	2	rawVial11	⋮	⋮	55	rawVial64
<i>id</i>	<i>file</i>										
1	rawVial10										
2	rawVial11										
⋮	⋮										
55	rawVial64										

$tmsOutput =$	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th rowspan="2" style="padding: 2px 5px;"><i>protID</i></th> <th rowspan="2" style="padding: 2px 5px;"><i>prob</i></th> <th colspan="3" style="padding: 2px 5px;"><i>evidence</i></th> </tr> <tr> <th style="padding: 2px 5px;"><i>peptide</i></th> <th style="padding: 2px 5px;"><i>score</i></th> <th style="padding: 2px 5px;"><i>spectrum</i></th> </tr> </thead> <tbody> <tr> <td rowspan="4" style="padding: 2px 5px;">Protein<sub>1</sub></td> <td rowspan="4" style="padding: 2px 5px;">99</td> <td style="padding: 2px 5px;">pep<sub>1</sub></td> <td style="padding: 2px 5px;">9</td> <td style="padding: 2px 5px;">spectrum<sub>vial10,5</sub></td> </tr> <tr> <td style="padding: 2px 5px;">pep<sub>2</sub></td> <td style="padding: 2px 5px;">7</td> <td style="padding: 2px 5px;">spectrum<sub>vial23,2</sub></td> </tr> <tr> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> </tr> <tr> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> </tr> <tr> <td rowspan="4" style="padding: 2px 5px;">Protein<sub>2</sub></td> <td rowspan="4" style="padding: 2px 5px;">96</td> <td style="padding: 2px 5px;">pep<sub>8</sub></td> <td style="padding: 2px 5px;">9</td> <td style="padding: 2px 5px;">spectrum<sub>vial57,3</sub></td> </tr> <tr> <td style="padding: 2px 5px;">pep<sub>1</sub></td> <td style="padding: 2px 5px;">8</td> <td style="padding: 2px 5px;">spectrum<sub>vial10,5</sub></td> </tr> <tr> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> </tr> <tr> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> </tr> <tr> <td style="padding: 2px 5px;">⋮</td> <td style="padding: 2px 5px;">⋮</td> <td colspan="2" style="padding: 2px 5px;">⋮</td> </tr> </tbody> </table>	<i>protID</i>	<i>prob</i>	<i>evidence</i>			<i>peptide</i>	<i>score</i>	<i>spectrum</i>	Protein <sub>1</sub>	99	pep <sub>1</sub>	9	spectrum <sub>vial10,5</sub>	pep <sub>2</sub>	7	spectrum <sub>vial23,2</sub>	⋮	⋮	⋮	⋮	⋮	⋮	Protein <sub>2</sub>	96	pep <sub>8</sub>	9	spectrum <sub>vial57,3</sub>	pep <sub>1</sub>	8	spectrum <sub>vial10,5</sub>	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
<i>protID</i>	<i>prob</i>			<i>evidence</i>																																					
		<i>peptide</i>	<i>score</i>	<i>spectrum</i>																																					
Protein <sub>1</sub>	99	pep <sub>1</sub>	9	spectrum <sub>vial10,5</sub>																																					
		pep <sub>2</sub>	7	spectrum <sub>vial23,2</sub>																																					
		⋮	⋮	⋮																																					
		⋮	⋮	⋮																																					
Protein <sub>2</sub>	96	pep <sub>8</sub>	9	spectrum <sub>vial57,3</sub>																																					
		pep <sub>1</sub>	8	spectrum <sub>vial10,5</sub>																																					
		⋮	⋮	⋮																																					
		⋮	⋮	⋮																																					
⋮	⋮	⋮																																							

**Fig. 1.** Complex values of types, respectively, TMSdata and ProteinCandidateList

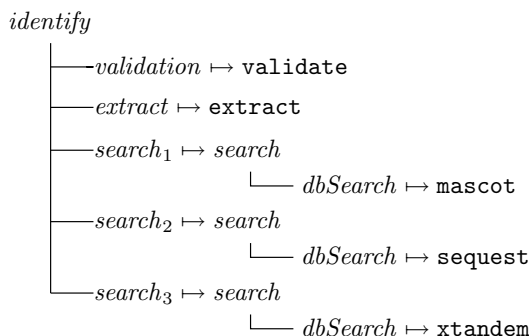
can appear as a service call in another dataflow, thus becoming its *subdataflow*. Each service must be supplied with a signature describing the types of its input parameters and its result. In our example, dataflow *identify* uses the services

$extract(raw : \mathbf{RawFile}) : \{\mathbf{TMSfile}\}$   
 $validation(p_1 : \mathbf{AAlist}, p_2 : \mathbf{AAlist}, p_3 : \mathbf{AAlist}) : \mathbf{ProteinCandidateList}$

and also the services  $search_1$ ,  $search_2$ , and  $search_3$ , all three with the same signature ( $list : \mathbf{TMSextracted}, p : \mathbf{Parameters}) : \mathbf{AAlist}$ .

Before we can execute a dataflow, we not only need to provide values for its parameters, but we also need to provide meaning to called services by assigning them to actual services. In our example, in dataflow *identify*, we bind *extract* and *validation* to external applications **extract** and **validate**, and we bind services  $search_1$ ,  $search_2$ , and  $search_3$  to dataflow *search*. The latter becomes thus a subdataflow of *identify*. As *search* also contains a service call, namely, *dbSearch*, we now need to provide binding for that service for each call to *search* in *identify*: to external service **mascot** in the first, to external service **sequest** in the second, and to external service **xtandem** in the third. (These three external services stand for three database search engines frequently used in identification of mass spectra in proteomics research.) We sum up all these service call bindings in a data structure which we call a *binding tree*, shown in Fig. 2.

Suppose now that we have executed dataflow *identify* with value *tmsInput* for parameter *data* and file PXML for parameter *p*. Suppose that value *tmsOutput* has been returned (see Fig. 1).



**Fig. 2.** Binding tree for dataflow *identify*

## 2.2 Past Executions of Dataflows

In order to keep a record of an execution of dataflow *identify*, it is not sufficient to store the input values, the result, and the binding tree for services. Indeed, the transient nature of the proteomics databases used by the search engines implies that also intermediate results must be stored. Since all NRC expressions are deterministic, only information about intermediate values of external services must be effectively stored. We naturally represent this information as a number of triples of the form  $(s, \sigma, v)$ , where  $s$  is an occurrence of a service call in the dataflow,  $\sigma$  its *value assignment*, i.e., assignment of input values to parameters, and  $v$  the value produced by the service call.

For example, for the execution of dataflow *identify* from Sect. 2.1, we would store the service-call triples shown in Fig. 3.

There,  $tmsExtracted$ ,  $tmsRes_1$ ,  $tmsRes_2$  and  $tmsRes_3$  represent complex values produced in the corresponding step of the computation. As a matter of fact, a complete record of the entire past execution can be automatically derived from such triples, as we have shown elsewhere [2].

**Caveat.** *The complete record of a past execution of some workflow (applied to certain input), leading to a final result, is commonly called the **workflow provenance** of that result. In this paper, we refer to such a record as a **run**.*

In our model, a run is basically a set of triples like those for the service calls, except that now we have a triple for each occurrence of each subexpression in the dataflow. Note that, in particular, the final result value is also contained in the run, namely, in the triple for the entire top-level expression of the dataflow. Figure 4 shows a few examples of such additional triples that would be part of our example run of dataflow *identify*.

Of course, when a service call has been executed that was bound to a subdataflow, we necessarily also have a run of that subdataflow for the given values of its parameters. In our example, the three service-call triples for  $search_1$ ,  $search_2$  and  $search_3$ , will need to be linked to corresponding runs of the dataflow *search*.

<i>service</i>	<i>assignment</i>	<i>value</i>
<i>extract</i>	<i>data</i> = <i>tmsInput</i> <i>p</i> = PXML <i>x</i> = $\langle id : 1, file : rawVial10 \rangle$ <i>raw</i> = <i>rawVial10</i>	<i>spectrum</i> <sub>vial10,1</sub> ⋮ <i>spectrum</i> <sub>vial10,5</sub>
<i>extract</i>	<i>data</i> = <i>tmsInput</i> <i>p</i> = PXML <i>x</i> = $\langle id : 2, file : rawVial11 \rangle$ <i>raw</i> = <i>rawVial11</i>	<i>spectrum</i> <sub>vial11,1</sub> <i>spectrum</i> <sub>vial11,2</sub> <i>spectrum</i> <sub>vial11,3</sub>
⋮	⋮	⋮
<i>extract</i>	<i>data</i> = <i>tmsInput</i> <i>p</i> = PXML <i>x</i> = $\langle id : 55, file : rawVial64 \rangle$ <i>raw</i> = <i>rawVial64</i>	<i>spectrum</i> <sub>vial64,1</sub> ⋮ <i>spectrum</i> <sub>vial64,13</sub>
<i>search</i> <sub>1</sub>	<i>data</i> = <i>tmsInput</i> <i>p</i> = PXML <i>list</i> = <i>tmsExtracted</i>	<i>tmsRes</i> <sub>1</sub>
<i>search</i> <sub>2</sub>	<i>data</i> = <i>tmsInput</i> <i>p</i> = PXML <i>list</i> = <i>tmsExtracted</i>	<i>tmsRes</i> <sub>2</sub>
<i>search</i> <sub>3</sub>	<i>data</i> = <i>tmsInput</i> <i>p</i> = PXML <i>list</i> = <i>tmsExtracted</i>	<i>tmsRes</i> <sub>3</sub>
<i>validation</i>	<i>data</i> = <i>tmsInput</i> <i>p</i> = PXML <i>list</i> = <i>tmsExtracted</i> <i>p</i> <sub>1</sub> = <i>tmsRes</i> <sub>1</sub> <i>p</i> <sub>2</sub> = <i>tmsRes</i> <sub>2</sub> <i>p</i> <sub>3</sub> = <i>tmsRes</i> <sub>3</sub>	<i>tmsOutput</i>

**Fig. 3.** Service-call triples from our example run

### 2.3 NRC Dataflow Repository Model

To summarize, if we want to have a complete record of an execution of a dataflow, we need to store the value assignment for its parameters, the binding tree for its services, all service-call triples of the run, and links to the runs of its sub-dataflows.

All this information, for different dataflows and executions, can be stored in a global dataflow repository. A conceptual schema illustrating the different entities that play a role in such a repository, and their relationships, is given in Fig. 5. There, mapping *internalcall* links runs of dataflows to the runs of its subdataflows. Given the identifier of a run of a dataflow, and a service-call triple from that run with the service bound to a subdataflow, the mapping will indicate the run identifier of the corresponding subdataflow run.

A very important integrity constraint is that the repository is *closed* by *internalcall*, i.e., if the repository contains a run of some dataflow, then it also

<i>subexpression</i>	<i>assignment</i>	<i>value</i>	
<i>let</i>	$data = tmsInput$ $p = PXML$	$tmsOutput$	
<i>for</i>	$data = tmsInput$ $p = PXML$	$tmsExtracted$	
$\langle id, spectra \rangle$	$data = tmsInput$ $p = PXML$ $x = \langle id : 2, file : rawVial11 \rangle$	<i>id</i>	<i>spectra</i>
		1	$spectrum_{vial11,1}$
		2	$spectrum_{vial11,2}$
			$spectrum_{vial11,3}$

Fig. 4. Some run triples from our example run

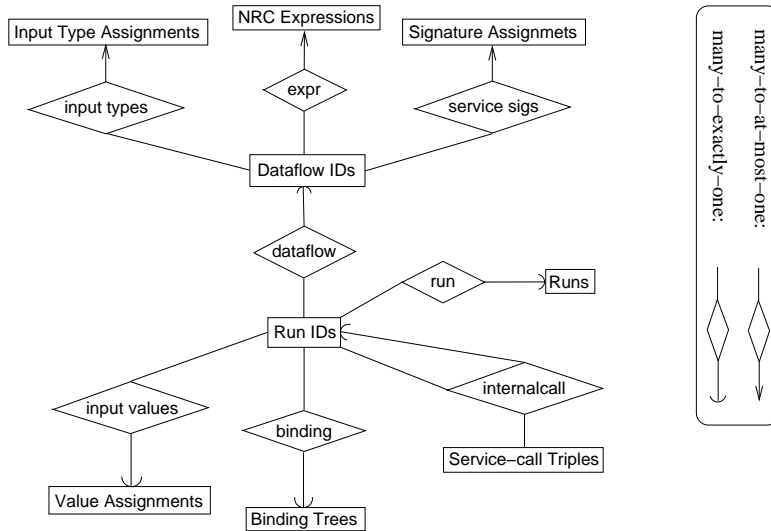


Fig. 5. E/R diagram of the NRC dataflow repository model

contains, for all its service-call triples, the corresponding runs of its subdataflows. Again, for a more detailed description of the repository and its constraints we refer to the paper [2].

### 3 Formal Definition of OPM Graphs

In this section we present a set-theoretic definition of the Open Provenance Model, adapted from the original timeless causality graph data model [1]. The main difference is in the treatment of account memberships, which we consider to be labels of nodes and edges, and as such we define the account-membership function *accountOf* to be part of an OPM graph. In our opinion this is a cleaner formalisation; the original formalisation of accounts in OPM [1] seems flawed. We also define an alternate as a set of accounts, rather than merely a pair of accounts. We believe this added generality can be useful in practice.

As we need only artifact nodes and process nodes to represent an NRC run in the OPM, we leave out agent nodes and their associated edges from the definitions.

All primitive sets are assumed to be pairwise disjoint. The set  $OPMGraph$ , as defined below, is the set of all possible OPM graphs.

$$\begin{aligned}
ProcessId & : \text{primitive set containing all process nodes} \\
ArtifactId & : \text{primitive set containing all artifact nodes} \\
Role & : \text{primitive set containing all roles} \\
Account & : \text{primitive set containing all accounts} \\
Used & \stackrel{def}{=} ProcessId \times Role \times ArtifactId \\
WasGeneratedBy & \stackrel{def}{=} ArtifactId \times Role \times ProcessId \\
WasTriggeredBy & \stackrel{def}{=} ProcessId \times ProcessId \\
WasDerivedFrom & \stackrel{def}{=} ArtifactId \times ArtifactId \\
Alternate & \stackrel{def}{=} \mathbb{P}(Account)
\end{aligned}$$

$$\begin{aligned}
OPMGraph & \stackrel{def}{=} \{ \langle A, P, U, G, T, D, AL, accountOf \rangle \mid A \subseteq ArtifactId, \\
& P \subseteq ProcessId, U \subseteq P \times Role \times A, G \subseteq A \times Role \times P, \\
& T \subseteq P \times P, D \subseteq A \times A, AL \subseteq Alternate, \\
& accountOf : (A \cup P \cup U \cup G \cup T \cup D) \rightarrow \mathbb{P}(Account) \}
\end{aligned}$$

Before we reformulate relevant aspects of the OPM according to the adapted definition, we introduce the following convenient notations for any given OPM graph  $g = \langle A, P, U, G, T, D, AL, accountOf \rangle$ :

$$\begin{aligned}
A^g & \stackrel{def}{=} A & U^g & \stackrel{def}{=} U \\
P^g & \stackrel{def}{=} P & G^g & \stackrel{def}{=} G \\
Nodes^g & \stackrel{def}{=} A \cup P & T^g & \stackrel{def}{=} T \\
AL^g & \stackrel{def}{=} AL & D^g & \stackrel{def}{=} D \\
accountOf^g & \stackrel{def}{=} accountOf & Edges^g & \stackrel{def}{=} U \cup G \cup T \cup D \\
Elements^g & \stackrel{def}{=} Nodes^g \cup Edges^g
\end{aligned}$$

Apart from  $Nodes^g$ ,  $Edges^g$  and  $Elements^g$ , the above notations may seem superfluous, but they will prove convenient when referring to several OPM graphs at the same time. Observe also that a  $g \in OPMGraph$  is completely determined by  $Elements^g$ ,  $AL^g$ , and  $accountOf^g$ . We will make use of this observation when defining new OPM graphs.

*Edges and Equality of Edges* Note that any edge  $e \in Edges^g$ , for an OPM graph  $g$ , either belongs to

$$Used \cup WasGeneratedBy$$

and is then of the form  $e = \langle x_1, r, x_2 \rangle$  with  $r$  some role, or belongs to

$$WasTriggeredBy \cup WasDerivedFrom$$

and is then of the form  $e = \langle x_1, x_2 \rangle$ . In both cases we introduce the notation  $Src(e)$  to denote the source node of  $e$ , i.e.,  $x_1$ , and  $Dest(e)$  to denote the destination node of  $e$ , i.e.,  $x_2$ . We also say that  $x_1$  and  $x_2$  are *incident* to  $e$ , and denote this by  $isIncident(x_i, e)$  for  $i = 1, 2$ . Note that two causality edges are considered to be equal simply if they are equal in the mathematical sense, i.e., they are the same tuple.

*Effective Account Membership* For a given OPM graph  $g$ , we define the function

$$effectiveAccountOf^g: Elements^g \rightarrow \mathbb{P}(Account)$$

as follows:

- If  $x \in Nodes^g$ , then

$$effectiveAccountOf^g(x) = accountOf^g(x) \cup \bigcup \{accountOf^g(e) \mid e \in Edges^g \text{ and } isIncident(x, e)\} .$$

- If  $e \in Edges^g$ , then we simply put  $effectiveAccountOf^g(e) = accountOf^g(e)$ . (This latter definition may seem superfluous but will prove convenient in the definition of account views.)

*The Union of Two OPM Graphs* Let  $g_1$  and  $g_2$  be two OPM graphs. We define the *union* of  $g_1$  and  $g_2$ , denoted by  $g_1 \sqcup g_2$ , as follows:

$$\begin{aligned} Elements^{g_1 \sqcup g_2} &\stackrel{def}{=} Elements^{g_1} \cup Elements^{g_2} , \\ AL^{g_1 \sqcup g_2} &\stackrel{def}{=} AL^{g_1} \cup AL^{g_2} , \end{aligned}$$

and  $accountOf^{g_1 \sqcup g_2}$  is the point-wise union of  $accountOf^{g_1}$  and  $accountOf^{g_2}$ .

*Account Views* For a given OPM graph  $g$  and an account  $\alpha$ , we now formally define the *account view* of  $g$  according to  $\alpha$ , denoted by  $view(g, \alpha)$ , as follows:

- $Elements^{view(g, \alpha)} \stackrel{def}{=} \{x \in Elements^g \mid \alpha \in effectiveAccountOf^g(x)\}$ ;
- $accountOf^{view(g, \alpha)}$  is the restriction of  $accountOf^g$  to  $Elements^{view(g, \alpha)}$ ;
- $AL^{view(g, \alpha)} \stackrel{def}{=} \{alt \cap ActAcc \mid alt \in AL^g\}$ , where  $ActAcc$  stands for the set of accounts that actually appear in the image (range) of  $accountOf^{view(g, \alpha)}$ .

Note that  $view(g, \alpha)$  is again an OPM graph.

*Legal Account Views* Before we formally define legal account views of an OPM graph, we point out that we can associate to any given OPM graph  $g$  a classical directed graph  $DG(g) = (V(g), E(g))$  with set of vertices  $V(g)$  equal to  $Nodes^g$  and set of directed edges  $E(g)$  equal to  $\{(Src(e), Dest(e)) \mid e \in Edges^g\}$ . Accordingly, we call  $g$  *acyclic* precisely when  $DG(g)$  is.

Now for an OPM graph  $g$  and an account  $\alpha \in Account$ , the account view  $view(g, \alpha)$  is considered to be *legal* when it is acyclic, and there do not exist two different edges in  $G^{view(g, \alpha)}$  with the same source node, i.e.

$$\forall e_1, e_2 \in G^{view(g, \alpha)}: Src(e_1) = Src(e_2) \Rightarrow e_1 = e_2 .$$

*Legal OPM Graph* An OPM graph is *legal* when all its account views are legal.

*Alternate* For an OPM graph  $g$  and  $alt \in AL^g$ , we call  $alt$  an *alternate* in  $g$ , and we call each  $\alpha \in alt$  an *alternative* in  $alt$ .

*Legal Alternate* For an OPM graph  $g$  and an alternate  $alt \in AL^g$ , we call  $alt$  *legal* for  $g$  if

$$\bigcap_{\alpha \in alt} Nodes^{view(g,\alpha)} \neq \emptyset.$$

## 4 Mapping NRC Dataflow Runs to OPM Graphs

Recall that a run of an NRC dataflow is modeled as a table

$$R(subexpression, assignment, value)$$

holding triples of the form  $(e, \sigma, v)$ , where  $e$  is an occurrence of a subexpression of the dataflow,  $\sigma$  the value assignment, and  $v$  the produced complex value. An important property of a run is that the pair  $(subexpression, assignment)$  is a key for the table  $R$ .

We now define an OPM graph  $g$  representing the information stored in  $R$ . To do so we first specify all nodes of  $g$ , then all the edges, and finally, the account membership function *accountOf* and set *Alternate*. The graph  $g$  will be a legal OPM graph. We introduce extra labels for the nodes of the graph  $g$ , such that the information contained in  $g$  will be sufficient to reconstruct  $R$  on the basis of its structure, and node and edge labels alone.

*Process Nodes* First we specify the set  $P^g$  of all process nodes of  $g$ . As each triple in  $R$  actually describes one step of the computation, we need at least one process node for each triple. For most triples it is indeed sufficient to construct one process node. We can construct a unique ID for each node simply by using its corresponding triple as an ID:

$$P^g \stackrel{def}{=} \{[t] \mid t \in R\}.$$

We label each process node  $[t]$ , for  $t = (e, \sigma, v)$ , by  $e$ 's top-level NRC operator (or service call).

The only exception are for-loops. In order to model the different parallel executions of the body of a for-loop (namely, one execution for each element of the set over which the for-loop operates), we split each process node  $[t]$  as above, when  $e$  is a for-loop, into two process nodes  $[dispatch, t]$  and  $[collect, t]$ .

So formally we redefine  $P^g$  as follows:

$$P^g \stackrel{def}{=} \{[t] \mid t = (e, \sigma, v) \in R \text{ and } e \text{ is not a for-loop}\} \\ \cup \bigcup \{[dispatch, t], [collect, t] \mid t = (e, \sigma, v) \in R \text{ and } e \text{ is a for-loop}\}.$$

*Artifact Nodes* Next we construct the set  $A^g$  of all artifact nodes of  $g$ . As each triple in  $R$  contains the complex value produced by the corresponding step, we need one artifact node for each triple. We can again construct a unique ID for each node by using its corresponding triple as a part of its ID: for each triple  $t = (e, \sigma, v)$  we have an artifact node with ID  $[val, t]$ ; we label this node with the value  $v$ .

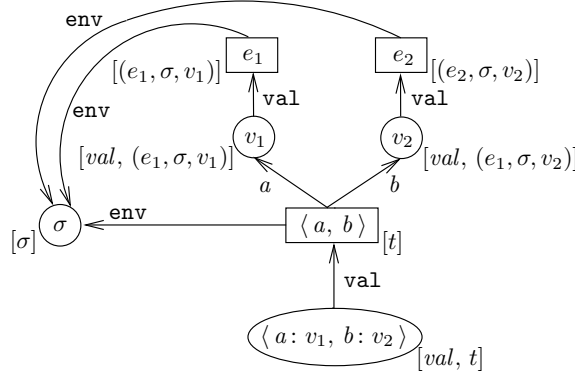
Moreover, each triple contains the value assignment, under which the corresponding step is performed. This value assignment is another artifact in our run. Thus, for each value assignment we create an artifact node, and we can use the value assignment itself as its ID.

We conclude that:

$$A^g \stackrel{def}{=} \{[val, t] \mid t \in R\} \cup \{[\sigma] \mid \exists e, v: (e, \sigma, v) \in R\} .$$

*Edges* We now define the “was generated by” and “used” edges that play a role for a given process node  $[t]$  with  $t = (e, \sigma, v)$ . Since for-loops and let-statements have a body that involves a local variable, these operators involve the extension of their value assignment with a new value for the local variable, and we must treat them separately.

So first assume  $e$  is not a for-loop or a let-statement. The general idea is that  $v$  is produced from the value, or values, that resulted from the constituent subexpressions of  $e$ . For example, if  $e$  is a record construction  $\langle a: e_1, b: e_2 \rangle$ , then  $v$  equals  $\langle a: v_1, b: v_2 \rangle$  where  $v_1$  (resp.  $v_2$ ) is the result of ...  $e_1$  (resp.  $e_2$ ) under the same value assignment  $\sigma$ . We thus generate the edges shown in Fig. 6. Note the role **val** for the edge from  $v$  to  $e$ , and the role **env** (short for environment) for the edge from  $e$  to  $\sigma$ . (Indeed,  $\sigma$  provides the “environment” for the evaluation of  $e$ .) Note also the roles  $a$  and  $b$  that connect  $e$  to  $v_1$  and  $v_2$ , which are in turn



**Fig. 6.** OPM subgraph for a record construction operation

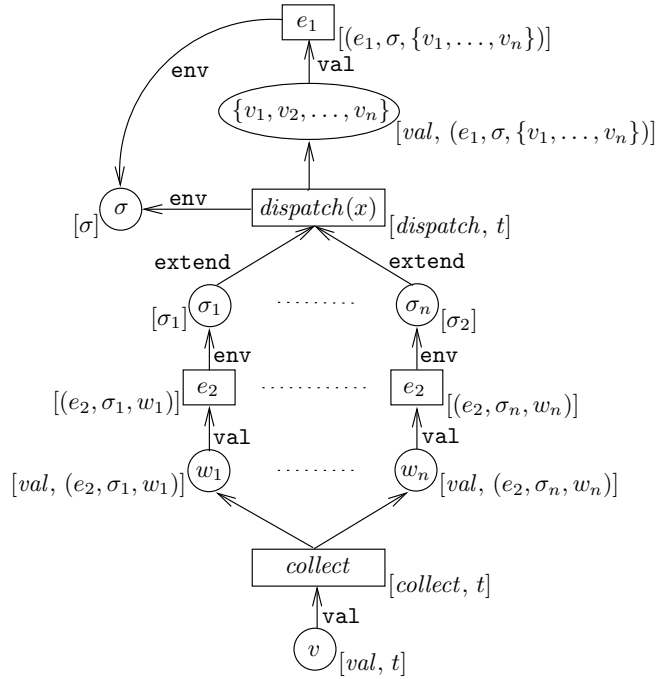
connected to  $e_1$  and  $e_2$  by edges with the role **val**. Clearly,  $e_1$  and  $e_2$  may have constituent subexpressions of their own, so the generation of edges continues

from there. The generation of edges for other operators, except for-loops and let-statements, is analogous.

Now assume  $e$  is a for-loop of the form  $\text{for } x \text{ in } e_1 \text{ return } e_2$ . We recall from the semantics of NRC [2] the semantic rule for such an expression:

$$\frac{\sigma \models e_1 \Rightarrow \{v_1, \dots, v_n\} \quad \forall i \in \{1, \dots, n\} : \sigma_i = \text{extend}(\sigma, x = v_i) \models e_2 \Rightarrow w_i}{\sigma \models \text{for } x \text{ in } e_1 \text{ return } e_2 \Rightarrow v = \{w_1, \dots, w_n\}}$$

We then have the edges as shown in Fig. 7. Note the role **extend** for the edges



**Fig. 7.** OPM subgraph for a for-loop

from  $\text{dispatch}(x)$  to the artifact nodes  $\sigma_1$  through  $\sigma_n$  (the different value assignments for  $e_2$ ). Again,  $e_1$  and  $e_2$  may have constituent subexpressions of their own, so the generation of edges continues from there. The construction of edges for a let-statement is analogous.

Since an NRC dataflow is basically a functional computation, each artifact is either used or generated by a step of the computation. Hence the set  $D^g$  of all “was derived from” edges is empty. Likewise, processes are only connected through artifacts that they either use or generate, so the set  $T^g$  of all “was triggered by” edges is also empty.

*Accounts and Alternate* Finally we need to define account membership for all nodes and edges of the graph  $g$ . We believe that it is sufficient to assign a unique account for all elements of  $g$ , namely the ID of the run  $R$  in the NRC dataflow repository. Indeed, if  $g$  is a representation of run  $R$  with ID  $r$ , then the whole graph  $g$  can be considered to be “an account of the execution according to  $r$ ”:

$$\forall x \in Elements^g : accountOf^g(x) = \{r\} .$$

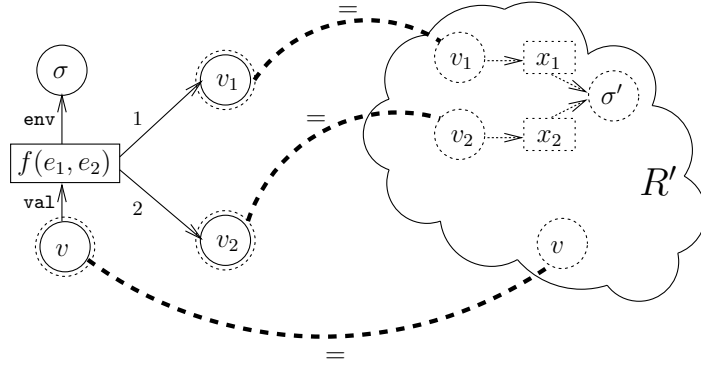
As there is only one account membership for all elements of  $g$ , the set  $AL$  is empty.

#### 4.1 Amendment for Multiple NRC Runs

So far we have constructed a legal OPM graph  $g$  for one run  $R$  stored with ID  $r$ . As an NRC dataflow repository will contain many runs, we would like to be able to generate distinct OPM graphs for all of them. Therefore, we need to amend the identifiers of the nodes of  $g$  by adding the ID of the run to each of them: each node  $[n]$  becomes node  $[r, n]$ .

#### 4.2 Incorporating Runs of Subdataflows

In an NRC dataflow repository, if a dataflow contains subdataflows, then for each run of that dataflow in the repository, there are links to the runs of its subdataflows. These links are provided by the mapping *internalcall* (Sect. 2.3). In Fig. 8 we show how we can merge the OPM graphs of these runs. On the left



**Fig. 8.** Merging the OPM graph of a run  $R$  with the OPM graph of its linked run  $R'$

we see the OPM subgraph of some run  $R$ , representing a service call  $f$ , with two actual parameter values  $v_1$  and  $v_2$  (produced by subexpressions  $e_1$  and  $e_2$ ), and the result value  $v$  of the call.

If  $f$  is bound to a subdataflow, the repository will contain the corresponding run  $R'$  of that subdataflow. In the OPM graph for  $R'$ , we have a value assignment  $\sigma'$  containing the values  $v_1$  and  $v_2$  of the formal parameters  $x_1$  and  $x_2$ . We also find back in this graph the artifact node representing the final result value of the subdataflow; this value obviously equals  $v$ . The nodes on the left (from the OPM graph for  $R$ ) have account  $r$ , i.e., the ID of run  $R$ . The nodes on the right (from the OPM graph for  $R'$ ) have account  $r'$ , i.e., the ID of run  $R'$ . It is clear that account  $r'$  serves as a refinement of account  $r$ .

In order to obtain one combined OPM graph, we identify the left-hand nodes  $v_1$ ,  $v_2$  and  $v$  with the corresponding right-hand nodes  $v_1$ ,  $v_2$  and  $v$ . These identifications are shown by the thick dashed lines in the figure. Then we take the union of the two OPM graphs for  $R$  and  $R'$  (with nodes just identified taken only once). The identified nodes have both accounts  $r$  and  $r'$ .

Finally, we add the composite account  $(r, r')$  to all elements of the combined graph, and we add the set  $\{r, (r, r')\}$  to  $AL$ . The account view of the graph according to  $(r, r')$  provides more details than the view according to  $r$ .

Now if the dataflow corresponding to  $R'$  contains a subdataflow of its own, we can combine the OPM graph for  $R$  and  $R'$  with the OPM graph for the run linked by *internalcall* to that subdataflow, say a run  $R''$  with ID  $r''$ . Then alternate  $\{r, (r, r')\}$  can be extended to  $\{r, (r, r'), (r, r', r'')\}$ . The process can be repeated further, for each subdataflow found in the binding tree for  $R$ . The set  $AL$  of the final OPM graph will thus contain one alternate resulting from this process, its size bound by the depth of the binding tree for  $R$ .

### 4.3 Adding Subvalue Provenance to an OPM Graph

A major feature of the NRC dataflow model is that it can model the manipulation of complex data structures built as nested record and set constructions.

In the OPM mapping presented so far, the complex nesting structure of values is not yet represented. We can add this information by adding to the OPM graph a new account containing the structure information. Doing so also enables us to add “was derived from” edges in the OPM graph that track the *provenance* (or origin) of subvalues of complex data values.

Formal inference rules exist for the automatic generation of these provenance edges [2]. Since these inference rules are specific to the NRC dataflow model, and rely on the specific semantics of NRC operators on complex objects, they are not mere refinements of the OPM inference rules [1] for “was derived from” edges.

This is illustrated in Figure 9. The main account on top shows the OPM subgraph for a big union operation applied to the nested set  $\{\{1, 2\}, \{1, 3\}, \{3, 4\}\}$ . The nested value structure account is shown in dotted lines. The thick lines show two subvalue provenance edges that can be inferred by our provenance rules. The lines show that the value 1 produced by the big union operator comes from two different sets belonging to the nested set operated upon by the big union.

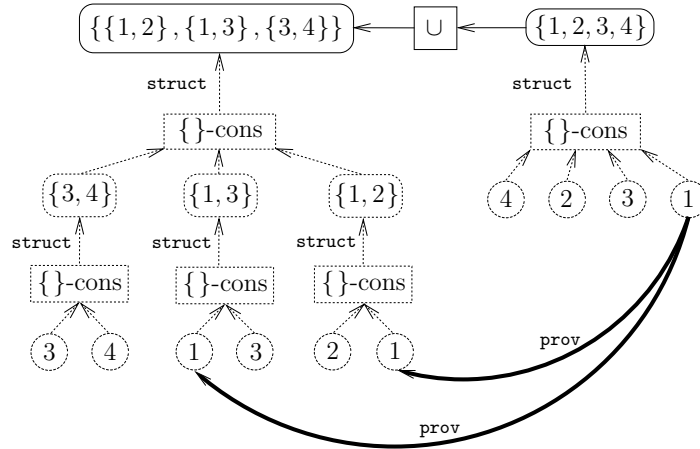


Fig. 9. Structure of complex values and provenance edges

## 5 Conclusion

We believe the NRC dataflow model is important, because it provides fully formal definitions of the complex interactions that occur in a repository consisting of many different executions of many different, interrelated dataflows involving complex data structures.

In order to validate the NRC dataflow model, we found it important to map it to the Open Provenance Model, as that model has been especially designed as an exchange framework for workflow provenance information. Our mapping also serves as a validation of the OPM.

It is interesting to explore further how the OPM mapping we have presented here can also serve as a basis for visualization of NRC dataflow runs. Of course we also have to design an implementation.

## References

1. Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J., Paulson, P.: The open provenance model. Technical Report 14979, University of Southampton, School of Electronics and Computer Science (2007)
2. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: A formal model of dataflow repositories. In Cohen-Boulakia, S., Tannen, V., eds.: DILS. Volume 4544 of Lecture Notes in Computer Science., Springer (2007) 105–121
3. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. *Theoretical Computer Science* **149** (1995) 3–48
4. Dumont, D., Noben, J., Raus, J., Stinissen, P., Robben, J.: Proteomic analysis of cerebrospinal fluid from multiple sclerosis patients. *Proteomics* **4**(7) (2004) 2117–2124