A Formal Model of Dataflow Repositories

Jan Hidders[1], Natalia Kwasnikowska[2], Jacek Sroka[3], Jerzy Tyszkiewicz[3], and Jan Van den Bussche[2]

[1] University of Antwerp, Belgium
[2] Hasselt University and Transnational University of Limburg, Belgium
[3] Warsaw University, Poland

Abstract. Dataflow repositories are databases containing dataflows and their different runs. We propose a formal conceptual data model for such repositories. Our model includes careful formalisations of such features as complex data manipulation, external service calls, subdataflows, and the provenance of output values.

1 Introduction

Modern scientific research is characterized by extensive computerized data processing of lab results and other scientific data. Such processes are often complex, consisting of several data manipulating steps. We refer to such processes as *dataflows*, to distinguish them from the more general *workflows*. General workflows also emphasize timing, concurrency, and synchronization aspects of a complex process, whereas in this paper we are less interested in such aspects, and our focus is mainly on data manipulation and data management aspects.

Important data management aspects of scientific dataflows include:

- Support of complex data structures, such as records containing different attributes of a data object, and sets (collections) of data objects. When combining, merging, and aggregating data, complex compositions of records and sets can arise.
- It must be possible to iterate operations over all members of a set.
- It must be possible to call external resources and services, like GenBank.
- Subdataflows must be supported, i.e., one dataflow can be used as a service in another dataflow.
- Dataflows must be specified in a clean, high-level, special purpose programming formalism.
- A dataflow can be run several times, often a large number of times, on different inputs.
- The data of these different runs must be kept, including input parameters, output data, intermediate results (e.g., from external services), and metadata (e.g., dates).

The last item above is of particular importance and leads to the notion of a *dataflow repository*: a database system that stores different dataflows together with their different runs. Dataflow repositories can serve many important purposes:

– Effective management of all experimental and workflow data that float around in a large laboratory or enterprise setting.
– Verification of results, either within the laboratory, by peer reviewers, or by other scientists who try to reproduce the results.
– Tracking the provenance (origin) of data values occurring in the result of a dataflow run, which is especially important when external service calls are involved.
– Making all data and stored dataflows available for complex decision support or management queries. The range of such possible queries is enormous; just to give two examples, we could ask "did an earlier run of this dataflow, using an older version of GenBank, also have this gene as a result?", or "did we ever run a dataflow in which this AA sequence was also used for a BLAST search?"

The idea of dataflow repository is certainly not new. It has been repeatedly emphasized in the database and bioinformatics literature, and practical dataflow systems such as Taverna [1] or Kepler [2] do accommodate many of the features listed above. What is lacking so far, however, is a formal, conceptual data model of dataflow repositories. This paper contributes towards this goal.

A conceptual data model for dataflow repositories should offer a precise specification of the types of data (including the dataflows themselves) stored in the repository, and of the relationships among them. Such a data model is important because it provides a formal framework that allows:

– Analyzing, in a rigorous manner, the possibilities and limitations of dataflow repositories.
– Comparing, again in a rigorous manner, the functionalities of different practical systems.
– Highlighting differences in meaning of common notions as used by different authors or in different systems, such as "workflow", "provenance", or "collection".

For the dataflow programming language, our model uses the nested relational calculus (NRC), enhanced with subtyping and external functions. NRC [3] is a well-studied language with exactly the right set of operations that are needed for the manipulation of the types of complex data that occur in a dataflow [4]. The suitability of NRC (in the form of a variant language called CPL) for scientific data manipulation and integration purposes has already been amply demonstrated by the Kleisli system [5,6]. We have confirmed this further by doing some case studies ourselves (e.g., of a proteomics dataflow [7]). A detailed report on several case studies of bioinformatics dataflows modeled using our formalism will be presented in a companion paper.

In this paper we provide formalisations of a number of fundamental notions related to dataflow repositories, such as:

– the notion of run of a dataflow;
– the provenance tracking of dataflow results;
– the binding of service names to external functions or to subdataflows; and
– the relationship between a run of a dataflow and the runs of its subdataflows.

2 Example

In this section we provide a simple example to illustrate different aspects involved in modelling of both dataflows and dataflow repository.

We begin by showing two dataflows, expressed in the nested relational calculus, for the following protocol: "Given two organisms A and B, extract all messenger RNA sequences from GenBank belonging to A. Then for each found sequence, search for similar sequences belonging to B."

dataflow *findSimilar*(A : **Organism**, B : **Organism**) : MatchedSeqs is
  $\bigcup$ for s in *entrez*(A, `genbank`) return
    if $s.moltype = \texttt{mRNA}$
      then $\{\langle a\colon s, b\colon \mathit{filter}(\mathit{blast}(s, \texttt{1e}-\texttt{4}), \texttt{300}, B)\rangle\}$
      else $\varnothing$

dataflow *filterBlastRep*(*rep* : **BlastRep**, *min* : **Int**, *org* : **Organism**) : Seqs is
  $\bigcup$ for a in *accDb*(*rep*, *min*) return
    let $seq := \mathit{getSeq}(a.accessionnr, a.database)$ in
      if $seq.organism = org$
        then $\{seq\}$
        else $\varnothing$

These dataflows use the following complex types:

MatchedSeqs = $\{\langle a\colon \text{Seq}, b\colon \text{Seqs}\rangle\}$,
Seqs = $\{\text{Seq}\}$,
Seq = $\langle \mathit{organism}\colon \textbf{Organism}, \mathit{moltype}\colon \textbf{MolType}, \mathit{content}\colon \textbf{NCBIXML}\rangle$,
AccNrDB = $\langle \mathit{accesionnr}\colon \textbf{AccessionNr}, \mathit{database}\colon \textbf{Database}\rangle$.

The dataflows also contain various service calls, with the following signatures:

entrez(*org*: **Organism**, *db*: **Database**): Seqs,
filter(*rep*: **BlastRep**, *score*: **Int**, *org*: **Organism**): Seqs,
blast(*seq*: Seq, *evalue*: **String**): **BlastRep**,
accDb(*rep*: **BlastRep**, *score*: **Int**): {AccNrDB},
getSeq(*acc*: AccNrDB): Seq.

Before we can execute the dataflows, we must bind the service names used to express service calls to actual services. We bind *entrez* and *blast* to external services provided by NCBI. We bind *filter* to dataflow *filterBlastRep*, which thus becomes a subdataflow of *findSimilar*. Now we have to bind all service names appearing in *filterBlastRep*, i.e., *accDb* and *getSeq*. We choose to bind both of them to some external service. The binding process stops here, as *filterBlastRep* does not have any subdataflows.

Suppose now that we have executed *findSimilar* with value `cat` for parameter A, and `mouse` for B. Suppose the following value has been returned:

$\{\langle a: \langle organism: \texttt{cat}, moltype: \texttt{mRNA}, ncbiXML: \texttt{AY800278}\rangle,$
$\quad b: \{\langle organism: \texttt{mouse}, moltype: \texttt{mRNA}, ncbiXML: \texttt{XM\_908677}\rangle,$
$\quad\quad \ldots,$
$\quad\quad \langle organism: \texttt{mouse}, moltype: \texttt{DNA}, ncbiXML: \texttt{NW\_042634}\rangle\}\rangle,$
$\ldots,$
$\langle a: \langle organism: \texttt{cat}, moltype: \texttt{mRNA}, ncbiXML: \texttt{NM\_001079655}\rangle,$
$\quad b: \{\langle organism: \texttt{mouse}, moltype: \texttt{mRNA}, ncbiXML: \texttt{NM\_053015}\rangle,$
$\quad\quad \ldots,$
$\quad\quad \langle organism: \texttt{mouse}, moltype: \texttt{DNA}, ncbiXML: \texttt{NT\_078297}\rangle\}\rangle\},$

where values like AY800278 are used as place holders for the corresponding XML documents. We denote this complex value by *finalresult*.

In our dataflow repository model, however, not just the final result value will be kept, but also information about the service calls that have happened during the run. Such information would look as follows:

$(entrez, [(A, \texttt{cat}), (B, \texttt{mouse}), (org, \texttt{cat}), (db, \texttt{genbank})], catseqs),$
$(blast, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_1), (seq, cat_1), (evalue, \texttt{1e}-4)], rep_1),$
$(filter, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_1),$
$\quad (rep, rep_1), (score, \texttt{300}), (org, \texttt{mouse})], foundcat_1),$
$(blast, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_2), (seq, cat_2), (evalue, \texttt{1e}-4)], rep_2),$
$(filter, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_2),$
$\quad (rep, rep_2), (score, \texttt{300}), (org, \texttt{mouse})], foundcat_2),$

Here, *catseqs* is a set containing the following tuples (among many others):

$$cat_1 = \langle organism: \texttt{cat}, moltype: \texttt{mRNA}, ncbiXML: \texttt{AY800278}\rangle,$$
$$cat_2 = \langle organism: \texttt{cat}, moltype: \texttt{mRNA}, ncbiXML: \texttt{NM\_001079655}\rangle.$$

Also, rep_i would be documents of type **BlastRep**, and, for instance, $foundcat_2$ would be a set containing the following tuples (among several others):

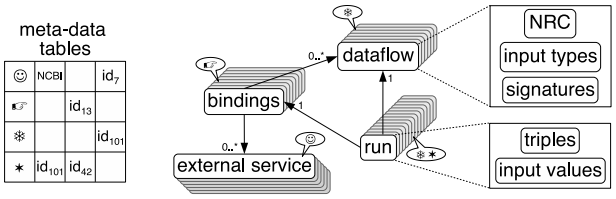
$$m_1 = \langle organism: \texttt{mouse}, moltype: \texttt{mRNA}, ncbiXML: \texttt{NM\_053015}\rangle,$$
$$m_2 = \langle organism: \texttt{mouse}, moltype: \texttt{DNA}, ncbiXML: \texttt{NT\_078297}\rangle.$$

Since all information needed to reconstruct the entire run is available, we can trace the provenance (origin) of a particular subvalue appearing in *finalresult*, say m_1. We produce a back-trace of the entire run by using subexpression occurrences and their respective input values, as follows:

$\{(\bigcup, [(A, \texttt{cat}), (B, \texttt{mouse})], m_1),$
$(\textsf{for}, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_2)], m_1),$
$(\textsf{if}, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_2)], m_1),$
$(\{\,\}, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_2)], m_1),$
$(\langle\rangle, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_2)], m_1),$
$(filter, [(A, \texttt{cat}), (B, \texttt{mouse}), (s, cat_2), (rep, rep_2), (min, \texttt{300}), (org, \texttt{mouse})], m_1)\}.$

Note that *filter* is bound to a subdataflow, so if desired, we can further track the provenance of m_1 in the corresponding run of *filterBlastRep*.

The main idea of a dataflow repository is that dataflows, as well as their runs, type declarations, input values, intermediate results, and subdataflow links, are all stored together in a database system. Here is an illustration of this idea:



The database should also contain meta-data stored in various additional tables. By meta-data we understand annotating information such as author of a dataflow, date of a run, or version of an external service. Through dataflow identifiers, run identifiers, and referential integrity, these meta-data tables are linked to the repository tables. In the above illustration, we use call-outs to represent these links.

3 Dataflows, Runs, and Provenance

In this section, we present the formal dataflow model. Due to space limitations, we will only be able to give a sample of the formal definitions, and all proofs of mathematical properties will be omitted.

Complex Values. We model the complex data structures occurring in a workflow using *complex values*. Complex values are constructed, using record and set constructions, from *base values*. Base values can be numbers or strings, but can also be XML files; it is essentially up to the application to decide which kinds of values are considered to be "atomic", and of which kinds of values we want to explicitly model the internal structure within the dataflow.

For example, consider the report returned by a BLAST search. One can consider the entire report as a base value, e.g., an XML file, and use an XQuery operation to extract information from it. This implies that the dataflow will model the XQuery operation as a single step: we will model such single steps by *service calls*. On the other hand, one can consider the structure of the report, modeled as a long record with various attributes, including a set of search results, and model this explicitly as a combination of record and set structures. Since our dataflow model includes the operations of the nested relational calculus, the process of extracting information from the report can be fully modeled in the dataflow. It always depends on the designer of the workflow application which data manipulation aspects of the dataflow need to be explicitly modeled, and which can be modeled as a single step: a good formal model should not enforce this choice in a particular direction.

Formally, we assume a given countably infinite set $\mathcal{A}$ of *base values*. To label attributes in *tuples* (records), we also need a countably infinite set $\mathcal{L}$ of

labels. Then the set $\mathcal{V}$ of *complex values* is the smallest set satisfying the following: $\mathcal{A} \subseteq \mathcal{V}$; if $v_1, \ldots, v_n \in \mathcal{V}$, then the finite set $\{v_1, \ldots, v_n\}$ is a complex value; if $v_1, \ldots, v_n \in \mathcal{V}$, and $l_1, \ldots, l_n \in \mathcal{L}$ are distinct labels, then the tuple $\langle l_1 \colon v_1, \ldots, l_n \colon v_n \rangle$ is also a complex value. The positioning of elements within a named tuple is arbitrary.

Note that we work with sets as the basic collection type, because other kinds of collections can be modeled as sets of records. For an ordered list, for example, one could use a numerical attribute that indicates the order in the list.

Complex Types. Types are a basic mechanism in computer programming to avoid the application of operations to inputs on which the operation is not defined. Thus, all data occurring in our dataflow model is strongly typed. We use a type system for complex objects with tuples and sets, known from database theory, that includes a form of subtyping.

Our type system starts from a finite set $\mathcal{B}$ of *base types*. Then the set $\mathcal{T}$ of *complex types* is the smallest set satisfying the following: $\bot \in \mathcal{T}$; $\mathcal{B} \subseteq \mathcal{T}$; if $\tau \in \mathcal{T}$, then the expression $\{\tau\}$ is also a complex type, called a *set type*; if $\tau_1, \ldots, \tau_n \in \mathcal{T}$, and $l_1, \ldots, l_n \in \mathcal{L}$ are distinct labels, then the expression $\langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle$ is also a complex type, called a *tuple type*. The positioning of elements within a tuple type is arbitrary.

The purpose of base types is obviously to organize the base values in classes. The purpose of $\bot$ is to have a generic type for the empty set; that type is the set type $\{\bot\}$. More generally, the semantics of types is that for each type τ we have a set $[\![\tau]\!]$ of *values of type* τ, defined in the obvious manner (omitted).

Reasons of flexibility require that the type system is equipped with a form of subtyping [8]. Base types provide an organization of the different types of base values into different classes, and it is standard to allow for classes and subclasses. For example, base types "Protein" and "Peptide" could be subclasses of a base type "AminoAcidSeq", which in turn could be a subtype of "BioSeq". Moreover, subtyping allows a flexible typing of if-then-else statements in dataflows. Thus, the type system of our dataflow model, while guaranteeing safe execution of operations, does not impede flexible specification of dataflows. Due to space limitations, however, we omit all details concerning subtyping.

Abstract Services. A common and general view of dataflows is that of a complex composition of atomic actions. In our model, the composition is structured using the programming constructs of the nested relational calculus (NRC). Moreover, the basic data manipulation operators of the NRC are already built in as atomic actions. Any further atomic actions are modeled in our formalism as service calls. Service calls can be really calls to external services, such as NCBI BLAST, but can also be calls to library functions provided by the underlying system, such as addition for numbers or concatenation for strings, or the application of an XQuery to an XML file. Moreover, one dataflow can appear as a service call in another dataflow, thus becoming its subdataflow.

In our model, dataflows use abstract *service names* to denote services. The type system requires *signatures* to be attached to these names. Only at the time a

dataflow needs to executed we provide meaning to the service names by assigning them *service functions*. In this section, service functions are merely abstract non-deterministic functions, as this is already sufficient to formally define a run of a dataflow. In Section 4, we will need to be more specific and distinguish between external services (or library functions) on the one hand, and subdataflows on the other hand.

Formally, a *signature* is an expression of the form $\tau_1, \ldots, \tau_n \to \tau_{\text{out}}$, where the τ's are types. Likewise, a *service function* is an (infinite) relation L from $[\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$ to $[\![\tau_{\text{out}}]\!]$, that is total in the sense that for any given values $v_1, \ldots, v_n$ of types $\tau_1, \ldots, \tau_n$ respectively, there must exist at least one value v_{out} of type τ_{out} such that $(v_1, \ldots, v_n, v_{\text{out}}) \in L$. We denote the universe of all possible signatures by $\mathcal{S}$, and that of all possible service functions by $\mathbb{F}$.

Service functions thus model the input-output behavior of services. Note that service functions can be non-deterministic, in that there may be more than one output related to a given input. This is especially important for modeling external services over which we have no control. The internal database of an on-line service (e.g., BLAST) may be updated, or the service may from time to time fail and produce an error value instead of the actual output value.

Note that we assume service functions to be total. For external services over which we have no control, or to model system failures, totality can always be guaranteed using wrappers. We also assume that wrappers take care of all compatibility issues between used services, as data integration aspects are beyond the scope of this paper.

The Nested Relational Calculus. NRC is a simple functional programming language [3], built around the basic operations on records and sets, with for-loops and if-then-else (and let-expressions) as the only programming constructs. We naturally augment NRC with service calls.

Formally, we assume countably infinite sets $\mathcal{X}$ of variables and $\mathcal{N}$ of service names. Then the *NRC expressions* are defined by the following BNF grammar:

Expr → *BaseExpr* | *CompositeExpr*
BaseExpr → *Constant* | *Variable* | "$\varnothing$"
CompositeExpr → "{" *Expr* "}" | *Expr* "$\cup$" *Expr* | "$\bigcup$" *Expr* |
"$\langle$" *Element* ("," *Element*)* "$\rangle$" | *Expr* "." *Label* |
"for" *Variable* "in" *Expr* "return" *Expr* |
Expr "=" *Expr* | *Expr* "= $\varnothing$" |
"if" *Expr* "then" *Expr* "else" *Expr* |
"let" *Variable* ":=" *Expr* "in" *Expr* |
ServiceName "(" *Expr* ("," *Expr*)* ")"
Element → *Label* ":" *Expr*
Constant → $\mathsf{a} \in \mathcal{A}$
Variable → $x \in \mathcal{X}$
Label → $l \in \mathcal{L}$
ServiceName → $f \in \mathcal{N}$

The variables in an expression that are introduced by a for- or a let-construct are said to occur *bound*; all other occurrences of variables in an expression constitute the *free variables* of an expression. For simplicity of exposition, we disallow that different for- or let-subexpressions bind the same variable. We also disallow that a free variable occurs bound at the same time. We denote the set of free variables of an expression e by $FV(e)$. Naturally, the free variables are the input parameters of the dataflow expressed by the expression. We also use the notation $SN(e)$ for the set of service names used in expression e.

When we want to run an expression, we need to assign input values to the free variables, and we need to assign service functions to the service names used in the expression. Formally, a *value assignment* is a mapping σ from $FV(e)$ to $\mathcal{V}$, and a *function assignment* is a mapping ζ from $SN(e)$ to $\mathbb{F}$. The evaluation of expressions is then defined by a system of rules (one rule for each construct of NRC) by which one can infer judgments of the form $\sigma, \zeta \models e \Rightarrow v$, meaning "value v is a possible final result of evaluating e on σ and ζ". Recall that there can be more than one possible final result value, if non-deterministic service functions are involved in the evaluation.

Since these inference rules are known from the literature [3], we just present a sample of them, using big union, if-then-else, and for-loops as examples. We also show the rule for service calls:

$$\frac{\sigma, \zeta \models e \Rightarrow \{v_1, \ldots, v_n\}}{\sigma, \zeta \models \bigcup e \Rightarrow v_1 \cup \cdots \cup v_n}$$

$$\frac{\sigma, \zeta \models e_1 \Rightarrow \{w_1, \ldots, w_n\} \qquad \forall i \in \{1, \ldots, n\} \colon add(\sigma, x, w_i), \zeta \models e_2 \Rightarrow v_i}{\sigma, \zeta \models \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \Rightarrow \{v_1, \ldots, v_n\}}$$

$$\frac{\sigma, \zeta \models e_1 \Rightarrow \mathtt{true} \qquad \sigma, \zeta \models e_2 \Rightarrow v}{\sigma, \zeta \models \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow v} \qquad \frac{\sigma, \zeta \models e_1 \Rightarrow \mathtt{false} \qquad \sigma, \zeta \models e_3 \Rightarrow v}{\sigma, \zeta \models \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow v}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \sigma, \zeta \models e_i \Rightarrow v_i \qquad (v_1, \ldots, v_n, w) \in \zeta(f)}{\sigma, \zeta \models f(e_1, \ldots, e_n) \Rightarrow w}$$

In the rule for for-loops, by $add(\sigma, x, w_i)$ we mean the value assignment obtained from σ by updating the value of x to w_i.

In order to guarantee that expression evaluation will not fail, we must typecheck the expression. The typechecker requires that we declare types for the free variables, and that we declare *signatures* for the service names. Formally, a *type assignment* for e is a mapping Γ from $FV(e)$ to $\mathcal{T}$, and a *signature assignment* is a mapping Θ from $SN(e)$ to $\mathcal{S}$. Typechecking is then defined by a system of rules (omitted due to space limitations) by which one can infer judgments of the form $\Gamma, \Theta \vdash e : \tau$, meaning "$e$ is well typed given Γ and Θ, with result type τ". The rules are such that there can be at most one possible result type for e given Γ and Θ.

The following property now states that the type system assures safe execution of expressions.

Property 1. If $\Gamma, \Theta \vdash e : \tau$ can be inferred, and σ and ζ are value and function assignments consistent with Γ and Θ, then there always exists a value v of type τ such that $\sigma, \zeta \models e \Rightarrow v$ can be inferred.

Runs. In a dataflow repository, we want to keep the information about the different runs we have performed of each dataflow. For this, it is not sufficient to just keep the input values. Indeed, if external services are called in the dataflow, merely rerunning the dataflow on the same inputs may not produce the same result as before, because the behavior of the external service may have changed in the meantime, or because it may even fail this time. It is also not sufficient to keep only the final result value of every run in addition to the input values. Indeed, the repository should support provenance tracking of output values, by which the system can show how certain output values were produced during the dataflow execution. Again, as before, merely rerunning the dataflow will not do here.

We conclude that it is necessary to keep, for each run of an expression e, the information about the service calls that have happened during the run. We can naturally represent this information as a number of triples of the form (e', σ', v'), where: e' is a service call subexpression of e; σ' is the value assignment constituting the input values of the service call; and v' is the output produced by the service call. Note that there can be many such triples, even if e contains only one service call subexpression, because that service call may occur inside a for-loop.

From that information, the *entire* run can then be reconstructed. We can represent the entire run equally well as a set of such triples, where now e' is not restricted to just service calls, but where we consider all subexpressions instead.[4] Specifically, we have defined a new system of inference rules (one rule for each construct of NRC) that allow to infer judgments of the form $\sigma, \zeta \approx e \Rightarrow R$, meaning that R is a possible run of e on σ and ζ. Recall that service functions may be non-deterministic, so that for the same value and function assignments, there may be several different runs. The rules also define the final result value of the run. Moreover, because we will need this for provenance tracking, our rules define the set of *subruns* of a run R — these are runs of subexpressions of e that happened as part of R. Formally, each subrun is represented by a triple of the form (e', σ', R'), where e' is a subexpression of e and $\sigma', \zeta \approx e' \Rightarrow R'$ holds.

Like before, we only show a sample of the rules:

$$\frac{e = \bigcup e' \qquad \sigma, \zeta \approx e' \Rightarrow R' \qquad v = \bigcup \mathit{result}(R') \qquad R = R' \cup \{(e, \sigma, v)\}}{\sigma, \zeta \approx e \Rightarrow R \qquad \mathit{result}(R) \stackrel{def}{=} v \qquad \mathit{Subruns}(R) \stackrel{def}{=} \mathit{Subruns}(R') \cup \{(e, \sigma, R)\}}$$

[4] For simplicity of exposition, in the present version of this paper, we will ignore the complication that a subexpression may have several different occurrences in an expression. We know how to incorporate this in the formalism.

$$\frac{\begin{array}{c} e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \qquad \sigma, \zeta \models\!\approx e_1 \Rightarrow R' \\ \mathit{result}(R') = \{w_1, \ldots, w_n\} \qquad \forall i \in \{1, \ldots, n\} \colon \mathit{add}(\sigma, x, w_i), \zeta \models\!\approx e_2 \Rightarrow R_i \\ v = \{\mathit{result}(R_1), \ldots, \mathit{result}(R_n)\} \qquad R = R' \cup R_1 \cup \cdots \cup R_n \cup \{(e, \sigma, v)\} \end{array}}{\begin{array}{c} \sigma, \zeta \models\!\approx e \Rightarrow R \qquad \mathit{result}(R) \stackrel{\mathit{def}}{=} v \\ \mathit{Subruns}(R) \stackrel{\mathit{def}}{=} \mathit{Subruns}(R') \cup \mathit{Subruns}(R_1) \cup \cdots \cup \mathit{Subruns}(R_n) \cup \{(e, \sigma, R)\} \end{array}}$$

$$\frac{\begin{array}{c} e = f(e_1, \ldots, e_n) \qquad \forall i \in \{1, \ldots, n\} : \sigma, \zeta \models\!\approx e_i \Rightarrow R_i \\ (\mathit{result}(R_1), \ldots, \mathit{result}(R_n), v) \in \zeta(f) \qquad R = R_1 \cup \cdots \cup R_n \cup \{(e, \sigma, v)\} \end{array}}{\begin{array}{c} \sigma, \zeta \models\!\approx e \Rightarrow R \qquad \mathit{result}(R) \stackrel{\mathit{def}}{=} v \\ \mathit{Subruns}(R) \stackrel{\mathit{def}}{=} \mathit{Subruns}(R_1) \cup \cdots \cup \mathit{Subruns}(R_n) \cup \{(e, \sigma, R)\} \end{array}}$$

Let us explain the rule for the flatten expression $e = \bigcup e'$. We see that, in order to be able to derive a possible run R of e on given σ and ζ, we must first derive a possible run R' for e' on σ and ζ. From this particular R', we construct a final result value v for e, and a run R of which v is the final result value. This R is one of the possible runs of e on σ and ζ, in particular the one that has R' as its subrun. Therefore all subruns of R' are also subruns of R.

The run inference rules have the following property.

Property 2. Given a run R, for each subexpression e' and each σ' there is at most one R' such that $(e', \sigma', R') \in \mathit{Subruns}(R)$. We denote this run R' by $\mathit{Subrun}(e', \sigma', R)$.

Provenance. We are now ready to consider provenance tracking. We define provenance tracking *for any occurrence of a subvalue* of the final result value of a run. The following simple example will illustrate what we mean by subvalue occurrences. Consider the simple expression $e = \langle a : x, b : f(5)\rangle$, where we declare x to be of type int, and assign the signature int $\rightarrow$ int to service name f. Suppose now that we run e on the value assignment where $x = 3$, and on a function assignment ζ by which $(5, 3) \in \zeta(f)$. Then the tuple $\langle a : 3, b : 3\rangle$ is a final result value of e. Note that 3 occurs twice as a subvalue in this result, but both occurrences have a quite different provenance: the first occurrence is simply a copy of the input value $x = 3$, whereas the second occurrence was produced by the service call $f(5)$.

Formally, we define a *subvalue path* of some complex value v as a path from the root in v, viewing v as a tree structure in the obvious manner. Space limitations prevent us from giving the detailed definition. We will use the notation $\varphi \leftarrow\!\bullet\ v$ to denote that φ is a subvalue path of v. Note that if v is a set value, and φ is not just v itself, i.e., φ leads to a proper subvalue, then φ is of the form $v; \varphi'$, with $\varphi' \leftarrow\!\bullet\ u$ for some $u \in v$. We will use that observation in the inference rules below.

Indeed, we have designed a new system of inference rules that defines, for any run R, the provenance $\mathit{Prov}(\varphi, R)$ for any subvalue path φ in $\mathit{result}(R)$. Intuitively, the provenance is the restriction of R to all subexpressions and subvalues of intermediate results that have contributed to the production of φ in R. Formally, considering that R is a set of triples of the form (e', σ', v'), we will

define $Prov(\varphi, R)$ as a set of triples of the form (e', σ', φ'), where φ' is a subvalue path of v'. Intuitively, such a triple represents the information that the intermediate result v' (resulting from an evaluation of the subexpression e') has partly contributed to φ in the output—φ' then indicates which part.

We give a sample of the provenance inference rules next.

$$\frac{\begin{array}{c} e = e'.l \qquad \sigma, \zeta \models e \Rightarrow R \\ v = result(R) \qquad \varphi \longleftarrow\!\bullet\ v \qquad S = Subrun(e', \sigma, R) \qquad v' = result(S) \end{array}}{Prov(\varphi, R) \stackrel{def}{=} Prov(v'; l; \varphi, S) \cup \{(e, \sigma, \varphi)\}}$$

$$\frac{\begin{array}{c} e = \langle l_1 \colon e_1, \ldots, l_n \colon e_n \rangle \qquad \sigma, \zeta \models e \Rightarrow R \qquad v = result(R) \\ i \in \{1, \ldots, n\} \qquad S = Subrun(e_i, \sigma, R) \qquad \varphi = v; l_i; \varphi' \qquad \varphi' \longleftarrow\!\bullet\ result(S) \end{array}}{Prov(\varphi, R) \stackrel{def}{=} Prov(\varphi', S) \cup \{(e, \sigma, \varphi)\}}$$

$$\frac{\begin{array}{c} e = \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \qquad \sigma, \zeta \models e \Rightarrow R \\ v = result(R) \qquad \varphi \longleftarrow\!\bullet\ v \qquad result(Subrun(e_1, \sigma, R)) = \mathtt{true} \end{array}}{Prov(\varphi, R) \stackrel{def}{=} Prov(\varphi, Subrun(e_2, \sigma, R)) \cup \{(e, \sigma, \varphi)\}}$$

$$\frac{\begin{array}{c} e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \qquad \sigma, \zeta \models e \Rightarrow R \qquad v = result(R) \\ w = result(Subrun(e_1, \sigma, R)) \qquad \forall w' \in w \colon S_{w'} = Subrun(e_2, add(\sigma, x, w'), R) \\ \varphi = v; \varphi' \qquad \varphi' \longleftarrow\!\bullet\ u \qquad u \in v \end{array}}{Prov(\varphi, R) \stackrel{def}{=} \bigcup_{\{\, w' \in w \mid result(S_{w'}) = u \,\}} Prov(\varphi', S_{w'}) \cup \{(e, \sigma, \varphi)\}}$$

$$\frac{\begin{array}{c} e = \bigcup e' \qquad \sigma, \zeta \models e \Rightarrow R \qquad v = result(R) \\ S = Subrun(e', \sigma, R) \qquad w = result(S) \qquad \varphi = v; \varphi' \qquad \varphi' \longleftarrow\!\bullet\ u \qquad u \in v \end{array}}{Prov(\varphi, R) \stackrel{def}{=} \bigcup_{\{\, w' \in w \mid u \in w' \,\}} Prov(w; w'; \varphi', S) \cup \{(e, \sigma, \varphi)\}}$$

The rule for tuple field selection delegates the provenance to the immediate subexpression. Note that the rule for tuple construction includes only information from the subrun of the subexpression corresponding to the tuple field in which the subvalue path φ occurs. The rule for if-then-else (only given for the then-case) is similar in this respect; only the then-branch is tracked. The rule for for-loops shows how provenance is tracked in all subruns that contributed a value in which φ occurs. The rule for big union is again similar in this respect.

4 Binding Trees

In a dataflow repository, different dataflows are stored together with their runs. An important feature is that the same dataflow may have been run several times, on distinct inputs (value assignments), but also with different function assignments. Recall that a function assignment binds the service names occurring in the dataflow expression to concrete service functions. While some of the service names will be bound to external functions, in a dataflow repository, it should

also be possible to use an existing dataflow as the functionality of some service name. In other words, one dataflow can be used as a "subdataflow" of another. A complication now is that subdataflows may in turn contain service names, so those must be bound as well. In order to avoid non-terminating executions, we must pay attention not to create cycles in this binding process. This is taken care by the notion of *binding tree*, which we formally introduce in the present section.

Formally, consider a set D of *dataflow identifiers*. Each dataflow id has an associated NRC-expression that serves as the dataflow expression. Formally, this corresponds to a given mapping $expr : D \to \text{NRC}$. Moreover, consider a set Ext of *external service identifiers*. We now define:

Definition 1. *A* binding tree *is a finite tree, where the nodes are labeled with dataflow identifiers or external service identifiers, and the edges are labeled with service names, with the following properties:*

- *The root is labeled with a dataflow identifier.*
- *Only leaves can be labeled with external service identifiers.*
- *Suppose a node x is labeled with a dataflow identifier d. Let $f_1, \ldots, f_n$ be the different service names used in $expr(d)$. Then x has precisely n children, with edges labeled by $f_1, \ldots, f_n$, respectively.*

Intuitively, a binding tree specifies, for the dataflow mentioned in the root, which service names in the dataflow expression are bound to external services, and which to subdataflows. For these subdataflows, the binding tree again specifies a binding for their own service names, and so on. Indeed, note that in a binding tree, a subtree rooted in a node labeled with a dataflow id is itself a binding tree. Note also that the same dataflow id can appear several times in a binding tree (and with different binding subtrees), and that also the same external service name can appear several times.

In order to define this formally, to begin with, we need an assignment of service functions to external service identifiers, i.e., a mapping $func : Ext \to \mathbb{F}$. We can then define the function assignment specified by a binding tree by induction on the height of the tree:

Definition 2. *Let β be a binding tree. Let the root of β be labeled with d, and let $expr(d) = e$. We define a function assignment ζ_β for e as follows. Let h be the height of β.*

- *If $h = 0$, then ζ_β is empty.*
- *If $h > 0$, then $\zeta_\beta(f)$, for any service name f used in e, is defined as follows. Let x be the f-child of the root of β.*
  - *If x is labeled with an external service id z, then we define $\zeta_\beta(f) := func(z)$.*
  - *If x is labeled with a dataflow id d', then let $e' = expr(d')$, and consider the subtree β' of β rooted at x. By induction, we already have a function assignment $\zeta'_{\beta'}$ for e'. Then we define $\zeta_\beta(f)$ to be the relation that associates input value assignments for e' to final result values, given $\zeta'_{\beta'}$.*

In the above, due to space limitations, we have ignored the signatures of service names and of external service identifiers. Incorporating these signatures requires that we enrich a binding tree with mappings that associate parameter positions of service calls to free variables of subdataflow expressions.

5 Repository Data Model

We are now in a position to give a formal definition of a dataflow repository. A conceptual schema illustrating the different entities that play a role in a repository, and their relationships, is given in Fig. 1. We use the following notation: $\mathbb{G}$ for all possible type assignments; $\mathbb{S}$ for all possible value assignments; *Runs* for all possible runs; $\mathbb{T}$ for all possible signature assignments; $\mathbb{B}$ for the set of all possible binding trees; and *Triples* for all possible triples in runs. Also, as in the previous section, we assume a given set *Ext* of external service identifiers and a mapping $\mathit{func}\colon \mathit{Ext} \to \mathbb{F}$. (External service identifiers also have signatures, but we ignore these due to space limitations.)

Definition 3. *A dataflow repository consists of two finite, pairwise disjoint sets D and R, whose elements are called* dataflow identifiers *and* run identifiers*, respectively, together with eight mappings of the following signatures:*

$$\begin{aligned} \mathit{expr}\colon D &\to \mathcal{E} & \mathit{inputtypes}\colon D &\to \mathbb{G} \\ \mathit{servicesigs}\colon D &\to \mathbb{T} & \mathit{dataflow}\colon R &\to D \\ \mathit{inputvals}\colon R &\to \mathbb{S} & \mathit{binding}\colon R &\to \mathbb{B} \\ \mathit{run}\colon R &\to \mathit{Runs} & \mathit{internalcall}\colon R \times \mathit{Triples} &\to R \end{aligned}$$

The first seven mappings are standard, total, many-to-one mappings; the last mapping, however, is partial but must be one-to-one.

Moreover, the mappings must satisfy the following integrity constraints, for any $d \in D$ and any $r \in R$:

- *$\mathit{inputtypes}(d)$ is defined on $FV(\mathit{expr}(d))$.*
- *$\mathit{servicesigs}(d)$ is defined on $SN(\mathit{expr}(d))$.*
- *$\mathit{expr}(d)$ is well-typed under $\mathit{inputtypes}(d)$ and $\mathit{servicesigs}(d)$.*
- *$\mathit{inputvals}(r)$ is defined on $FV(\mathit{expr}(\mathit{dataflow}(r)))$, and is compatible with $\mathit{inputtypes}(\mathit{dataflow}(r))$.*
- *The root of $\mathit{binding}(r)$ is labeled with $\mathit{dataflow}(r)$.*
- *$\mathit{run}(r)$ is a run of $\mathit{expr}(\mathit{dataflow}(r))$ on $\mathit{inputvals}(r)$, given $\zeta_{\mathit{binding}(r)}$.*
- *The repository is* closed *by the mapping internalcall.*

We still have to explain the last item in the above definition (closure). Closure is an important integrity constraint that corresponds to the following intuition: if the repository contains a run of some dataflow, then it also contains all corresponding runs of its subdataflows. (Note that if a subdataflow is inside a for-loop, the subdataflow may be run several times.) This is precisely the function of the mapping *internalcall*, which given a run and a call in that run to a service name bound to a subdataflow, will indicate the run identifier of the corresponding subdataflow run. Formally, we define:

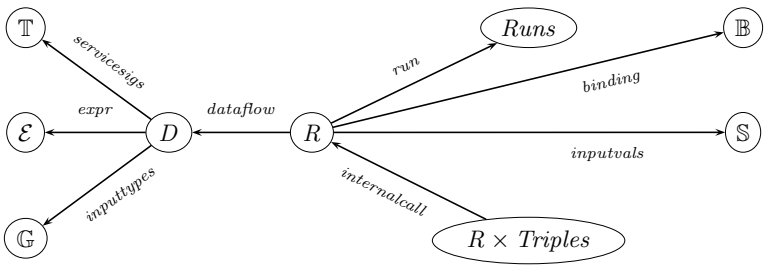


Fig. 1. Conceptual dataflow repository schema.

Definition 4. *A repository is* closed *by internalcall if for any $r \in R$ and any $t = (\Phi, \sigma, v) \in Triples$, the following holds:*

- *$internalcall(r, t)$ is defined if and only if $t \in run(r)$ and Φ is (an occurrence of) a service call to some service named f such that the f-child of the root of $binding(r)$ is labeled with a dataflow identifier d'.*
- *If $internalcall(r, t) = r'$ is indeed defined, then*
  - *$dataflow(r') = d'$;*
  - *$binding(r')$ equals the subtree of $binding(r)$ rooted in the f-child of the root of $binding(r)$;*
  - *$inputvals(r') = \sigma$; and*
  - *the final result value of $run(r')$ equals v.*

Note that we do not explicitly model meta-data in the repository data-model. However, it is possible to extend the conceptual data model with meta-data, for instance by adding mappings from various entities in the repository to annotation identifiers, which represent diverse meta-data entities. The actual content of meta-data is beyond the scope of this paper.

6 Related Work

Several researchers advocate integration of workflows and DBMSs [9,10,11], as they provide mechanisms for planning, scheduling, and logging. We believe that to properly design a dataflow repository, you need a formal model for dataflows and runs. Although there are several dataflow specification languages [9,12,13,1,2], to our knowledge, none of them presents a formal model of repository storing dataflows and runs. With increasing importance of provenance [14,15,16,17], often with different interpretations for this term, it is essential that our model includes a formal definition of the kind of provenance that our work targets. For instance, our notion of provenance largely covers the queries of the Provenance Challenge (http://twiki.ipaw.info/bin/view/challenge/). Pioneers in integration of workflows and DBMSs are ZOO [9] and OPM [10]. The ZOO system, implemented on top of an OODBMS, uses database schemas to model dataflows: object classes model data, and relations between them model operations (associated rules specify their execution). An instance contains run information. OPM

uses schemas for workflow design with separate object and protocol classes. Protocol classes employ attributes for input, output and connections, and constrains are used to enforce various rules. OPM is implemented in RDBMS, and runs are stored as instances of relational schemas. More recently, Shankar et al. [11] have proposed dataflow specification integrated with SQL and relational DBMS. Dataflows are modeled as active relational tables, and invoked through SQL queries. The Taverna [1] workflow system focuses on practical workflow design and integration of bioinformatics tools and databases. They store runs and associated meta-data in a provenance store implemented as a Web Service et al. [16]. There are also systems with dataflow design repositories, e.g., WOODSS [18], mainly focusing on workflow reuse. Tröger et al. [12] present a language for workflow design, similar to *in vitro* experiments. Although the compiler produces a persistent repository of workflow specifications and meta-data, it does not include (intermediate) results. Another well-known workflow system is Kepler [2]. Workflow design is actor-oriented and supports collections through an abstract data model for actor design [19]. Intermediate results are recorded through automatic report generation.

7 Towards a Dataflow Repository System

A dataflow repository system, following the conceptual model presented in this paper, could be implemented in various ways. An approach that seems promising to us, and which is the object of our current work, is to build the system on top of a modern relational DBMS using SQL/PSM and SQL/XML. A similar approach was also advocated by Shankar et al. [11]. Base values are implemented using SQL datatypes; more complicated base types such as NCBIXML can be implemented using the XML column type, or as large objects (LOBs). Complex values can be decomposed into tables using standard techniques. NRC expressions can be compiled into SQL procedures that, when run, will insert not only the final result value in the repository, but also the intermediate results of external service calls. Service calls can be implemented using SQL user-defined functions. The conceptual data model of the dataflow repository is readily mapped to the relational data model. All semi-structured data belonging to the repository, such as NRC expressions, type assignments, signatures, or binding trees, can be stored using XML columns.

Last but not least, the database may include various additional tables, which contain meta-data such as author of a dataflow, date of a run, version of an external databases, etc. Through dataflow identifiers, run identifiers, and referential integrity, these tables are linked to the repository tables.

8 Conclusions

In this paper we have presented an attempt to lay the formal groundwork of dataflow repository systems. Now that we have a precise specification of the various data stored in such repositories, we can start envisaging ways of querying

all this data. Note that computing provenance information can already be considered as a kind of query computed over a single run stored in the repository. But clearly much more is possible, given that many different dataflows, with many different runs, are in the database. Two examples of potential decision support queries were already given in the introduction. It remains to be investigated whether special-purpose query language mechanisms must be designed, or whether SQL/XML, where XQuery and SQL can be freely combined, provides enough flexibility and expressive power.

References

1. Oinn, T. et al.: Taverna: A tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17) (2004) 3045–3054
2. Ludäscher, B. et al.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice And Experience **18**(10) (2006) 1039–1065
3. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theor. Computer Science **149** (1995) 3–48
4. Stevens, R., Goble, C., Baker, P., Brass, A.: A classification of tasks in bioinformatics. Bioinformatics **17**(1) (2001) 180–188
5. Chen, J., Chung, S.-Y., Wong., L.: The Kleisli query system as a backbone for bioinformatics data integration and analysis. In Bioinformatics: Managing Scientific Data. Morgan Kaufmann (2003) 147–187
6. Davidson, S. et al.: The Kleisli approach to data transformation and integration. In The Functional Approach to Data Management. Springer (2004) 135–165
7. Gambin, A., Hidders, J., Kwasnikowska, N. et al.: NRC as a formal model for expressing bioinformatics workflows. Poster at ISMB 2005, Detroit, MI, USA
8. Pierce, B.: Types and Programming Languages. The MIT Press (2002)
9. Ailamaki, A., Ioannidis, Y., Livny, M.: Scientific workflow management by database management. Proceedings of SSDBM July 1998, IEEE Computer Society, 190–199
10. Chen, I., Markowitz, V.: An overview of the object protocol model (OPM) and the OPM data management tools. Information Systems **20**(5) (1995) 393–418
11. Shankar, S., Kini, A., DeWitt, D., Naughton, J.: Integrating databases and workflow systems. SIGMOD Record **34**(3) (2005) 5–11
12. Tröger, A. et al.: A language for comprehensively supporting the *In Vitro* experimental process *In Silico*. Proceedings of BIBE March 2004, IEEE Computer Society, 47–56
13. Zhao, Y. et al.: A notation and system for expressing and executing cleanly typed workflows on messy scientific data. SIGMOD Record **34**(3) (2005) 37–43
14. Cohen, S., Cohen Boulakia, S., Davidson, S.: Towards a model of provenance and user views in scientific workflows. Proceedings of DILS 2006, LNCS **4075** 264–279
15. Bose, R., Frew, J.: Lineage retrieval for scientific data processing: A survey. ACM Computing Surveys **37**(1) (2005) 1–28
16. Wong, S., Miles, S., Fang, W. et al.: Provenance-based validation of e-science experiments. Proceedings of ISWC 2005, LNCS **3729** 801–515
17. Mutsuzaki, M. et al.: Trio-One: Layering uncertainty and lineage on a conventional DBMS. Proceeding of CIDR Januari 2007, Asilomar, California
18. Medeiros, C. et al.: WOODSS and the Web: annotating and reusing scientific workflows. SIGMOD Record **34**(3) (2005) 18–23
19. McPhillips, T. et al.: Collection-oriented scientific workflows for integrating and analyzing biological data. Proceedings of DILS 2006, LNCS **4075** 248–263