



## Article

# LabelFlow Framework for Annotating Workflow Provenance

Pinar Alper <sup>1,\*</sup> , Khalid Belhajjame <sup>2</sup>, Vasa Curcin <sup>3</sup>  and Carole A. Goble <sup>4</sup><sup>1</sup> Luxembourg Centre for Systems Biomedicine, University of Luxembourg,  
L 4365 Esch-sur-Alzette, Luxembourg<sup>2</sup> LAMSADE Research Lab, Université Paris Dauphine, UMR CNRS 7243 Paris, France;  
Khalid.Belhajjame@dauphine.fr<sup>3</sup> Department of Population Health Sciences, King's College London, London SE1 1UL, UK;  
vasa.curcin@kcl.ac.uk<sup>4</sup> School of Computer Science, University of Manchester, Manchester M13 9PL, UK;  
carole.goble@manchester.ac.uk

\* Correspondence: pinar.alper@uni.lu

Received: 28 November 2017; Accepted: 21 February 2018; Published: 23 February 2018

**Abstract:** Scientists routinely analyse and share data for others to use. Successful data (re)use relies on having metadata describing the context of analysis of data. In many disciplines the creation of contextual metadata is referred to as *reporting*. One method of implementing analyses is with workflows. A stand-out feature of workflows is their ability to record *provenance* from executions. Provenance is useful when analyses are executed with changing parameters (changing contexts) and results need to be traced to respective parameters. In this paper we investigate whether provenance can be exploited to support reporting. Specifically; we outline a case-study based on a real-world workflow and set of reporting queries. We observe that provenance, as collected from workflow executions, is of limited use for reporting, as it supports queries partially. We identify that this is due to the generic nature of provenance, its lack of domain-specific contextual metadata. We observe that the required information is available in implicit form, embedded in data. We describe *LabelFlow*, a framework comprised of four *Labelling Operators* for decorating provenance with domain-specific *Labels*. *LabelFlow* can be instantiated for a domain by plugging it with domain-specific metadata extractors. We provide a tool that takes as input a workflow, and produces as output a *Labelling Pipeline* for that workflow, comprised of *Labelling Operators*. We revisit the case-study and show how *Labels* provide a more complete implementation of reporting queries.

**Keywords:** workflow; provenance; domain-specific annotation

## 1. Introduction

Computational data analysis has become an inseparable part of today's scientific practice [1]. The research ecosystem is now richer, where, in addition to the traditional accumulation and consumption of scientific knowledge as literature, we observe the accumulation and consumption of scientific data [2]. Scientists have new responsibilities in this data-rich environment; they need to devise ways to (1) deal with the data deluge and analyse data in automated yet transparent ways (2) share the data produced so that others can re-use it in follow-on studies. Metadata plays a key role in both avenues.

On the side of data sharing, metadata is needed for data discovery and interpretation [3]. Of particular importance is metadata describing the *context of a scientific study* that produces the data. Context commonly refers to:

- Study Subjects; such as the species, the stellar objects or the geographical regions that the data is about.
- Study Factors; these correspond to controlled variations within the research method used. One example could be the space of parameter values for a simulation. Another could be the different diets fed to lab mice prior to data collection.
- Data Origins; if the data is generated first hand from lab/field work, then origin often corresponds to certain attributes of study subjects, e.g., the gender and age of humans from which DNA has been sampled. If data is derived from already existing datasets then origin corresponds to attributes such as, data catalog names, versions, or access end-points.

In order to provision this contextual metadata, in recent years, the practice of *reporting* has emerged. Several disciplines have introduced guidelines [4] and models [5] to ensure that core contextual metadata can be created. Reporting can describe both physical and computational parts of studies. One key characteristic of reporting is its emphasis on fine-grained modelling of context. A study may embody multiple contexts involving multiple subjects, factors and alternate source datasets. In such cases reporting requires that each output should carry as metadata the distinct context that has led to its creation. As of today, reports primarily describe physical parts of studies and the metadata is manually curated by scientists.

On the side of data analysis scientists are equipped with diverse tools for automation. Scientific workflow engines [6–9], scripting platforms [10,11] are examples of such tools. Workflow engines in particular are favoured as they enable process transparency in addition to automation. Workflows make analytical processes explicit by modelling them as a network of analysis tasks and dataflow dependencies among tasks. Furthermore workflow systems automatically collect metadata, called *workflow provenance*, from analysis runs. Workflow provenance is also a network of task executions and the data consumed/produced by tasks. The data derivation paths within workflow provenance are commonly referred to as *lineage*. Lineage becomes particularly important when workflows encode iterated analyses ran over a large permuted set of parameters. In such cases lineage links analysis outputs to corresponding parameters [12]. Workflow systems exploit lineage to streamline workflow executions, through debugging, error/change propagation.

Against this outlook, we can see that on the one hand side there is a requirement, and on the other side there is a supply of metadata. As of today these two kinds of metadata are at a disconnect; meaning, workflow provenance is minimally exploited for reporting. As a result the reporting of computational data analyses remains adhoc. Data is either left unannotated or superficially annotated, where some high-level characteristics of the analysis design are described, yet the context(s) that arise in analysis execution are left out. Adhoc reporting relies on the fact that context information is implicitly available in file names, file headers or in data values.

The reasons for the metadata disconnect are two fold:

- Workflow provenance is generic. In order to allow processing of diverse scientific datasets using equally diverse tools; workflows systems are designed to be oblivious to what data and tasks internally represent. This is also known as the “black-box approach”, which generates provenance graphs comprised of opaque nodes [13]. Meanwhile, for reporting, we require domain-specific information on data and tasks. Henceforth workflow provenance requires further annotation in order to be useful in reporting.
- Workflows as automation artefacts proliferate data generation. A workflow is rarely run once, typically ran several times to explore the effects of parameter or input changes on outputs. As a result manual annotation of workflow outputs can be a daunting task for scientists. In a recent survey scientists have stated that they receive ample automation support for performing the analyses, much less so for post-analysis activities such as result management, annotation and sharing [14].

Past research has investigated the annotation of provenance graphs [15,16]. However, the requirements of reporting bring fresh challenges, which reveal gaps not addressed with state of the art techniques.

Early approaches have focused on manually annotating a workflow's design to denote fixed characteristics of tasks or task data. Design annotations can state, for example, that a task (hence consequently all its executions) consume a *genomic sequence* and perform *sequence alignment*. Meanwhile reporting requires dynamic characteristics that surface only at runtime, such as, the specific species/subject whose genome is sequenced, or the values of parameters used during alignment. In [17] Ailamaki et al. observe that automated processing of this category dynamic metadata is a crucial requirement for scientific data management. Recently we observe that acquisition of runtime attributes as metadata and their propagation is being added as a feature to workflow systems, specifically in the Galaxy [7] and Wings [8]. This feature relies entirely on the user's manual tagging of each individual input dataset. In the case of Galaxy, tags get propagated to all descendants of data regardless of the nature of data processing in individual steps. In the case of Wings in addition to annotating datasets the user is expected to configure elaborate rules on how metadata may propagate from inputs to outputs, as a result we have not observed a use of this feature in existing Wings workflows [18].

Scientific disciplines have standardised data formats, where data is accompanied with embedded contextual metadata. VOTable, HDF5, FITS in earth-observation and astronomy [17]; BAM, VCF, FASTQ, Blast Report in life science [19] are examples of such formats. The state of the art exploits data formats in two major ways. First one is raw-data processing tools such as NoDB [20] and Fastbit [21], which utilise various indexing techniques so that data can be kept in its raw form but can be loaded and queried when necessary. These approaches assume a vertical column-oriented organisation for data, where some of the columns may comprise metadata. The first drawback here is that not all scientific formats are column-oriented, in many cases metadata is often found in a dedicated file header. The second aspect is that these tools focus on single files and they are unaware of the dataflow of a scientific analysis. Data in one file rarely refers to data in other files, and recovering this information without the explicit dataflow is a research question on its own [22]. In our setting we have the explicit dataflow (lineage) in the workflow description and provenance, and this information needs to be augmented with information extracted from data. ARMFUL [23] is a recent work in the area of scientific workflows that investigates how raw data indexing can be combined with dataflow information. This approach assumes data-parallelism in workflows where an analysis is mapped onto an input file collection, producing a corresponding file collection, and the raw data indexing is applied to files. Rich fine-grained metadata can be generated and queried in combination with the dataflow. This approach is very close in spirit to *LabelFlow*, however, it is tightly coupled with positional indexing (that of FastBit et cetera) and therefore devotes itself entirely to tabular data formats. Furthermore a comprehensive empirical survey [24] have revealed that workflows are largely comprised of data grooming/adaptation steps, where during adaptation, data is stripped of its standard formatting and several data copies can be created, furthermore data granularity can change throughout the workflow where in one step the workflow may perform analyses on individual items, whereas in the next step it may perform one analysis/adaptation on the entire collection. Therefore, it is necessary that annotation takes into account the reality of (1) data adaptation and the fact that data may not always carry embedded metadata, and (2) the collection-item-level nature of analyses.

Finally, in past work, computational "lineage" or dataflow has been used synonymously with "origin", meaning, all data that is on the lineage of a result is considered to be the origin of that result [25]. In the context of scientific reporting origin has a more restricted interpretation than computational lineage.

In this paper, we hypothesise the following: The scientific context found in an embedded, implicit form (in data formats, file names), can be made explicit as annotations over data. Prior empirical evidence has shown that workflow tasks belong to predetermined functional categories [24]. We argue that such a categorisation turns workflow provenance from an opaque graph into a roadmap with

which we can (1) determine sources of implicit context and (2) the scope of reach of that context. We implement our techniques with technology-independent provenance models and we showcase their benefit using real-world workflows. We make the following contributions.

- *LabelFlow*, a generic framework, that can be plugged with discipline-specific metadata extractors, for annotating datasets. Central to this framework is a set of labelling operators for minting and propagating annotations.
- Labelling Pipelines, which are shadow annotator processes deduced per scientific workflow. The labelling pipeline of a workflow can be used for annotating results for all executions of that workflow. We provide a practical algorithm that consumes a workflow description and its activities' functional categories and produces a labelling pipeline.
- An implementation of *LabelFlow* based on technology-independent standard provenance models and its validation using a case study.

Early requirements for *LabelFlow* and the basics of our case study were published earlier in two workshop papers [26,27]. This current submission gives a complete formal specification and implementation for *LabelFlow*.

The paper is organised as follows. In Section 2 we introduce an example case-study workflow and the landscape of metadata surrounding workflows. Here we also discuss the requirements of reporting and the shortcomings of generic provenance. Afterwards, in Section 2 we provide an architectural overview of our approach to provenance annotation. In the sections that follow we elaborate on the components of this architecture. In Section 4 we describe *LabelFlow* in detail, outlining *Label Model*, *Labelling Operators* and the generation of *Labelling Pipelines*. In Section 6 we revisit the case workflow to assess the utility of having explicit domain-specific metadata for reporting. In Sections 7 and 8 we outline related work and discuss *LabelFlow* critically in comparison to related work. We conclude in Section 9.

## 2. Workflows, Provenance and Reporting: A Case

In this section, we provide an example Astronomy workflow [28] from the Taverna system [29] and its provenance. We also identify the requirements of reporting as queries over workflow provenance, which we illustrate with common queries adopted from the Provenance Challenge [30]. We believe the example workflow is representative of scientific workflows because:

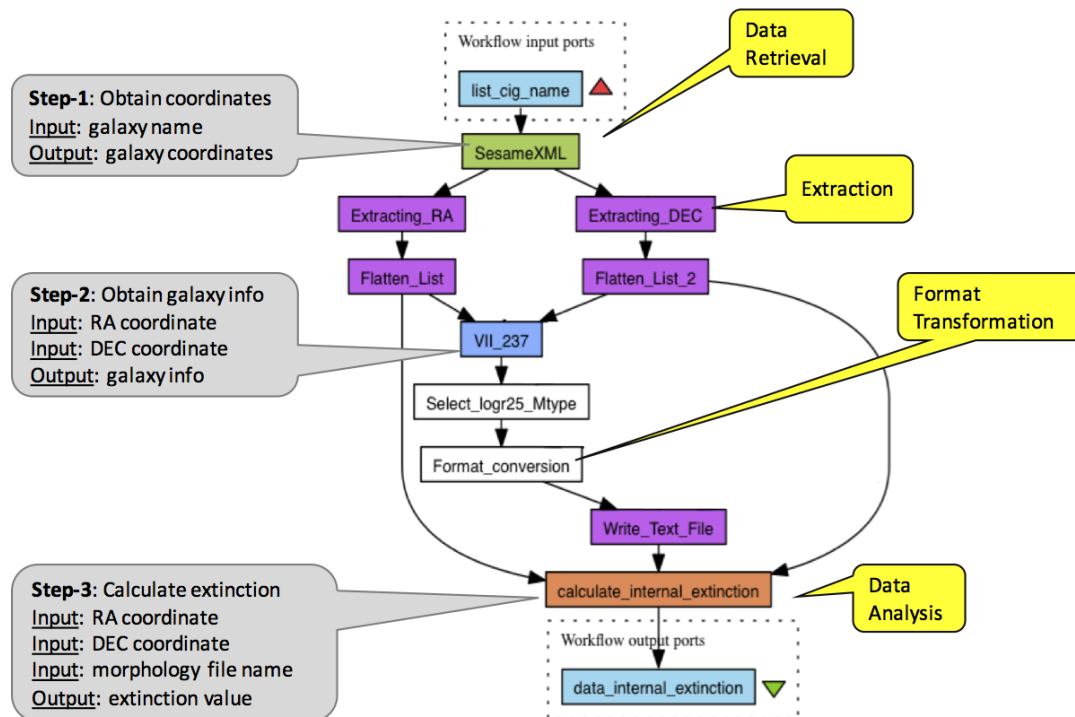
- It contains the commonly observed type of activities in workflows and reflects their occurrence percentage. As we discuss in Section 3, a comprehensive survey has categorised activity functionalities to certain Motifs, our workflow's activities have the data analysis and data retrieval motifs as well as extensive data adaptation.
- Analyses and data retrievals in this workflow are configured by input parameters, which, as we discuss in Section 2.3, become an important hook for querying provenance.
- It illustrates the genericity of workflow provenance, our prime motivation, which limits the use of provenance for reporting. It also illustrates the  $n - by - m$  pattern in provenance, which can be observed in workflows and negatively impacts the utility of both generic and annotated provenance.

While our case involves a Taverna example, our approach is primarily system-independent and builds on standard models of workflows and provenance.

### 2.1. Layers of Provenance

Taverna workflows are comprised of *tasks*, *input/output ports* of tasks and the *dataflow* links among ports. Figure 1 illustrates our workflow that takes as input a set of galaxy names (*list\_cig\_name*), and outputs extinction values per galaxy (*data\_internal\_extinction*). The workflow retrieves data, from remote astronomical databases (Steps 1 & 2) and uses this information to calculate extinction

values (Step 3). The unnumbered tasks in between are adapters, which perform (*Data*) *Extraction*, *Format Transformation* and similar. Taverna supports simple structural typing of data. Ports of tasks can be defined to hold data *Items* or nested *Collections* of data items.



**Figure 1.** Sample workflow from Astronomy developed by the Wf4Ever project that takes as input a set of galaxy names, and outputs extinction values per galaxy.

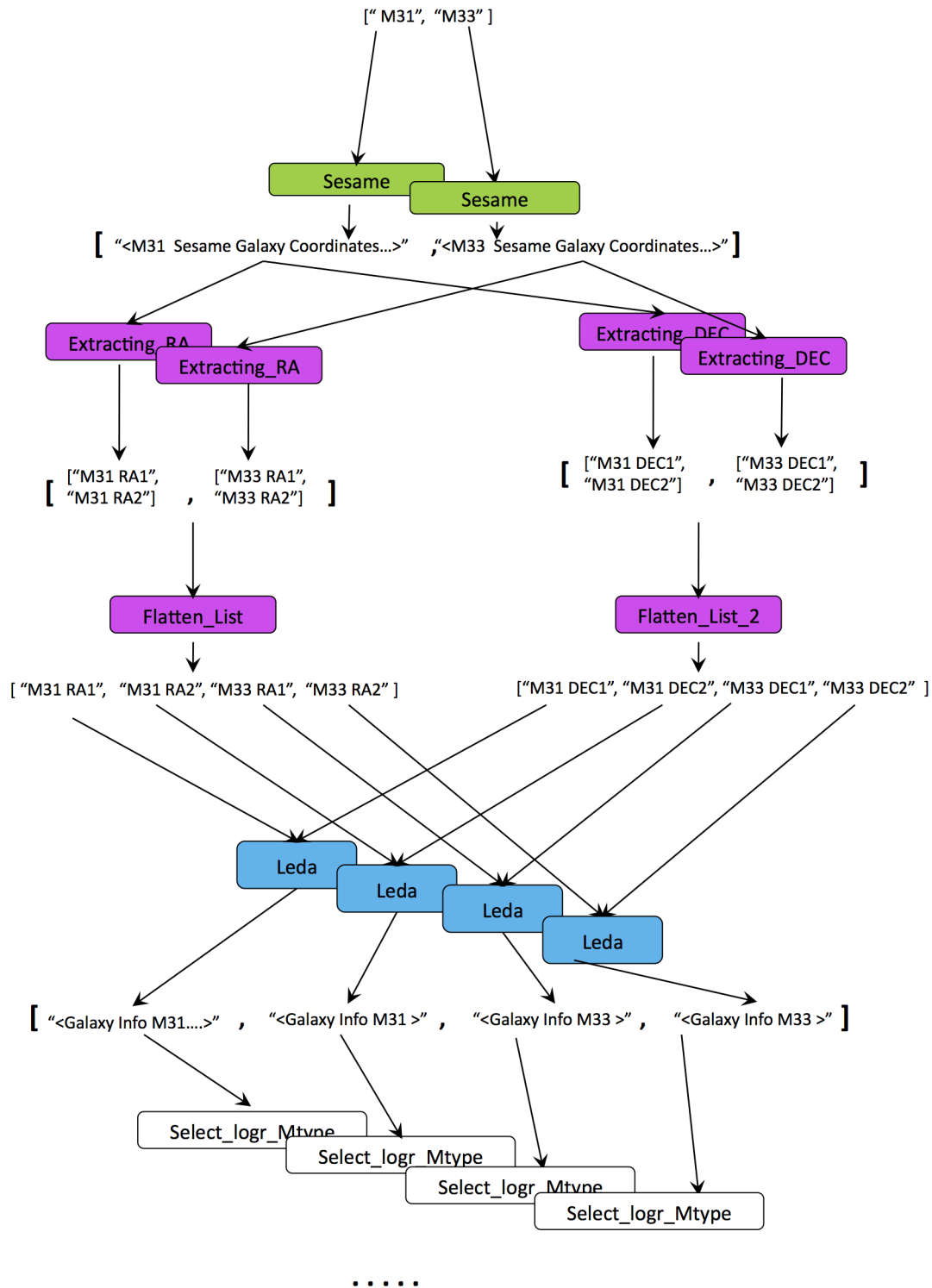
Figure 2 provides an illustration of the execution of this workflow with two input galaxies. The information given in Figure 2 also makes up the core of workflow provenance.

An important capability found in most workflow systems is iterated execution. Taverna achieves iteration by allowing tasks with ports having mismatching structural data types to be composed. e.g., a task that emits a collection of galaxy ids may be composed with a follow-on task that consumes a single id. In this case the follow-on task will be *iterated*. Similarly a task producing a single item can be composed with follow-on that accepts a collection, in this case the output would be wrapped into a collection of depth acceptable by the follow-on task.

Iteration is a crucial feature for scientists to run analyses on different subjects (galaxies in our example), or, to perform factorised analyses by changing input configurations (not shown in our example). Here we skip the details of Taverna iteration [31], as it is beyond the scope of our work. However, iteration does have an impact on annotation and reporting, which we summarise as follows:

- Iteration proliferates data generation. In our example in Figure 2 the number of outputs increases linearly with inputs ( $n$  inputs produce  $2n$  outputs). In cases of complex factorised analyses, where tasks are run on (cartesian) combinations of inputs, the increase in outputs becomes polynomial. As such, automation in annotation is a crucial requirement.
- Iteration is an intricate feature of workflows. It requires understanding nested collection structures as well as creating cartesian combinations of items from collections. When utilised correctly, iteration is the primary mechanism making provenance an index linking the subjects and factors of an analysis to corresponding results. However, in certain cases, such as to avoid repeated and costly invocation of remote services or for quicker data adaptation, iteration can be by-passed. This is illustrated with the “Flatten\_List” step in Figure 2. All the steps in the

workflow are repeated for each input, whereas “Flatten\_List” is executed once, processing data of multiple galaxies at a single step. This (anti)pattern, also known as the “ $n - by - m$  problem”, breaks input-to-output traceability; and, as we shall identify later in this section, it is one of the main factors that reduce the utility of provenance for reporting.

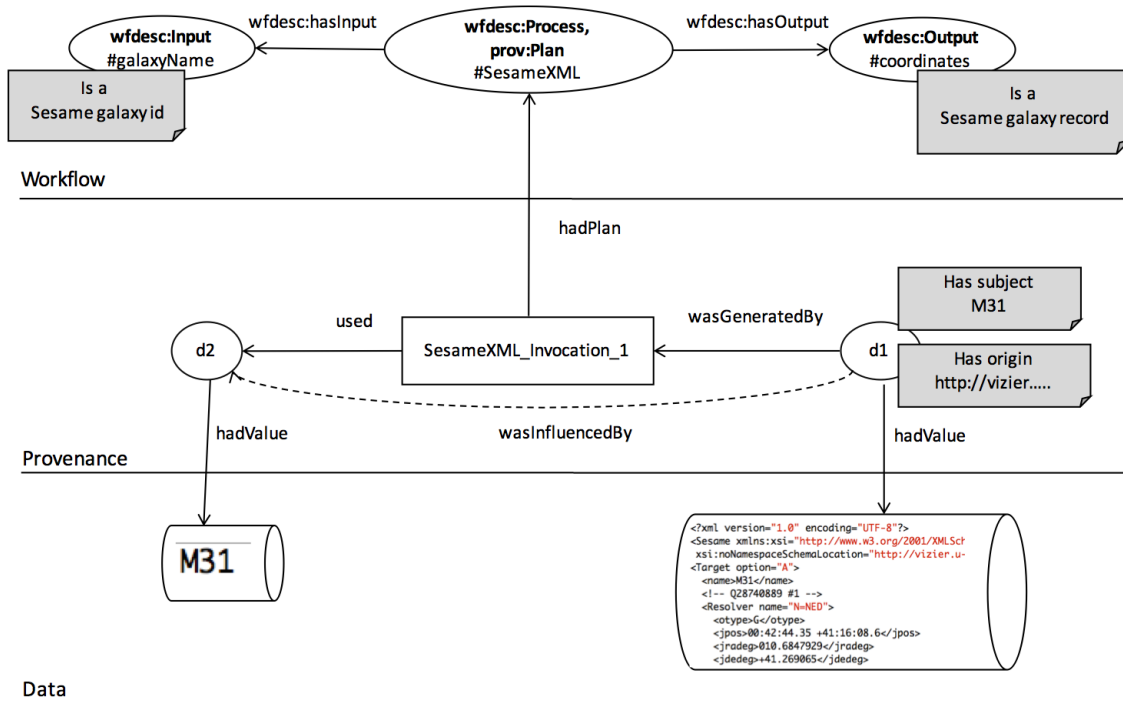


**Figure 2.** A fragment of execution illustrated for the Astronomy workflow.



In order to represent the trace of Figure 2, workflow systems utilise generic vocabularies. In Figure 3 the top “Workflow” layer contains the abstract specification of a workflow task in the Wfdesc model [32]. Wfdesc is a “node and directed-arc” RDF graph [33], where nodes are typed as *Processes* (tasks), and their *Input/Outputs* (ports). The middle “Provenance” layer provides a PROV-compliant [34] graph, containing an *Activity* node (rectangle) representing one execution of the workflow task, and two *Entity* nodes (ovals) representing the data consumed and produced by the *Activity*. At the data layer in Figure 3 we have data values stored within files (denoted with disk shapes). Most workflow systems including Taverna, adopt a separated storage scheme for data and provenance, where data is often stored in the file system and the provenance metadata is stored in graph/triple stores or relational databases.

Domain-specific annotations (denoted with grey-shaded boxes in Figure 3) are attached onto generic metadata. We identify two categories of domain-specific annotations. *Static* annotations are those at the “workflow” layer, which represent data/process characteristics valid for all executions of the workflow. e.g., Specifying that the semantic type of an input is a galaxy id. Most workflow systems support static annotations. *Dynamic* annotations are those at the “provenance” layer; they represent characteristics that may change from execution to execution. e.g., Specifying that the input of a particular task is the id for the Andromeda Galaxy (M31).



**Figure 3.** Illustration of Static and Dynamic Metadata at different layers of workflow provenance.

## 2.2. Modelling Workflows and Provenance

We now introduce a basic formalism for representing workflows and provenance with the following assumptions:

- We adopt a port-based modelling of communication among tasks. This is the approach taken in Taverna [29], Wings [8], Vistrails [9] systems, and it is also adopted by abstract models like Wfdesc [32] and D-PROV [35].
- We assume that a workflow is a directed acyclic graph of analytical tasks and dataflow dependencies among ports of tasks. We assume that provenance is a directed acyclic graph of (data) entities and influence relations among entities (produced/consumed by activities).

- We exclude information on the semantics of task iteration. Workflow systems all provide their own means to repeatedly apply tasks to data. The lack of iteration configuration specifics does not affect *LabelFlow*'s ability to operate. *LabelFlow* operates at the level of individual task invocations, and a task's execution footprint in provenance is the same in all mentioned workflow systems [31]. Meanwhile workflow systems differ in the way they reflect data granularity in provenance [31]. In Taverna an iterated task would be consuming individual items in a collection, whereas in Vistrails there is not collection modelling therefore all iterations would appear to consume/produce a single entity.

Throughout our formalisation we use  $\mathbb{S}$  to the set of Strings.

**Definition 1.** Workflow  $w$

A workflow is denoted with the triple  $\langle \text{PRO}, \text{POR}, \text{LINK} \rangle$  as well as the functions  $\text{inPort}$ ,  $\text{outPort}$ ,  $\text{src}$  and  $\text{snk}$ ;

$\text{PRO}$  is the set of processor names.

$\text{POR}$  is the set of port names.

$\text{LINK}$  is the set of dataflow links among ports.

$\text{inPort} \subseteq \text{PRO} \times \text{POR}$  and  $\text{outPort} \subseteq \text{PRO} \times \text{POR}$  are two interface relations that mapping processors to their inputs and output ports.

$\text{src} \subseteq \text{LINK} \times \text{POR}$  and  $\text{snk} \subseteq \text{LINK} \times \text{POR}$  are two functions that map links to their source and sink ports.

**Definition 2.** Provenance trace  $P_w$

The provenance trace obtained from a particular run of  $w$  is denoted with the tuple  $\langle \text{ACT}, \text{ENT} \rangle$  as well as the relations  $\text{wasGenBy}$ ,  $\text{used}$ ,  $\text{hadMember}$ ,  $\text{hadPlan}$ ,  $\text{invocations}$ ,  $\text{input}$ ,  $\text{output}$ ;

$\text{ACT}$  is the set of activities.

$\text{ENT}$  is the set of (data) entities.

$\text{hadItem} : \text{ENT} \times \text{ENT} \times \mathbb{N}^+$  is a relation, which designates that a collection-type entity has an item at the designated nesting level.

$\text{input} \subseteq \text{ACT} \times \text{POR} \rightarrow \text{ENT}$  is a function, which maps an activity and a port pair to the input entity that has been used by the designated activity at the designated port.

$\text{output} \subseteq \text{ACT} \times \text{POR} \rightarrow \text{ENT}$  is a function, which maps an activity and a port pair to the output entity that has been generated by that activity at that port.

$\text{invocations} \subseteq \text{PRO} \times \text{ACT}$  is a relation, which maps a processor in the workflow to the activities which are invocations of that processor during the workflow execution. Due to iteration a processor can map to multiple activities in the provenance trace.  $\text{invocations}$  is the inverse relation of the  $\text{hadPlan}$  function.

$\text{influencedBy} \subseteq \text{ENT} \times \text{ENT}$  is a relation that maps an entity to another, where the generation of the former is influenced by the latter.

$\text{influencedBy}^* \subseteq \text{ENT} \times \text{ENT}$  is a relation that holds the transitive closure of influence relations. The intentional rules for computing the influence relations is as follows:

```

influencedBy(F, E) :- input(A, P, E), output(A, Q, F)
influencedBy(D, C) :- influencedBy(D, I), hadItem(C, I)
influencedBy(D, I) :- influencedBy(D, C), hadItem(C, I)
influencedBy*(F, E) :- influencedBy(F, E)
influencedBy*(F, G) :- influencedBy*(F, E), influencedBy*(E, G)

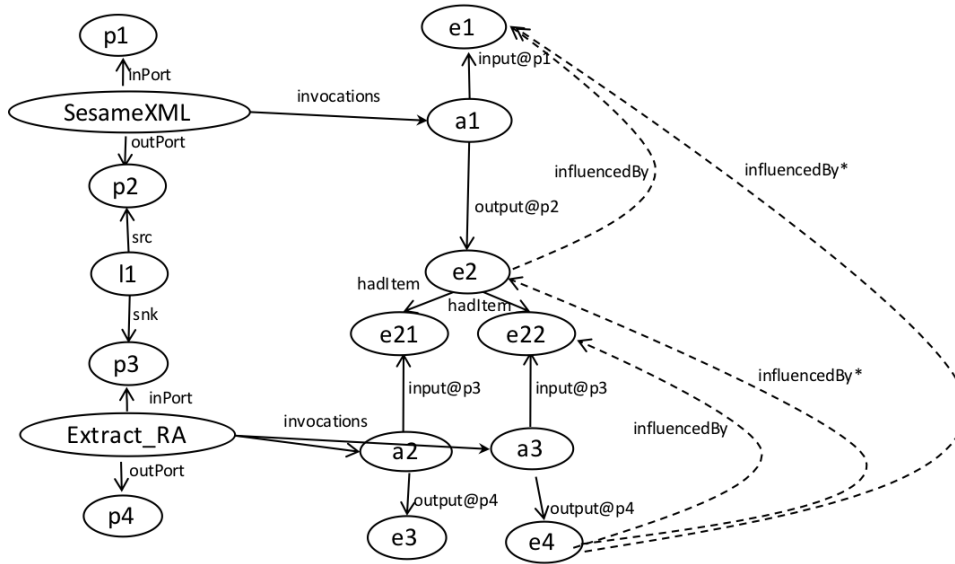
```

**Example 1.** (Workflow, Provenance Trace) In the left hand side of Table 1 we illustrate our formalisation with predicates outlining a simple workflow containing two processors (*SesameXML* and *Extract\_RA*). The diagrammatic view of predicates is given in Figure 4 where we denote set types with nodes, and functions and relations with arcs.



**Table 1.** Formal specification of an example workflow  $w1$  and its provenance  $P_{w1}$ .

|                 |   |
|-----------------|---|
| $w1 =$          | $\langle PRO, POR, LINK \rangle$  |
| $PRO =$         | $\{SesameXML, Extract\_RA\}$  |
| $POR =$         | $\{p1, p2, p3, p4\}$  |
| $LINK =$        | $\{l1\}$  |
| $inPort =$      | $\{\langle SesameXML, p1 \rangle, \langle Extract\_RA, p3 \rangle\}$  |
| $outPort =$     | $\{\langle SesameXML, p2 \rangle, \langle Extract\_RA, p4 \rangle\}$  |
| $src =$         | $\{\langle l1, p2 \rangle, \langle l1, p3 \rangle\}$  |
| $snk =$         | $\{\langle l1, p3 \rangle\}$  |
| $P_{w1} =$      | $\langle ACT, ENT \rangle$  |
| $ACT =$         | $\{a1, a2, a3\}$  |
| $ENT =$         | $\{e1, e2, e21, e22, e3, e4\}$  |
| $hadItem =$     | $\{\langle e2, e21, 1, \rangle, \langle e2, e22, 1 \rangle\}$   |
| $invocations =$ | $\{\langle \langle SesameXML, a1 \rangle, \langle Extract\_RA, a2 \rangle, \langle Extract\_RA, a3 \rangle \rangle\}$ |
| $input =$       | $\{\langle a1, p1, e1 \rangle, \langle a2, p3, e21 \rangle, \dots\}$  |
| $output =$      | $\{\langle a1, p2, e2 \rangle, \langle a2, p4, e3 \rangle, \dots\}$   |

**Figure 4.** Diagrammatic view  $w1$  and  $P_{w1}$ .

**Definition 3.** Data trace  $D_w$  A data trace is comprised of the values of all entities used or generated during a particular run of a workflow. (In our formalism data and provenance traces capture information of a particular run, we therefore omit any run identifier). We define the data trace for a workflow  $D_w$  as the combination of data traces of its processors  $D_p$ . Specifically;  $value \subseteq ENT \times \mathbb{S}$  is a function, which maps an entity in the provenance trace to the String representation of the value stored for that entity in the data layer (typically the file system), then the data trace for an activity, processor and workflow is defined as follows:

$$[H]D_a = \{v \in \mathbb{S} \mid \exists e \exists p (value(e, v) \wedge (input(a, p, e) \vee output(a, p, e)))\}, \quad (1)$$

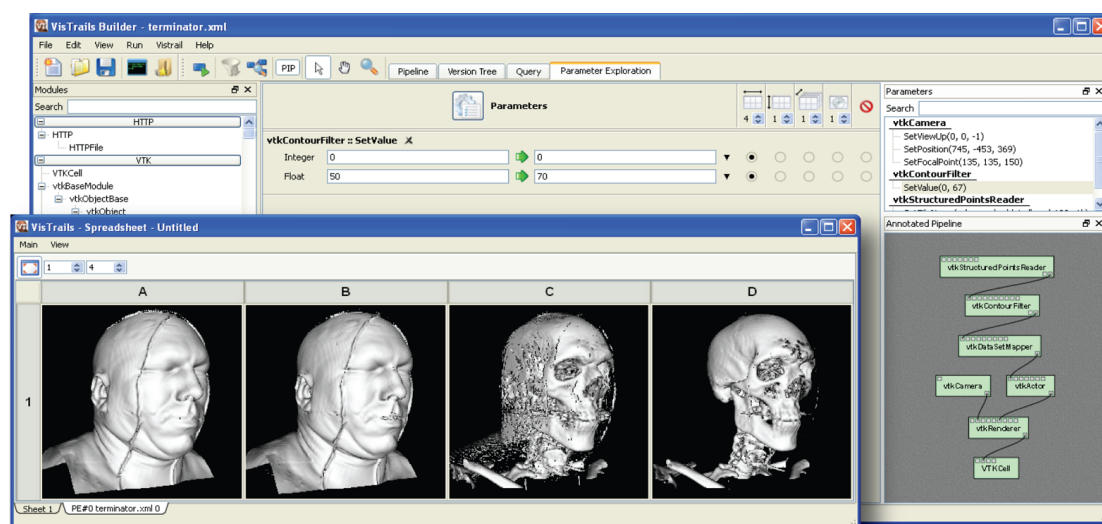
$$D_p = \bigcup_{\{a \in invocations(p, a)\}} D_a, \quad (2)$$

$$D_w = \bigcup_{p \in PRO} D_p \quad (3)$$

### 2.3. Using Provenance for Reporting

One can query workflow provenance to select and organise workflow results based on the context i.e., input configurations they originate from. In their reporting interfaces workflow systems often hide the provenance graph or the query result from the user but instead provide other views commonly spreadsheet like tables listing all data artefacts over a single thread of lineage. Vistrails's Spreadsheet View [9] (see Figure 5) or Graph2Tab [36] are examples. We believe the presentation of (provenance) graphs and presentation of results is separate and very important research thread, which we have not yet received sufficient attention [37]. In this paper, we focus on provenance queries that underpin reports.

In Table 2 we provide three queries, which are adapted from queries of the Provenance Challenge [25,30]. These are common provenance queries, which are comprised of first locating a node of interest and then traversing the lineage relations in a provenance graph. In fact 5 out of 9 of the Provenance Challenge queries [30] are based on restrictions on either data values or annotations, which are assumed to exist. Note that in this paper we do not intend to advance the state of the art on what possible queries can be over provenance. Our focus is be able to fully implement common workflow provenance queries.



**Figure 5.** A provenance report from Vistrails that show data generated at each step of the workflow for a particular parameter setting.

Q1 has two parts; first part seeks data by its origin, second part seeks detailed information on origin. The analysis in our example workflow is intended to run on data of multiple subjects. Q2 seeks to find all (intermediary and final) outputs that belong to one subject, i.e., M31, Andromeda Galaxy. Q3 seeks results of a particular activity, extinction calculation. This activity takes 3 input parameters: the two parts of a coordinate and a file path for morphology data.

**Table 2.** Three Provenance Queries for Reporting Data.

|  |
|--|
| Q.1 Which results are coordinates obtained from the Sesame database, from which database catalogs are they obtained. |
| Q.2 Select all results belonging to the Andromeda Galaxy.  |
| Q.3 Select extinction calculation results for the Andromeda Galaxy, where the morphology parameter setting was 0.45. |

As part of our case-based validation we implement queries as node selections and traversals over generic versus annotated provenance graphs. We use the “-G” and “-A” suffixes with query names to denote the nature of provenance they are run on. We measure query result precision as follows. To understand the utility of provenance for reporting we differentiate between *True Accuracy* vs. *Lineage Accuracy* of results to a query. All results obtained to all our queries (expressed as intentional rules over our formal model) has 100% Lineage Accuracy. Meanwhile these results may not have *True Accuracy*. A result has *True Accuracy* if it falls in the scope implied by the query (e.g., for Q1 the results that actually contain coordinate data that is retrieved from the Sesame database, not the results computed using those coordinates or their descendants. Or for Q2 the results that actually contain data belonging to galaxy M31 but not galaxy M32 and so on). In Figure 6 We present the *True Precision* of results, which we define as:

$$\frac{\text{\# of Truly Accurate results}}{\text{Total \# of results}}$$

**Q1-G:** We can realise Q1 partially over generic provenance. For the first part, the intent is to obtain data retrieved from the Sesame database or its local copies generated through adapter steps. Here we use lineage as a pseudo (replacement) mechanism to denote data origin. We seek results, whose derivation path includes and invocation of the *SesameXML* processor, which we know accesses the Sesame database. We’re unable to implement the second part of the query, which inquires catalog information. This is to be found embedded in the VO Table XML String, outputted from the *SesameXML* processor, as a combination of a tag name and a fragment of string value within that tag. Given that there are diverse data formats even in the context of astronomy there is no systematic way to place value restrictions with the provenance and query models outlined above. Let *s* denote the *SesameXML* processor of our workflow; then Q2 formally is:

(PARTIAL)

```
answer(D) :- PRO(s), PRO(s,P), invocations(s, A), output(A,P,O), influencedBy*(D,O)
```

Roughly one third of the results whose derivation path includes a Sesame DB lookup actually contain data that is retrieved from Sesame (See Q1-G precision in Figure 6). Remaining two thirds of results are those that are computed through analyses by using the data obtained from Sesame. As workflow activities are observed as black-boxes in provenance, we have opaque lineage that tells us there is some influence relation among data artefacts but falls short of differentiating between:

- lineage based on value-copying/data adaptation.
- lineage based on any other analytical computation

Designating origin via path-based linkage to an element identified in workflow design is weakly precise; yet it is robust to increases in the workflow inputs. Increasing the number of galaxies in a workflow run does not alter the fact that only one thirds of outputs are copies of data from the Sesame Database. The source catalog information sought in the second part of query is available within some of the data values, i.e., in the XML output of the *SesameXML* activities, there is a field that specifies the catalog (the relevant subpart in Sesame DB) that the result comes from. However when these results are stripped of their XML padding the catalog information is no longer associated with retrieved coordinates (outputs of the *Extract\_RA* and *Extract\_DEC* steps). Therefore this part of the query cannot be implemented.

**Q2-G:** We realise Q2 by seeking the input data node with value *M31* and later traversing provenance to find all data products that have this node in their lineage. Formally;

```
answer(D) :- ENT(e), value(e,"M31"), influencedBy*(D,e)
```

Figure 6 shows the precision for this query. Recall that iteration was by-passed at the *Flatten\_List* step, which consumes data belonging to all input subjects, thereby obfuscating the traceability between results and the subject. As a result Q2-G, which exploits lineage in provenance, rapidly loses its precision as an index as the workflow is run with increasing number of subjects.

**Q3-G:** Q3 is also encoded partially. The predicate stating that morphology parameter should be 0.45 requires access to morphology parameter's values. On the other hand, the extinction calculation activity accepts as input a configuration parameter file name, rather than the actual parameter value. This value is available in outputs of upstream adapter steps, or in the output of extinction calculation. However within these data values, the morphology information is present in tab delimited texts where multiple numeric values are present. Similar to Q1 this prohibits to build a mechanism to systematically predicate over a fragment of those texts. Therefore we omit the morphology value restriction, only seeking extinction values computed for a particular galaxy's coordinates. Let  $x$  denote the *calculate\_internal\_extinction* processor of our workflow, then Q3 formally is:

(PARTIAL)

```
answer(0) :- PRO(x), invocations(x,A), ENT(e), value(e, "M31"), influencedBy*(I,e),
            input(A,_,I), output(A,_,0)
```

As extinction calculation activity accepts inputs, which are no longer accurately traceable to input galaxy names (post *Flatten\_List*), the resulting query precision is as equally bad as Q2-G (see Figure 6).

Our case highlights the following issues:

- Correct implementation of iteration is a pre-requisite for provenance being useful in reporting. Queries that seek results belonging to a particular input (subject or factor) require discrete reachability between inputs and respective outputs. When traceability is broken, provenance, either generic or annotated, is of little use. The lack of discrete traceability that causes the sharp loss of precision in Q2-G and Q3-G is not a problem that we're trying to solve with *LabelFlow*. In prior work we've tackled this problem and shown that workflows can be analyzed to check whether their provenance will have the n-by-m pattern, i.e., lack of discrete traceability [31]. We find it important to highlight this pattern in the context of this paper, because, as we shall see in Section 6 if it exists in provenance it equally reduces provenance utility even after labelling.
- Domain specific information is key for reporting. Q1 seeks *static*, whereas Q2 and Q3 seek *dynamic* attributes, yet we realised our queries over generic provenance. In the absence of metadata, in order to find nodes of interest, we were forced to put selective criteria on data values (Q2-G and Q3-G), or in attributes like name (Q1-G). This approach proved to have the following disadvantages:
  - Due to the separated storage of data and provenance seamless implementation of provenance queries was not possible. In fact for queries Q2-G and Q3-G, as a precursor, we identified which node in the provenance graph corresponds to the M31 galaxy by first scanning through the data values stored in the file system.
  - As we realise queries over implicit information, and as there is no structure or vocabulary restrictions on this information, our approach is adhoc. e.g., the informativeness of a name for a workflow port or activity is at the disposal of workflow designer, names can be freely given and the same activity (e.g., Sesame database lookup) can have different names across workflows. Similarly in Q3-G we were unable to implement morphology criteria part of query as the data values were not self-describing and structured enough to allow a systematic implementation of this criteria.
- Transparent lineage is needed for reporting. One of our queries (Q1-G) was seeking data based on its origin. In our implementation we represented this with a query where we sought nodes that have some lineage relation to designated origin node. Our answers to this query were partly correct. This is because of we were using opaque lineage relations, a typical result of black-box workflow provenance. Opaque lineage tells us that one data artefact influences the other, but it does not specify the specific nature of this influence. On the other hand Q1-G requires more transparency, it seeks those data artefacts that descend from an origin artefact via a particular influence relation, i.e., value-copying.

A natural question that may arise is “Are the queries representative of the user’s requirements for reporting?”. We believe that queries in a workflow provenance setting are not entirely arbitrary. The user is equipped with a fixed set of hooks/abstractions with which she can inquire the provenance.

- she can ask for outputs of analytical tasks based on input configurations, (the quintessential workflow provenance query)
- she can inquire data origin for data retrieved from external databases (a typical motif in scientific workflows)

In addition, in this section we illustrated that fully and systematically implementing these queries over generic provenance was not possible. In the following section we introduce the *LabelFlow* approach for addressing the requirements identified above.

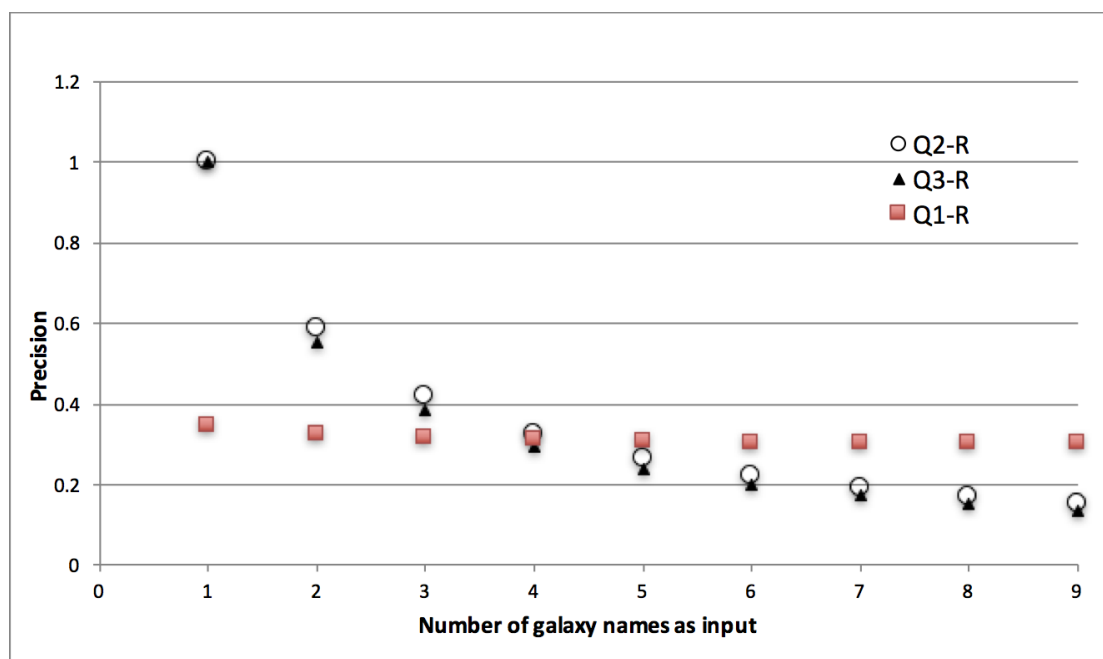


Figure 6. Precision of results when queries are run over generic provenance.

### 3. Architecture and Assumptions

Figure 7 illustrates the *LabelFlow* architecture. We undertake labelling as an offline process, which comes after the design (Step a1) and execution (Step a2) of scientific workflows. Workflow executions generate data and generic provenance, which make up our primary source of information for obtaining domain-specific metadata. We represent metadata with attribute-value pairs called *Labels*. We perform labelling through processes called *Labelling Pipelines*. Pipelines are created per scientific workflow (Step b1), using the workflow’s description and workflow annotations called *Motifs*. Motifs are the result of a previous study [24] in which we analysed 240 workflows from 4 systems to identify the nature of data processing in workflows. *Motifs* is a taxonomy of task functions in workflows [38]. This study has shown that a minority (30%) of tasks perform the scientific heavy lifting in a workflow through analysis, visualisations or data collection; whereas the remainder majority (70%) is dedicated to data adaptation. We use motif annotations when generating a *Labelling Pipeline* by composing *Labelling Operators*. More specifically a task’s *Motif* informs which labelling operator should stand in for that task in the associated *Labelling Pipeline*, and how that operator should be configured. *Labelling Operators* delegate the duty of extracting domain-specific *Labels* from data values to *Labelling Functions* (Step b2). Operators take care of the layering and propagation of *Labels* over PROV compliant provenance graphs (Step b3).

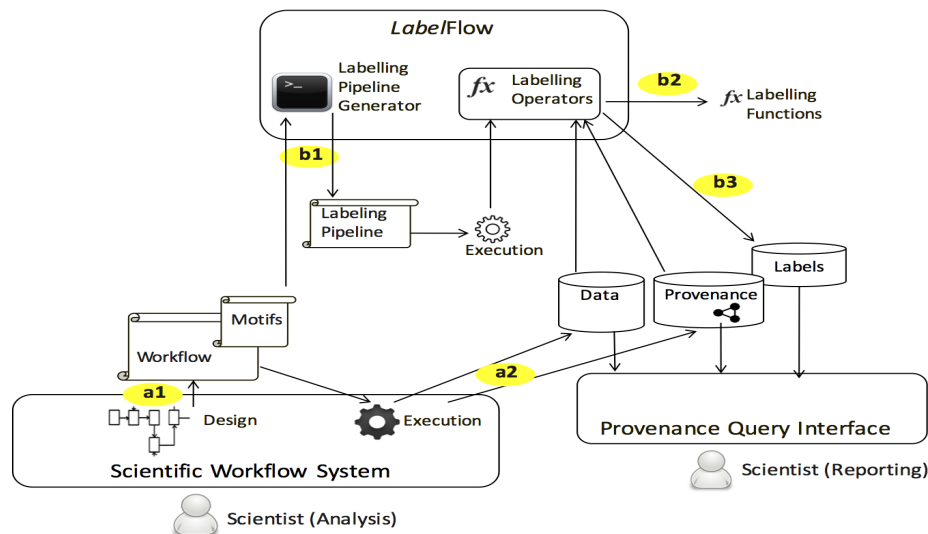


Figure 7. LabelFlow Architecture.

In implementing *LabelFlow* we assumed that:

- *Labelling Functions* are available for domains and support an invocation interface that allows them to be plugged into *LabelFlow*.
- Workflow tasks are annotated with an extended form of *Motif* annotation called a *Labelling Specification*. Note that *Motifs* only describe a task's function. Meanwhile, tasks have input/output ports and we require information on which ports shall receive labels and if/how ports are related. We discuss the information within *Labelling Specifications* in the next section. The mechanism to create *Labelling Specifications* is left out of scope in this paper.

#### 4. LabelFlow Framework

In *LabelFlow* we associate core labelling behaviour with workflow tasks. We achieve labelling through *MINT* and *PROPAGATE* operators. Table 3 lists all *Motifs* in our case workflow, some were illustrated (with right-hand side callouts) earlier in Figure 1. The scientifically significant activities are hotspots of data that can be used for minting domain specific labels which make explicit the contextual information found in data values. In our case these are the tasks for obtaining of Galaxy information from repositories (those with the *Data Retrieval Motif*) and the local extinction calculation (*Data Analysis Motif*). Our case also illustrates that data's inception and its subsequent use can be separated by several adapter tasks. For most adapters, task functionality implies certain transparency over the lineage between task inputs and outputs; the task's outputs are built by copying values of inputs. We exploit this transparency to extend the reach of annotations over a data artefact to its copies, through propagation of metadata.



**Table 3.** Workflow Motifs, whether they imply value-copying, and the associated labelling behaviour.

| Motif  | Value-Copying           | Example in Case  | Labelling Behaviour |
|--|-------------------------|--|---------------------|
| Data Analysis<br>Data Retrieval<br>Data<br>Visualization | N/A                     | “SesameXML”,<br>“VII_237”,<br>“calculate_int_extinction” | mint                |
| Augmentation   | $I \xrightarrow{m-1} O$ | Not present in case.                                     | propagate           |
| Extraction   | $I \xrightarrow{1-m} O$ | “Extract_DEC”,<br>“Extract_RA”                           | propagate           |
| Split  | $I \xrightarrow{1-1} O$ | Not present in case.                                     | propagate           |
| Merge  | $I \xrightarrow{1-1} O$ | “Flatten_List”   | propagate           |
| Filter   | $I \xrightarrow{1-1} O$ | “Select_logr25_Mtype”                                    | propagate           |
| Combine  | $I \xrightarrow{m-1} O$ | Not present in case.                                     | propagate           |

*MINT* obtains labels by invoking the external labelling function associated with a workflow task. The function expects as input all data artefacts that were used and generated by a particular invocation of the task. Mint operator is responsible for scouring the PROV trace to obtain the inputs and outputs of all invocations of that task and forward these to the labelling function. The labelling function will then extract metadata from data values and return them to the mint operator as labels. Finally the *MINT* operator attaches those labels on to the data artefacts nodes in the provenance graph. *PROPAGATE* transfers labels from designated inputs of a task to designated outputs by creating clones of labels. Note that labels may need to be propagated from multiple input (source) ports. In this case the propagate operator will create a union set of labels. *MINT* and *PROPAGATE* are generic, they can be used to decorate the traces of workflow tasks (from different workflow systems) as long as their execution is recorded in PROV.

In addition, we provide two further operators, namely *DISTRIBUTE* and *GENERALISE*. These operators cater for the collection-oriented nature of data and propagate labels upwards/downwards along the collection-item structure of data nodes in a PROV trace. While *DISTRIBUTE* and *GENERALISE* are also generic in behaviour, they have been developed primarily in response to Taverna workflow system’s iteration behaviour. Recall that *MINT* and *PROPAGATE* were associated with tasks. *DISTRIBUTE* and *GENERALISE* are associated with dataflow links that connect two ports, which by definition produce and consume data of mismatching structured types. In cases where the task at one end of a dataflow link produces a collection, and the other end consumes an item, *DISTRIBUTE* is responsible for propagating labels from the top-level collection to each item at a specified depth. In cases where the activity at one end produces individual items in a collection, and the other end consumes the collection *GENERALISE* is responsible for propagating labels from items to the enclosing collection at a specified depth.

In the following section formally introduce *LabelFlow*.

## 5. LabelFlow Model

**Definition 4.** *Label Definition* A label definition is the tuple  $\langle n, t \rangle \in \mathbb{S} \times T$ .  $n$  is the label name,  $t$  is the type designator with  $T = \{QName, String, Integer, Datetime\}$ . Currently *LabelFlow* implementation only supports *String* typed labels. We represent label definitions with the relation  $L_{def} \subseteq \mathbb{S} \times T$ .

**Definition 5.** *Label Instance* (short: *Label*) A label is denoted with the triple  $\langle d, v, t \rangle \in \mathbb{S} \times \mathbb{S} \times ENT$ . Here  $d$  is the name of the label definition,  $v$  is the label value and  $t$  is the target data entity in the provenance trace to which the label is attached. We refer to the domain of label instances with  $\mathbb{L} = \mathbb{S} \times \mathbb{S} \times ENT$ . We denote the label space populated by *LabelFlow* when for a particular run of workflow with the relation  $\mathbb{L}_{ins} \subset \mathbb{L}$ .

**Definition 6.** *Label Vector* A label vector is denoted with the tuple  $\langle n, D \rangle \in \mathbb{S} \times 2^{\mathbb{S}}$  is a named set of label definitions. Label and label vectors would be specific to each scientific domain.

**Example 2.** (Label Definition, Label Vector, Label Instance) In Table 4 we illustrate the label space for the simple workflow  $w1$  of Figure 4. We illustrate four label definitions and a vector comprised of those label definitions. Note that, we do not further utilise label vector or label definitions in our formalisation, however in the LabelFlow implementation, they serve a practical purpose, which we discuss in Section 5.1. The label space denoted with  $L_{ins}$  would have no labels but these would get created during the labelling process.

**Table 4.** Label definitions and a snapshot of the label space for  $w1$ .

|             |   |
|-------------|---|
| $L_{vec} =$ | $\langle \text{"astro"}, \{\text{"referenceURI"}, \text{"referenceCatalog"}, \dots\} \rangle$   |
| $L_{def} =$ | $\{ \langle \text{"referenceURI"}, \text{String} \rangle, \langle \text{"referenceCatalog"}, \text{String} \rangle, \langle \text{"hasMorphology"}, \text{String} \rangle, \langle \text{"hasSubject"}, \text{String} \rangle, \}$  |
| $L_{ins} =$ | $\{ \langle \text{"hasSubject"}, \text{"M31"}, e2 \rangle, \langle \text{"referenceCatalog"}, \text{"Sc = Simbad"}, e2 \rangle, \langle \text{"referenceCatalog"}, \text{"N = NED"}, e2 \rangle, \langle \text{"referenceURI"}, \text{"http : // cdsws.u - strasbg.fr /"}, e2 \rangle \}$ |

**Definition 7.** *Domain-Specific Labelling Functions* LabelFlow assumes that external (domain-specific) functions that create labels support a common interface. We denote the set of all such functions with  $\mathbb{F} = \{f \mid \text{dom}(f) = 2^{\mathbb{S}} \wedge \text{rang}(f) = 2^{\mathbb{L}}\}$

**Definition 8.** *Mint Function*  $\text{Mint} : \text{PRO} \times 2^{\text{POR}} \times \mathbb{S} \rightarrow \mathbb{L}$  is a function that generates labels by utilising the provenance trace, the data trace and a domain-specific labelling function. Its specification is as follows:

$$\text{Mint}(p, T, f) = \bigcup_{\langle a, t \rangle \in \{a \mid \text{invocations}(p, a)\} \times T} (\text{pipe}(D_a, f) \times \{e \in \text{ENT} \mid \text{output}(a, t, e)\})$$

*Mint* accepts as input a processor  $p$ , a set of target ports  $T \subset \text{POR}$ , which are outputs of  $p$ , plus a domain-specific labelling function  $f \in \mathbb{F}$ . Within *Mint* for each invocation  $\{a \mid \text{invocations}(p, a)\}$  and target port  $t \in T$  combination we obtain the data trace of that invocation  $D_a$ . We apply the given domain-specific labelling function to the data trace,  $\text{pipe}(D_a, f)$  (Here we borrow the *pipe* higher-order function from functional programming). The domain-specific labelling function returns a set of label definition and value pairs (For convenience let's call these proto-labels). We then bind these proto-labels to the output  $e$  of the activity  $a$  with a cartesian product.

**Definition 9.** *Propagate Function*  $\text{Propagate} : \text{PRO} \times 2^{\text{POR}} \times 2^{\text{POR}} \rightarrow \mathbb{L}$  is a function that produces labels for the outputs of an activity using the labels of its inputs. Its specification is as follows:

$$\text{Propagate}(p, S, T) = \bigcup_{\langle a, s, t \rangle \in \{a \mid \text{invocations}(p, a)\} \times S \times T} \{ \langle d, v, f \rangle \mid \exists e \exists f (\mathbb{L}_w(d, v, e) \wedge \text{input}(a, s, e) \wedge \text{output}(a, t, f)) \}$$

*Propagate* accepts as input a processor  $p$ , a set of source/input ports of  $p$ ,  $S \subset \text{POR}$ ; a set of target/output ports  $T \subset \text{POR}$ . Within *Propagate* for each invocation  $a$ , source port  $s$  and target port  $t$  combination we obtain all labels of the entity  $e$  used by the activity  $a$  at source port  $s$ . For each label bound to  $e$ , we create a label bound to the output  $f$  at target  $t$  of  $a$ .

**Definition 10.** *Generalize Function*  $\text{Generalize} : \text{PRO} \times \text{POR} \times \mathbb{N}^+ \rightarrow \mathbb{L}$  is a function that produces labels for a collection entity using the labels of its items at a designated nesting level. Its specification is as follows:

$$\text{Generalize}(p, r, n) = \bigcup_{\{a \mid \text{invocations}(p, a)\}} \{ \langle d, v, c \rangle \mid \exists k \exists c (\mathbb{L}_w(d, v, k) \wedge \text{output}(a, r, k) \wedge \text{hadMember}(c, m, n)) \}$$

*Generalize* accepts as input a processor  $p$ , an output port of  $p$ ,  $r \in \text{POR}$ , and a nesting level  $n$ . For each invocation  $a$  of  $p$  we obtain the labels attached to the output entity  $k$  generated by the activity at port  $r$ . For each label bound to  $k$ , we create a label bound to the collection entity  $c$  that holds  $k$  as its item at nesting level  $n$ .

**Definition 11.** *Distribute Function*  $\text{Distribute} : \text{PRO} \times \text{POR} \times \mathbb{N}^+ \rightarrow \mathbb{L}$  is a function that copies over labels of a collection entity to items at a designated nesting level. Its specification is as follows:

$$\text{Distribute}(p, r, n) = \bigcup_{\{a | \text{invocations}(p, a)\}} \{ \langle d, v, i \rangle \mid \exists k \exists i (\mathbb{L}_w(d, v, k) \wedge \text{output}(a, r, k) \wedge \text{hadMember}(k, i, n)) \}$$

*Distribute* accepts as input a processor  $p$ , an output port of  $p$ ,  $r \in \text{POR}$ , and a nesting level  $n$ . For each invocation  $a$  of  $p$  we obtain the labels attached to the output entity  $k$  generated by the activity at port  $r$ . For each label bound to  $k$ , we create a label bound to the item  $i$  that is a member of collection  $k$  at nesting level  $n$ .

In practice a *Label* is an object that adheres to the *LabelInstance* class given in Figure 8. Labels are comprised of a *definition*, *target* and *value*. The *target* and *value* (both of type *String*) represent a simple key-value based metadata structure. The *target* uniquely identifies a data artefact, which the label describes and the *value* holds the annotation content. A label is typified its *definition*. A *LabelVector* is a named collection of label definitions. We use label vectors to configure the execution of labelling operators. A label vector informs label propagation operators to the kinds of labels they should pick up from parts of the provenance graph and propagate to other parts. Label and label vector definitions would be specific to each scientific domain or investigation, and can be used to decorate workflows from these domains. In the following section we elaborate how we practically deliver the labelling functions, which we outlined formally.

*LabelFlow* is a generic framework, which requires configuration for use in a particular domain. Formally;

**Definition 12.** *LabelFlow* is the tuple  $\langle O, T \rangle$ , where:

$O = \{\text{MINT}, \text{PROPAGATE}, \text{GENERALISE}, \text{DISTRIBUTE}\}$  is the set of labelling operators.

$T$  is a tool that can take as input a scientific workflow  $w$  and generate a labelling pipeline for that workflow  $\Pi_w$ .

**Definition 13.** A particular configuration of *LabelFlow* is the 8-tuple  $\langle O, T, F, w, \Pi_w, P_w, D_w, L_w \rangle$  where:

$F$  is a set of domain-specific labelling functions.

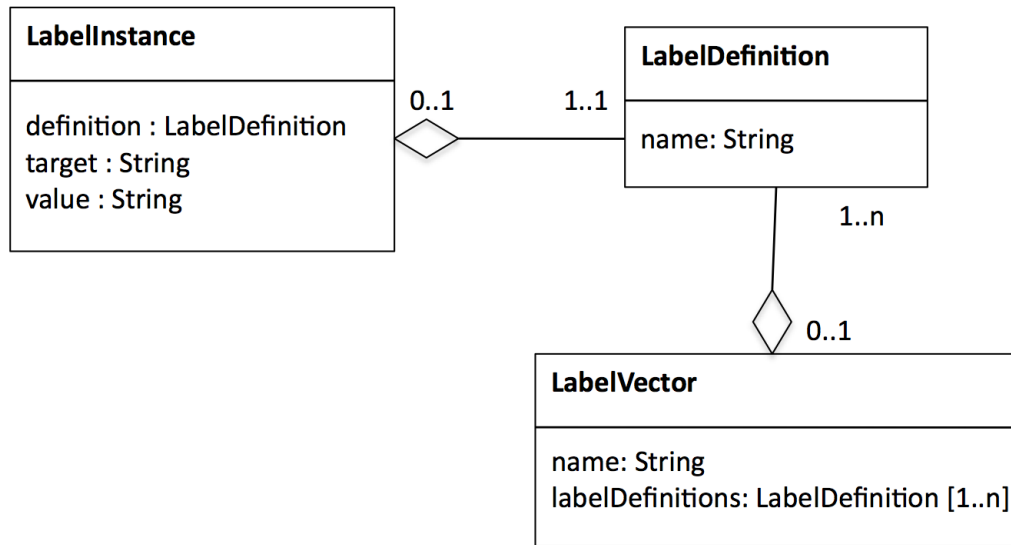
$w$  is a Motif annotated scientific workflow.

$\Pi_w$  is the labelling pipeline for  $w$ .

$P_w$  is a provenance trace for a particular execution of  $w$ .

$D_w$  is a data trace for a particular execution of  $w$ .

$L_w$  is an initially empty label space that supports the information model given in Section 5 to hold labels generated during execution of  $\Pi_w$ .



**Figure 8.** UML Class Diagram denoting information model of Labels.

### 5.1. Labelling Operators

We will now describe *MINT*, *PROPAGATE*, *GENERALISE*, *DISTRIBUTE* operators these are implementations of the labelling behaviours specified formally in the previous section (Definitions 8–11). For each operator, we give a high-level behavioural view as UML activities [39] and provide an algorithmic specification. The auxiliary methods used by operators are given in the Appendix. We also provide UML’s Activity diagram notation reference in the Appendix.

**Definition 14.** *MINT Operator is a computational process that is configurable as given in Figure 9. It accepts as input a processorId, a functionId, and a targetList. The processorId is the identifier of an (analytical) task in workflow  $w$ , whose provenance and associated data artefacts will be exploited to generate labels. The functionId is the identifier for the domain-specific label provisioning function. targetList contains identifiers of those output ports of the designated task, that will be the target of labels generated.*

The *MINT* operator reads data trace  $D_w$  and provenance trace  $P_w$ , and upon execution submits the labels to the label space  $L_w$ . The procedural specification of this operator is given in Algorithm 1. Here we obtain all invocation records of the processor designated by the *processorId*, for each invocation record we obtain all data related to that invocation (inputs/outputs) As Taverna uses the file system for its data storage layer, these are references to files. We submit data to the labelling function named *functionId*. The function returns a set of labels that are to be associated with the target outputs, finally we associate these labels with all data artefacts that have appeared at a port in the *targetList*. The computational complexity of Algorithm 1 is  $O(t.N)$  where  $N$  is the number of invocations of *processorId* and  $t$  is a constant denoting the number of *targetList*. Scientifically significant processors typically have a single output port so often  $t = 1$ .

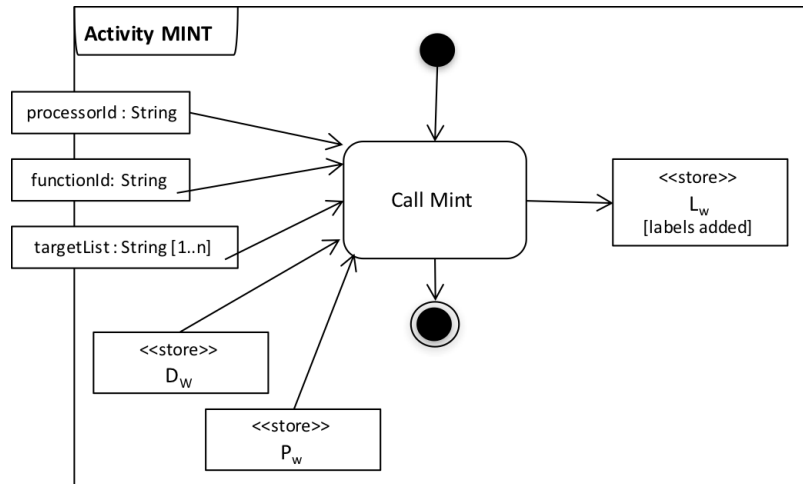


Figure 9. Mint Operator Specification.

**Algorithm 1:** Mint

---

**Input:** *processorId*, *functionId*, *targetList*  
 $labellingFunction \leftarrow \text{getFunctionFromRegistry}(functionId);$   
**foreach** *activity* **in**  $\text{getInvocations}(processorId, provStore)$  **do**  
   $activityData \leftarrow \text{getAllActivityData}(activity, provStore, dataStore);$   
   $Labels \leftarrow labellingFunction.invoke(activityData);$   
  **foreach** *output* **in** *targetList* **do**  
     $outData \leftarrow \text{getActivityOutData}(activity, output);$   
     $BoundLabels \leftarrow \text{bindLabelsToData}(outData, Labels);$   
  **end**  
   $\text{submitLabels}(BoundLabels, labelStore);$   
**end**

---

**Definition 15.** *PROPAGATE Operator is a computational process, that is configurable as given in Figure 10. It accepts as input a processorId, a srcList and a targetList. The processorId is the identifier of an (adapter) task, which has a Motif implying a value copying relation from its inputs to its outputs (recall Table 3). The srcList contains identifiers of those input ports of the designated task, from which labels are to be picked up. The targetList contains identifiers of output ports to which labels shall be propagated.*

The *PROPAGATE* Operator reads the label space and provenance trace  $L_w$  and  $P_w$  and updates the label space  $L_w$  with propagated labels. The procedural specification of this operator is given in Algorithm 2. Here we first obtain the invocation record of processors with designated *processorId*, for each invocation we obtain the labels of data nodes at a source port in *srcList*, we aggregate them with set Union and, finally we associate these labels with all data artefacts that have appeared at a port in the *targetList*. The practical utility of the Label Vector comes into play during propagation. We anticipate that re-usable domain specific labelling functions will generate labels exhaustively. They will mint labels for all recognisable metadata in a data file. In a workflow setting however we may not want to propagate all of those labels to all data copies in the scientists workspace. We therefore limit propagation by focusing on the labels, whose definition are in the label vector. The Label Vector would typically contain label types that represent input parameters/configurations of a workflow/processor. In our case study these are subject the data is about, the coordinates of the object and calibration settings for analytical steps (e.g., morphology). The computational complexity of Algorithm 2 is  $O(s.t.N)$  where  $N$  is the number of invocations of *processorId* and  $s$  and  $t$  are constants denoting the size of *srcList* and *targetList*, i.e., input output ports among which label propagation occurs.

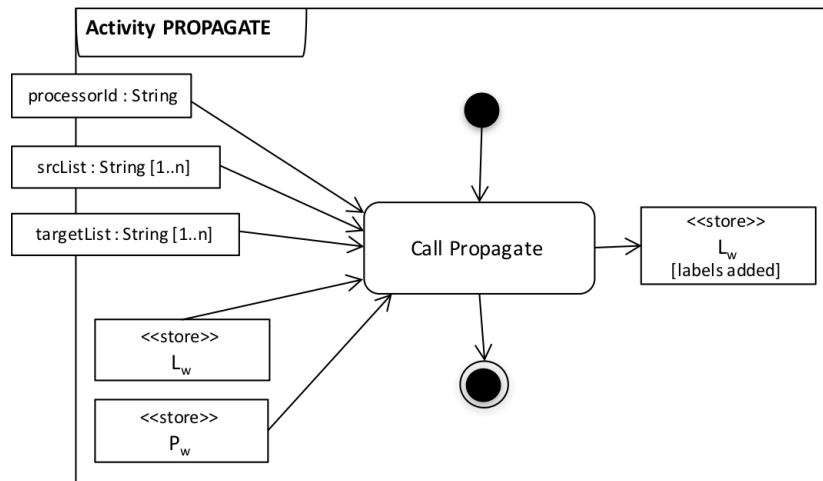


Figure 10. Propagate Operator Specification.

**Algorithm 2:** Propagate

---

```

Input: processorId, srcList, targetList
foreach activity in getInvocations(processorId, provStore) do
    LabelDefinitions ← getLabelVector();
    foreach output in targetList do
        foreach outData in getActivityOutData(activity, output, provStore) do
            Labels ← ∅;
            foreach input in srcList do
                foreach inData in getActivityInData(activity, input, provStore) do
                    Labels ← Labels ∪ getLabels(inData, LabelDefinitions, labelStore);
                end
            end
            BoundLabels ← bindLabelsToData(outData, clone(Labels));
        end
        submitLabels(BoundLabels, labelStore);
    end
end

```

---

**Definition 16.** *DISTRIBUTE/GENERALISE Operators are computational processes, that are configurable as given in Figure 11. They accept a processorId, an src and a depthDifference. These operators are designed to propagate labels up and down the structure hierarchy of collection-typed data artefacts in provenance. While the MINT and PROPAGATE are labelling proxies for tasks, these are labelling proxies for dataflow links in the workflow, specifically those links with structural data type mismatches between the ports at the link's two ends. The processorId and src parameters jointly identify an output port of a particular processor (the source end of a mismatched datalink). The level of mismatch is specified with the depthDifference.*

Consider the case where one processor, by definition, produces an output of a collection, which has depth 1 and is linked to a follow-on processor that consumes single items (i.e., of nesting depth 0). This case corresponds to a *depthDifference* of 1 among two ends of a dataflow link. In order to adjust for this mismatch we would have to push down the labels associated with the output collection occurring at port *src* of the designated processor to the collection's items that are two-level deep in the data structure. We achieve this by using the *DISTRIBUTE* operator. The reverse procedure of pulling up labels is performed by the *GENERALISE* operator. Similar to the *PROPAGATE* operator the



*DISTRIBUTE*/*GENERALISE* operators read the label space  $L_w$  and provenance trace  $P_w$  and update the label space  $L_w$ . The procedures that for these operators are given in Algorithms 3 and 4 respectively.

The computational complexity for Algorithms 3 and 4 is  $O(N)$  where  $N$  is the number of invocations of *processorId*. We do not use the *getInvocations* method as we did in other operators. We instead use *getAllGeneratedOutputs* which returns outputs from all invocations. We then loop over this collection to push down or pull up labels to other entities at designated nesting levels. These other entities are obtained by a single method call (*getEnclosingCollection* in the case of *GENERALISE* and *getItems* in the case of *DISTRIBUTE*).

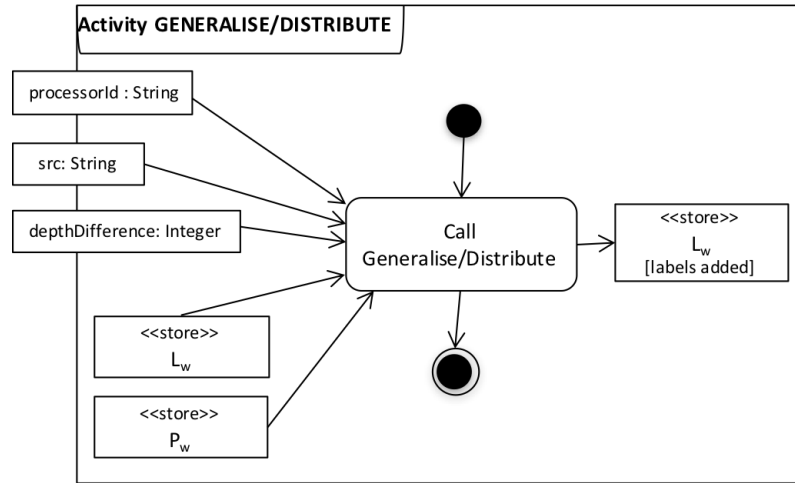


Figure 11. Distribute and Generalise Operator Specifications.

---

#### Algorithm 3: Distribute

---

**Input:** *processorId*, *src*, *depthDifference*  
 $LabelDefinitions \leftarrow getLabelVector();$   
**foreach** *outData* in *getAllGeneratedOutputs*(*processorId*, *src*, *provStore*) **do**  
   $Labels \leftarrow getLabels(outData, LabelDefinitions, labelStore);$   
  **foreach** *item* in *getItems*(*outData*, *depthDifference*, *provStore*) **do**  
     $BoundLabels \leftarrow bindLabelsToData(outData, clone(Labels));$   
     $submitLabels(BoundLabels, labelStore);$   
  **end**  
**end**

---



---

#### Algorithm 4: Generalise

---

**Input:** *processorId*, *src*, *depthDifference*  
 $LabelDefinitions \leftarrow getLabelVector();$   
 $OutData \leftarrow getAllGeneratedOutputs(processorId, src, provStore);$   
**foreach** *coll* in *getEnclosingCollections*(*OutData*, *depthDifference*, *provStore*) **do**  
   $Labels \leftarrow \emptyset;$   
  **foreach** *item* in *getItems*(*coll*, *depthDifference*, *provStore*) **do**  
     $Labels \leftarrow Labels \cup getLabels(item, LabelDefinitions, labelStore);$   
  **end**  
   $BoundLabels \leftarrow bindLabelsToData(coll, clone(Labels));$   
   $submitLabels(BoundLabels, labelStore);$   
**end**

---

## 5.2. Labelling Pipelines

In order to put the capability encapsulated by operators into action we use labelling pipelines. We formally define the elements of pipeline generation as follows:

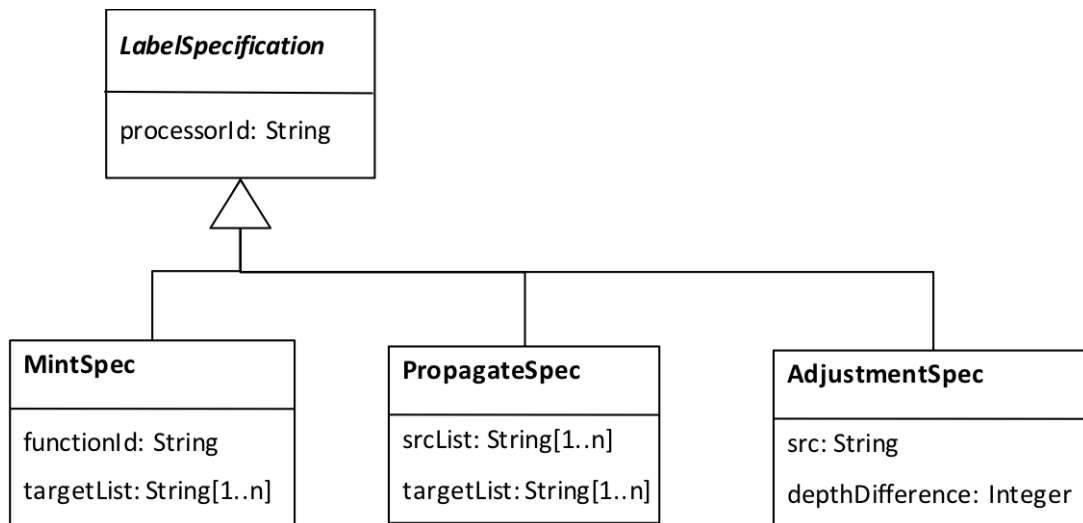
**Definition 17.** *Labelling Specification, Annotated Workflow* An annotated workflow  $w$  is the triple  $\langle PRO, POR, LINK \rangle$  together with the functions  $pSpec$  and  $lSpec$ .  $pSpec : PRO \rightarrow \mathbb{S}$ , maps processors of the workflow to their Labelling Specification, which encapsulates information necessary for a labelling operator to run. The range of  $pSpec$  is the string representations of the domains of Mint and Propagate (recall from Definitions 8 and 9;  $dom(Mint) = PRO \times 2^{POR} \times \mathbb{S}$  and  $dom(Propagate) = PRO \times 2^{POR} \times 2^{POR}$ ).  $lSpec : LINK \rightarrow \mathbb{S}$  maps dataflow links to their labelling specifications. The range of  $lSpec$  is the string representations of the domains of Generalize and Distribute (recall from Definitions 10 and 11;  $dom(Generalize) = PRO \times POR \times \mathbb{N}^+$  and  $dom(Distribute) = PRO \times POR \times \mathbb{N}^+$ ).

In practice, a *Labelling Specification* is an object that is an instance of concrete classes given in Figure 12. Notice that the *MintSpec* and *PropagateSpec* represent information passed as input to the respective operators, whereas the *AdjustSpec* represents the input of *GENERALISE* and *DISTRIBUTE* operators. A labelling specification of an annotated processor can be of types *MintSpec* or *PropagateSpec* only.

**Definition 18.** *Labelling Pipeline Generator*  $T$  is a tool provided as part of *LabelFlow* that accepts as input an annotated workflow  $w$  and produces as output a labelling pipeline  $\Pi_w$  for  $w$ .

**Definition 19.** *Labelling Pipeline* is a specification for a computational process comprised of (1) sub-processes based on calls to *MINT*, *PROPAGATE*, *GENERALISE* and *DISTRIBUTE* operators (as per Definitions 14–16) and (2) control-flow relations among those processes. So  $\Pi_w = \langle OP, CTRLINK \rangle$ .

We will first illustrate labelling pipelines and later discuss how pipeline generator works.



**Figure 12.** UML Class Diagram denoting information model of Labelling Specifications.

### 5.2.1. Example Labelling Pipeline

For the case-study workflow given earlier in Figure 1 the labelling pipeline is given in Figure 13 using UML Activity Diagram notation. For each of the three scientifically significant activities in the workflow, namely *SesameXML*, *VII\_237* and *calculate\_internal\_extinction* (in Figure 1) there is a corresponding *MINT* process in the labelling pipeline (in Figure 13). This is because all three

processors had associated labelling specifications (extended Motif annotations) of type *MintSpec* (in Figure 12). Label specifications of these processors have become sets of constant-valued inputs (denoted with value-pins) for each corresponding *MINT* process in the pipeline. For example for the *SesameXML* step in the workflow, the corresponding *MINT* process is configured with:

- String value of “SesameXML” for the *processorId* input parameter.
- String value of “SesameLabeller” for *functionId* input parameter,
- a Set containing the String value “return” for *targetList* input parameter.

Using this input triple the *MINT* process is undertaken by calling the operation detailed in Algorithm 1, which will decorate data outputs that appear at the port named “return” of processor “SesameXML” with labels obtained through invocation of domain-specific function “SesameLabeller”. Recall from the specification of *MINT* process that it reads from data and provenance traces  $D_w$  and  $P_w$  and writes to the label space  $L_w$ . In addition, unlike all other operators, *MINT* does not read from the label space as it generates labels in the first place. As a result *MINT* processes can start simultaneously upon the start of labelling pipeline (denoted with a fork of control links from start node to all three *MINT* processes).

For some of the data adapter steps in our case-study workflow, namely *Extract\_RA*, *Extract\_DEC*, *Select\_logr\_Mtype*, *Flatten\_List*, *Flatten\_List\_2*, we have *PROPAGATE* processes in the labelling pipeline. The labelling specifications, of type *PropagateSpec*, associated with these adapter processors, has become input configurations for the *PROPAGATE* processes in the pipeline. The labelling specifications denote from which input ports (*srcList*) to which output ports (*targetList*) label propagation should occur. Note that the *Format\_conversion* step in our workflow, despite being a data adapter having the *FormatTransformation* Motif, does not have a corresponding labelling process in the pipeline. This is because, as per Table 3, *FormatTransformation* is not a Motif for which a labelling behaviour has been defined. Consequently *Format\_conversion* step does not have an associated labelling specification and therefore has no footprint in the labelling pipeline. As per its specification the *PROPAGATE* process reads from and write to the label space  $L_w$ . In order for a *PROPAGATE* process to run, all other processes in the pipeline that decorate data at ports in the *srcList* parameter of that *PROPAGATE* process shall be completed. This requirement is represented with control flow links among relevant processes in the pipeline.

The pipeline in Figure 13 also contains a *GENERALISE* and *DISTRIBUTE* processes to propagate labels along data structure hierarchies in cases of mismatched data types at the two ends of a dataflow link. One example is the *GENERALISE* process, which is configured to propagate labels of the *nodeList* output of *Extract\_RA* processor one level up to their enclosing collection, as it is these collections that get consumed by the follow-on processor *Flatten\_List* in the workflow. There is a difference in the way *GENERALISE/DISTRIBUTE* processes are included in a labelling pipeline when compared to the way *MINT* and *PROPAGATE* processes are included. *MINT* and *PROPAGATE* processes are directly informed by annotations in the form of labelling specifications (either a *MintSpec* or a *PropagateSpec*) associated with processors in the workflow. On the other hand there is no such annotation for the *GENERALISE/DISTRIBUTE* processes. Their inclusion happens through an analysis of dataflow links in the workflow and the corresponding creation of labelling specifications of type *AdjustmentSpec* (Figure 12). We discuss the details of labelling pipeline creation in the next section.

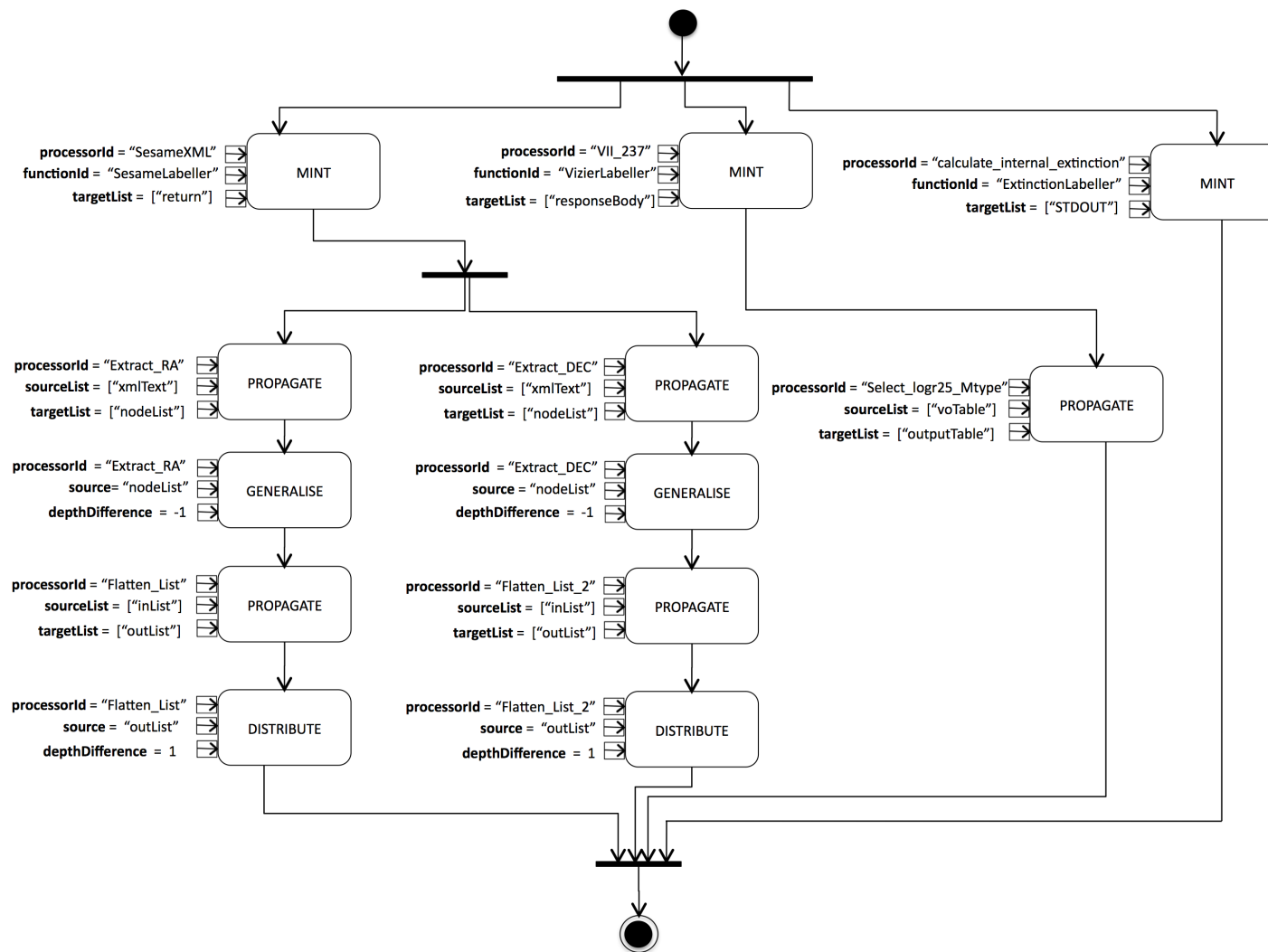


Figure 13. The labelling pipeline for the case-study workflow.

### 5.2.2. Pipeline Generation Procedure

The procedure followed by the Labelling Pipeline Generator is given in Algorithm 5. Inputs *wfProcessors* and *wfDatalinks* are the set of Processors and Datalinks that make up a workflow *w*. The procedure initialises two empty collections *pipelineOps* and *pipelineCtrlLinks* to hold the Labelling Operators and the Control Links within the result pipeline  $\Pi_w$ . The procedure is comprised of two phases for populating these two collections.

The first phase begins by traversing all processors of *w* to check whether that have associated with them a labelling specification. If that is the case then the labelling specification is transferred to  $\Pi_w$ , more specifically it will be added to the *pipelineOps* collection. In case a processor in the workflow has no labelling specification associated, then it will simply be skipped. As a follow-on step we eliminate dangling *PROPAGATE* operators. Dangling operators are those that are configured to obtain labels from source ports, where no labelling operator is configured to populate. The final step in the creation of labelling operators is the addition of *GENERALISE* or *DISTRIBUTE* type adjustment operators. We do this by iterating over every datalink in workflow *w* (items of *wfDatalinks*). We check whether the source of the datalink is being labelled by any of the *MINT* or *PROPAGATE* type operators. If that is the case, and if the datalink is one which is unbalanced due to mismatched datatypes of ports at its two ends then we create a corresponding adjustment specification either *DISTRIBUTE* or *GENERALISE* and add it into the *pipelineOps* collection.

In the second phase we create control flow links. We do this by iterating over operators in the pipeline, finding each the operator's predecessor operators and creating control links among them. The predecessors of a *PROPAGATE* operator can be multiple and they are those that have as their labelling target a port that is in the source port list of *PROPAGATE*. Adjustment type operators *GENERALISE* or *DISTRIBUTE* have a single predecessor, which is the one that has as labelling target the source port of adjustment operator.

We represent the labelling pipeline from this procedure with the Wfdesc workflow model [32]. Note that the pipeline is comprised of operators and control flow links. The repetitive application of labelling for multiple invocations of processors and for multiple label kinds in a label vector are handled within the labelling operators. Therefore the basic model of Wfdesc in representing processes and their dataflows is sufficient for us in representing our pipeline. The details of how this abstract representation is mapped to a concrete executable form is provided in the following Section.

### 5.3. Implementation

We have implemented *LabelFlow* in Java [40]. The auxiliary functions that *LabelFlow* uses to access the provenance and label spaces (given in Table A1) are methods of a single Java class. Our default implementation supports a PROV-O [41] based RDF representation of provenance. We have chosen PROV-O and RDF because they are the most common encoding among PROV implementations listed in [42]. We use SPARQL queries to implement the provenance accessor methods within Table A1. For performing the query precision analysis in our case study we implemented provenance queries as Java methods that build upon the auxiliary functions of *LabelFlow* in Table A1. As provenance traversal is a rather standard process we do not elaborate on these query methods. Querying provenance in its native storage form (be it Relational, RDF or XML) is always undoubtedly more efficient than querying through APIs. We have relied on an API as we wanted to abstract away from any storage technology for labelling and querying is an aid to the labelling process.

**Algorithm 5:** Pipeline Generation

---

```

Input: wfProcessors, wfDatalinks
pipelineOps  $\leftarrow \emptyset$ ;
pipelineCtrlLinks  $\leftarrow \emptyset$ ;
/* PHASE-1: Create Labelling Operators */
foreach processor in wfProcessors do
    if hasLabellingSpec(processor) then
        | pipelineOps  $\leftarrow$  getLabellingSpec(processor);
    end
end
do
    found  $\leftarrow$  false;
    danglingOperator  $\leftarrow$  null;
    foreach op in pipelineOps do
        if isPropagate(op) and isDangling(op) then
            | danglingOperator  $\leftarrow$  op;
            | found  $\leftarrow$  true;
            | break;
        end
    end
    if found then
        | remove(pipelineOps, op);
    end
    while found;
    foreach link in wfDatalinks do
        if isLinkSourceLabelled(link, pipelineOps) & isImbalanced(link) then
            | pipelineOps  $\leftarrow$  createAdjustmentSpec(link);
            | break;
        end
    end
/* PHASE-2: Create Control Links Among Operators */
foreach op in pipelineOps do
    if isPropagate(op) then
        | foreach src in getSrcList(op) do
            | predecessorOp  $\leftarrow$  getOperationWithTarget(src);
            | if predecessorOp  $\neq$  null then
            | | pipelineCtrlLinks  $\leftarrow$  createCtrlLink(predecessorOp, op);
            | end
        | end
    end
    else if isGeneralise(op) or isDistribute(op) then
        | predecessorOp  $\leftarrow$  getOperationWithTarget(getSrc(op));
        | pipelineCtrlLinks  $\leftarrow$  createCtrlLink(predecessorOp, op);
    end
end

```

---



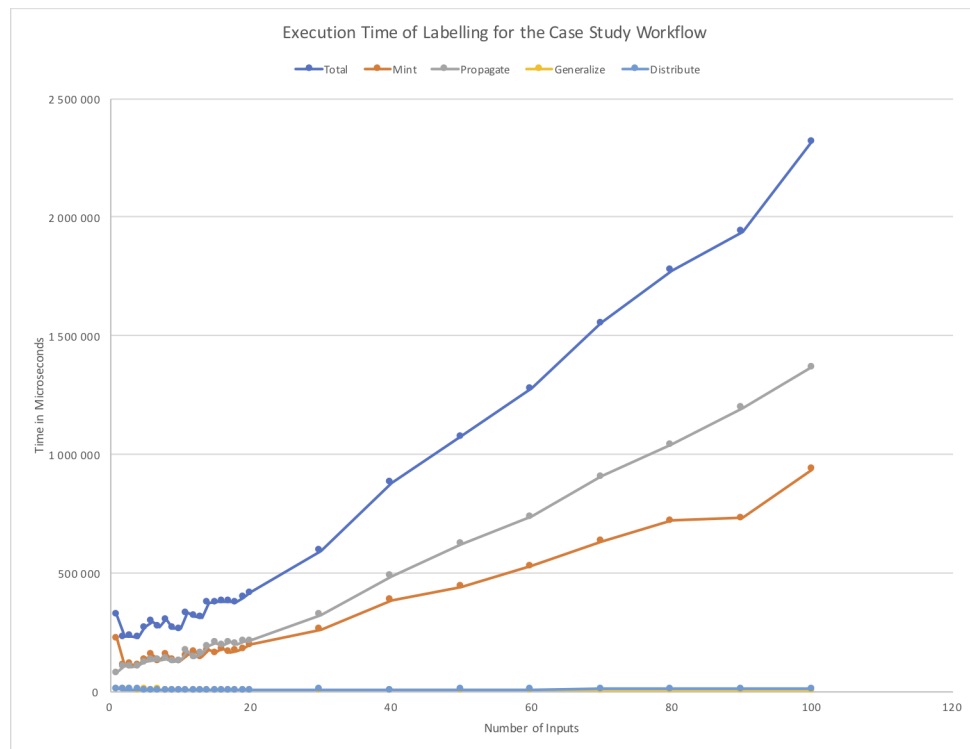
In our default (RDF-based) implementation a label definition is represented with an OWL Datatype property. Label instances are RDF statements where the subject corresponds to the target of the label, the predicate is the datatype properties and objects are metadata values of type *xsd:string*. The Provenance and Label spaces are RDF files in *turtle* syntax. For each workflow run we load these into in-memory Jena [43] models. These could also be loaded into a single model in case native (SPARQL) querying of labelled provenance is desired. The sample provenance and label spaces used in the performance evaluation of *LabelFlow* can be accessed from the source code repository here [40]. We have tested the PROV compliance of provenance traces we use with the ProvToolbox online validator [44].

Labelling pipelines are represented in an abstract manner with Wfdesc [32]. This abstract representation can be converted to a concrete executable form using any workflow language that supports a simple data flow among activities, and can access resources exposed through Java APIs. For our case-study tests we interpret the abstract Wfdesc specification by traversing the activities in the topological order of their respective workflow elements in the scientific workflow, and make API calls to invoke respective operators. Note that the pipeline is only responsible for coordinating the execution of labelling operators, whereas the core of labelling takes place within operators.

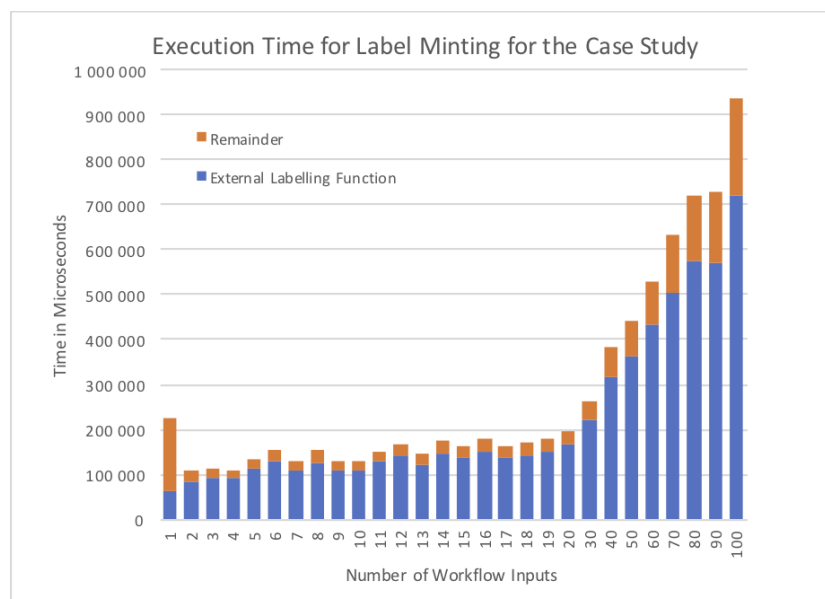
#### 5.4. Performance of *LabelFlow*

We did a performance evaluation of *LabelFlow* by running the case study workflow with increasing number of inputs. The evaluation was performed on a MacBook Pro with 2.9 GHz i7 CPU and 16 GB RAM. Figure 14 presents the execution time (in micro seconds) for the entire total and four each kind of operator. As expected *LabelFlow* performance is linearly dependent to the number of processor iterations that are in turn driven by the number of workflow inputs. For our case study workflow Label Minting and Propagation take up almost the all of the time of labelling. Mint and Propagate operators operate on data of processors, whose iteration increase as the input increases. On the other hand the Generalize and Distribute operators are always executed a single time because the processors they are associated with (the infamous *Flatten\_List* steps causing the n-by-m pattern in provenance) are executed once, even though the input size increases. Also recall from Section 2.1 that for the single input case ( $n = 1$ ) some processors (and consequently their associated propagate/mint operators) run  $2n$  times. So even in the case of single input Propagation takes more time than Distribution or Generalization. Ultimately the break down of the total labelling time into categories would be dependent on the workflow. As stated in Section 3, our earlier empirical survey of workflows revealed that an average of 70% of activities is data adaptation. So for each workflow we can anticipate that labelling workflows for those workflows would contain the propagate operator to that degree, and that propagation would be the most significant portion of labelling.

In Figure 15 we present the percentage of time taken by the domain-specific labelling functions within the time taken in the mint operator. Performance of minting will largely determined by the performance of these external functions, which would need to perform IO on the data files. For our case study for all input sizes external function took up more than 60% of total minting time.



**Figure 14.** Total execution time of labelling for our case study workflow in increasing number of iterations.



**Figure 15.** Percentage of the time taken by external Mint function within the total time taken by the Mint operator.

## 6. Revisiting Case-Study

In order to assess the benefit of labels in reporting we have used *LabelFlow* to annotate execution traces of our case-study workflow. As prerequisite to obtaining a labelling pipeline for this workflow we performed the following:

- we implemented three simple domain-specific labelling functions, one for each scientifically significant step in the workflow (as discussed in Section 5.2.1). These functions can parse the

data consumed/generated by these activities and create labels that correspond to either input configurations (context) or data origin.

- we associated Labelling Specs with workflow activities according to the information model given in Figure 12. For the data adapter activities, for which a corresponding labelling behaviour is given (Table 3), we created *PropagateSpecs* and for the scientifically significant activities, we created *MintSpecs*, pointing to the labelling functions.

This time we implement the queries in Table 2 using label-based annotations (denoted with the “-A” suffix). The precision in obtaining relevant results for each query is given in Figure 16.

**Q1-A** Rather than using lineage as a pseudo mechanism to seek coordinates retrieved from Sesame Database, we now inquire about data origin directly using labels. Formally;

```
answer(E,C) :- L_ins("referenceURI", "http://cdsws.u-strasbg.fr/",E),
               L_ins("referenceCatalog", C, E)
```

We use *referenceURI*, *referenceCatalog* datatype properties created for our case study. Note we are now able to fully implement the query and seek the source catalog information about the coordinates. As Figure 16 shows, with label-based queries we are able to retrieve with 100% true accuracy the data that comes from the Sesame database and its local copies.

**Q2-A** In this query we use the *hasSubject* label to seek results about a particular galaxy. Formally;

```
answer(E,G) :- L_ins("hasSubject",G,E), member(G,["M31","UGC 454"]),
               L_ins("referenceCatalog", G, E)
```

Note that a typical characteristic of scientific data repositories is that they use heterogeneous identification schemes. So, in Astronomy a Galaxy has several identifiers from respective databases. The Visier and Sesame databases accesses within our example workflow use different identifiers. Therefore in our query, we need to refer to all possible domain identifiers of a Galaxy. As seen from Figure 16, the precision deteriorates as it was the case with Q2-G. A combination of broken factorial design (at the *Flatten\_List* step) and liberal label propagation causes inaccurate labels to be created. While each output from “SesameXML” bears the correct label denoting the associated galaxy, all items in the output of “Flatten\_List” would bear a set of labels (for all galaxies), even though each contains the data of one. Recall from our case-study that iteration is not sustained at the *Flatten\_List* step, in other words it is executed only once consuming all galaxy coordinates. Meanwhile as per its Motif, we know that this step builds its output by coalescing all items in the input collection. As a result our labelling pipeline will first generalise all labels and compute a label for the top-level collection element consumed by *Flatten\_List*. This label will get propagated to *Flatten\_List*’s output, which is a list. On the other hand, this list is not consumed as a whole by downstream activities, instead each item in it is used. Therefore each item inherits the labels from the enclosing list (through a distribute operator). As a result, we end up with inaccurately labelled items.

**Q3-A** We are now able to fully represent Q3. Let  $x$  denote the *calculate\_internal\_extinction* processor of our workflow then Q3:

```
answer(O) :- PRO(x),invocations(x,A), input(A,_,I), output(A,_,O),
             L_ins("hasSubject", G, I1), member(G,["M31","UGC 454"]),
             L_ins("hasMorphology", "0.45", O)
```

The labelling function for the *calculate\_internal\_extinction* step creates *hasMorphology* labels for the output to capture the context represented by the morphology input parameter. When we look at the precision of Q3-A it also deteriorates with increasing inputs. This is because the coordinate inputs to the extinction calculation are inaccurately labelled due to upstream *Flatten\_List* step.

As discussed in Section 2.3 the existence of the n-by-m pattern (or the lack of discrete traceability) is a critical characteristic that determines the utility of provenance. Even after labelling, we observe the same sharp decrease in precision of Q2-A and Q3-A (in Figure 16).

We now review related work, followed by a critical discussion of our case-based assessment.

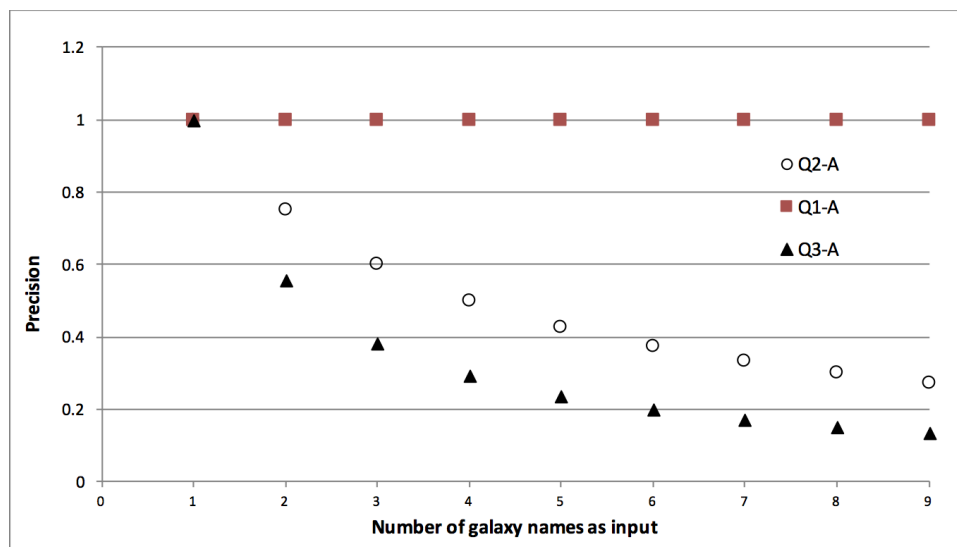


Figure 16. Precision of results when queries are run over annotated (labelled) provenance.

## 7. Related Work

*LabelFlow* brings together two capabilities, (1) the provisioning of domain-specific annotations by promoting data to become metadata, and (2) the propagation of annotations among nodes in the provenance graph. We therefore review related work in these two categories.

### 7.1. Obtaining Annotations

In early works on provenance annotation [45,46], the primary focus has been on capturing (through manual annotations) the static metadata, characterising elements of a workflow description and propagating those characteristics to execution provenance.

Cao et al. [16] were the first to focus on dynamic metadata. This work brings annotation capabilities to a desktop application that allows users to perform analyses by interacting with remote services available on a Grid. The authors propose the use of specialist *Annotators* that crawl over data nodes in a provenance graph that are known to be of a specific domain type (e.g., a BLAST [47] report from Bioinformatics). Annotators can parse data values in known formats and can create annotations using data values. Here the objective is to create metadata exhaustively by exploiting all possible metadata headers/fields in a data file. On the other hand the authors do not discuss how this rich metadata will be utilised by the application.

In [48] Sahoo et al describe the SPADE system, where they highlight dynamic metadata, and they too exploit data artefacts as the source of metadata. The authors propose using “semantic provenance modules”, similar to Cao’s domain-specific annotators to create elaborate metadata. They propose such modules be inserted in-between analytical steps in workflows. Similar to Cao’s work, SPADE focuses on providing one particular domain-specific ontology and elaborate metadata conforming to that ontology. Note that this approach requires altering the original scientific workflow to denote points of interruption, where the annotator will execute. The details of SPADE implementation is not available [48], given their ontology, we presume the resulting metadata would be rich semantic annotations

In a recent paper De Oliveira et al. [49] question “how much domain data should be in provenance?”. Their answer is that it should be under the control of the user. De Oliveira’s approach, named ARMFUL in a follow-on paper [23] is a dynamic metadata provisioning capability tailored to support parallel scientific workflow systems. Similar to our labelling functions, they associate

user-designed *Extractor* classes with outputs of selected workflow activities. These extractors utilise raw data indexing techniques (e.g., FastBit) to generate metadata in the form of attribute-value pairs. Similar to SPADE, De Oliveira's approach requires alterations to the original workflow to denote the extractor class per activity. The need to embed the metadata generation capability into the workflow is justified by a need to have and query this metadata while the workflow is still running. In the parallel workflow setting workflow activities are long running, and one way to detect anomalies in the execution is to retrieve intermediary results based on domain specific characteristics and to inspect them. ARMFUL and SPADE systems are closest in spirit to *LabelFlow* as they combine domain-specific metadata with dataflow information.

The distinctive aspects of *LabelFlow* compared these are:

- it is non-intrusive to the execution of workflow. As the metadata is sourced from the data, as long as the data values are kept, annotation can take place as an offline process any later time. As it is offline, however, *LabelFlow* may miss out on metadata that is available only at runtime and does not get serialised into task outputs.
- is not tied to a particular raw data form or indexing technology. In our survey of workflow Motifs [24] we observed that metadata is not always presented as additional columns in a tabular layout. It is often found in dedicated headers as in a Blast Report, a Variant Call File or a NIFTI file header for fMRI data.
- it focuses on capturing the context that surrounds a particular analytical activity. All prior approaches focus on extracting metadata that they assume exists within raw data. As illustrated in our case study workflow, context may not be consistently available within the data, therefore we focus on the cases where the context is spread out among input parameters and data copies. As we use labelling functions that consume all data (input/output) associated with an activity, we provide a mechanism to weave back this context and propagate it to data copies.

## 7.2. Propagating Annotations

Attribute propagation has been first studied in the context of part-whole relations in Object-Centered systems [50] and in Description Logics [51], where attributes of parts can be considered attributes of wholes and vice versa. In [50] authors describe an attribute propagation mechanism in Object-Oriented databases that exploits the part-whole relations. Two types of propagation is outlined *invariant* and *transformational*, where the latter is typically used to aggregate attributes of parts to obtain an attribute for the whole (e.g., a car's weight is the sum the weight of all its components).

Metadata propagation has been explored in digital library research. In [52] authors accelerate the curation of shared research work products through propagation of basic metadata, such as authorship, subject, or publication date, from the research articles to their supplementary material (data, visualisations, charts). Such propagation may result in incorrect annotations (e.g., not all charts of a paper may have been authored by the same person). Authors propose that inaccuracies are later corrected via manual curation.

Propagating annotations of data artefacts to other data artefacts by exploiting provenance has been studied in the context of databases. In DBNotes [53] authors track query results to corresponding source tuples for Select-Project-Join-Union queries. DBNotes uses set union to gather all annotations over source tuples to obtain an annotation set for the result. In [54] authors describe a logic-based approach, which exploits semantics of relational query operators to propagate of schema-level semantic annotations through queries. Rules for propagation of annotations through each relational operator is represented as a logic constraint. Authors speculate that such an approach can find applicability in semi-automated annotation of workflows.

What sets *LabelFlow* apart from annotation propagation over white-box database provenance is that it operates over grey box provenance. In a scientific workflow setting, data structures and computations are diverse, hence we cannot make the restrictive assumptions on the structure of data (as relations and tuples), and the kinds of data-processing (relational queries). On the other hand

our empirical analysis on workflows and the Motif categorisation has shown that certain types of computation (data adapters) are not entirely arbitrary, and their operation can be made explicit in a rough-cut manner through semantics annotations specifying, from which inputs to which outputs value-copying occurs. We will discuss the implications of the grey-box in the next section.

## 8. Discussion

Revisiting the case-study revealed that the combination of liberal label propagation with broken factorial design can lead to inaccurate labels, which shows that this (anti)pattern not only degrades the standalone use of provenance traces but also degrades the operation of provenance enhancement applications, as our annotation approach.

Our propagation of labels is liberal as it is a combination of the following behaviours:

- We act on partial information (grey-box transparency denoting some value-copying occurring between inputs and outputs of an activity). When an activity invocation consumes a collection of items with distinct labels, and produces another collection of items, grey-box transparency does not allow us to accurately propagate labels item-wise. So instead we first *GENERALISE* labels to the top level input collection and *PROPAGATE* them to the top level output collection.
- To further expand the reach of labels we *DISTRIBUTE* labels at the top level collection to each item.

Other provenance annotation approaches do not have the inaccuracy issue either because:

- they do not support propagation of labels as in SPADE [48] or De Oliveira's [49] approach,
- they do not support a fine-grained provenance capability where annotations from distinct fine grained sources need to be managed as in the Galaxy workflow system's metadata propagation feature [7].
- or they require the user to not only supply the initial annotations but also the rules of propagation per workflow activity as in the Wings workflow system [8].

Rather than having these restrictions, a promising solution could be integrating workflow analysis with labelling. The workflow analysis rules can deduce whether lineage traces from multiple sources will be joined up at an activity invocation or not. We anticipate that by superimposing the label generation and propagation capabilities of activities as additional rules, the tool can also inform us whether labels from disparate sources will be joined up or not. This approach would not solve the inaccurate label propagation problem but would provide source points of potentially inaccurate labels. These could be provided as feedback to the user, prompting her to refactor the design configurations of her workflow. We plan to investigate these in our future work. A buggy label minting function could lead to inaccurate labels, *LabelFlow* does not provide a remedy for labels that are minted erroneously in the first place. We believe that this risk can be minimised by developing external labelling functions against standard data formats and pooling such functions in a library.

The cost involved in adapting our system is the manual annotation of workflow activities with labelling specifications and developing labelling functions for the focal data generation points in workflows. These are one-time costs. Both labelling specs and labelling functions can be reusable as tasks in workflows are underpinned by common components from local libraries as in Kepler [6] Vistrails [9] and Galaxy [7] workflow systems, or from both local and remote catalogues [55] as in Taverna [29]. A function that is capable of extracting labels from the VO Table of Sesame DB Inquiry could be reused in any workflow involves Sesame DB querying. Similarly the labelling spec generated for one processor is re-usable for all occurrences of that processor in workflows.

Earlier we mentioned that we designed *MINT* and *PROPAGATE* operators with re-usability in mind. Given that our operators decorate standard PROV traces, they have the potential to be used for labelling traces of workflow systems other than Taverna. Assessing the re-usability of our operators remains part of our future work.

As we saw in related work, any attempt at automating the generation of dynamic metadata has to assume the existence of a metadata extraction capability, i.e., labelling functions. The cost



of developing such functions can be minimised by exploiting existing libraries in parsing and transforming standardised scientific data formats. As guidance we provide the methodology that we followed in creating domain-specific labelling functions for the case-study workflow:

- Start by identifying a metadata profile that is applicable to your domain. Profiles are lightweight metadata schemas, typically comprised of a set of attributes. Profiles find increasing use in the context of data publishing. In our case-study this was the Astronomy Visualisation Metadata scheme available at the UK Digital Curation Center portal [3].
- Check whether there are existing tools/extractors that can produce metadata conforming to this profile. In our case a tool did not exist. However there were several VOTable parsers, or plain XML parsers we could utilise, we the latter.
- Develop a label minting function that either wraps an existing tool or is built afresh, which returns labels, whose definition correspond to attributes from the applicable metadata profile.
- For a workflow or group of workflows identify provenance querying hooks, these are input parameters that can be permuted at run time, or processor configurations determined during workflow design. Create a labelling vector that is comprised of label definitions that would carry this information. In our case this was simply a subset of the AVM metadata profile.

We think the most suitable user group to build labelling functions are developers who build the library of analytical processors for workflow systems. These re-usable processors are called Components in Taverna, Tools in Galaxy and Modules in Vistrails. Within Taverna a Component is an analytical or data adaptation step wrapped into a sub-workflow. If the component that a developer has built performs an analytical or data retrieval task, then the developer can also build the associated labelling function. If the component is instead a data adapter, then the developer simply needs to specify the data copying relationship from the adapters inputs and outputs, which will be saved as a Labelling Specification. Then, there will be users who run workflows built out of components from the library. These users will be posing their provenance queries with abstractions/hooks typical in a workflow provenance setting (input parameters, activity configurations). To pose a query over labels, the user would need simply select the corresponding label definitions for the hooks.

With labels we have adopted a very simplistic model to represent metadata. This can be viewed as a middle-ground between having no explicit metadata and having fully-fledged ontology-based representations that conceptually describe provenance artefacts [56], and interlink them with entities in the Linked Open Data (LOD) cloud [57]. In our case we are attempting at annotation at a very fine grain, we have therefore opted for a simple representation.

## 9. Conclusions

In this paper we described an architecture where (1) we use workflows and provenance traces associated with annotation behaviour as a roadmap to collect and propagate domain specific metadata and (2) we use data values as the source of domain specific metadata in the form of labels. We described two core operators, which operate at the granularity of workflow tasks and either create labels or propagating labels over a provenance graph depending on the function undertaken by the workflow task. We further described two operators that operate at the granularity of data artefacts shared among tasks (produced by one, consumed by the other). These operators propagate labels along the Collection-Item structure of data.

We assessed the utility of *LabelFlow* architecture and labels with a case-study. We observed that labels allow us to fully implement reporting queries, which in the absence of labels were only partially implemented. On the other hand we observed that correct implementation of iteration is crucial in order for both raw and labelled provenance to be useful.

**Acknowledgments:** Authors would like to thank the members of the e-Science Lab at the University of Manchester for their with using Taverna workflow system's APIs.

**Author Contributions:** This paper describes parts of the PhD dissertation research of Pinar Alper performed under the supervision of Carole A. Goble at the University of Manchester. Pinar Alper performed the research and wrote

the paper. Carole A. Goble supervised the research and reviewed all drafts of the paper. Khalid Belhajjame also supervised the research, edited Introduction and Case Study and Related Work sections. Vasa Curcin reviewed the initial drafts of the paper and guided Pinar Alper on suitable formal representation for *LabelFlow*.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Appendix A.

### Appendix A.1. Auxiliary Methods

The methods utilised by labelling operators to access and update the PROV compliant provenance space and the data and label spaces is given in Table A1.

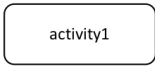





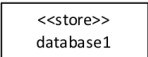


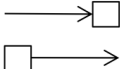
**Table A1.** Auxiliary methods used for annotating PROV compliant traces.

|   |
|---|
| <code>getInvocations(processorId:String, provStore:String):String[0..n]</code><br>Obtains identifiers of all the <i>PROV : activity</i> nodes in the trace that are documented to have occurred using <i>processorId</i> as a <i>PROV : plan</i> .  |
| <code>getAllGeneratedOutputs(processorId:String, port:String, provStore:String):String[0..n]</code><br>Obtains identifiers of all the <i>PROV : entity</i> (data) nodes in the trace that have been in a qualified <i>PROV : generation</i> relationship with some activity, where the activity has occurred according to a <i>PROV : plan</i> of identifier <i>processorId</i> and the generated data had role ( <i>PROV : hadRole</i> ) <i>port</i> . |
| <code>getAllActivityData(activityId:String, provStore:String) :String[0..n]</code><br>Obtains identifiers all the <i>PROV : entity</i> (data) nodes in the trace that have been in a <i>PROV : usage</i> or a <i>PROV : generation</i> relationship with the designated <i>activityId</i> .   |
| <code>getActivityOutData(activityId:String, port:String, provStore:String):String[0..n]</code><br>Obtains identifiers of the <i>PROV : entity</i> (data) nodes in the trace that are in a <i>PROV : generation</i> relation with the designated <i>activityId</i> , where the generation is qualified stating that the data node played the role ( <i>PROV : hadRole</i> ) identified with <i>port</i> .  |
| <code>bindLabelsToData(dataId:String, labels:LabelInstance[0..n]) :LabelInstance[0..n]</code><br>Returns a copy of the labels, where the target of each copy is set to the designated data record.  |
| <code>clone(labels:LabelInstance[0..n]):LabelInstance[0..n]</code><br>Creates a copy of all the labels in the input set .   |
| <code>submitLabels(labels:LabelInstance[0..n], labelStore:String)</code><br>Stores all the label instances in the designated label space.   |
| <code>getItems(coll:String, depthDifference:Integer, provStore:String):String[0..n]</code><br>Obtains identifiers of <i>PROV : entity</i> (data) nodes in the trace that are contained ( <i>PROV : hadMember</i> ) by the designated <i>PROV : Collection coll</i> at <i>depthDifference</i> level deep.  |
| <code>getLabels(item, labelDefinitions, labelStore:String) :LabelInstance[0..n]</code><br>Obtains all the labels, whose target is the designated item.  |
| <code>getEnclosingCollections(items:String[0..n], depthDifference:Integer, provStore:String):String[0..n]</code><br>Obtains the identifiers of <i>PROV : Collection</i> nodes in the trace, which at <i>depthDifference</i> level deep contain ( <i>PROV : hadMember</i> ) the designated items.  |

### Appendix A.2. UML Activity Diagram Syntax

Table A2 provides a subset of elements from this notation and their definitions as per UML reference model.

**Table A2.** UML Activity Diagram elements notation and definitions.

|   |   |   |  |   |
|---|---|---|--|---|
|  |  |  |  |  |
| <b>Action/Activity node</b>   | <b>Start Node</b>   | <b>End Node</b>   | <b>Fork/Join</b>   | <b>Control Flow</b>   |
|  |  |  |  |  |
| <b>Object Node</b>  | <b>Datastore Node</b>   | <b>Input Pin</b>  | <b>Value Pin</b>   | <b>Object Flow</b>  |

An activity diagram is a graph of nodes denoting a process comprised of steps of computation and flows of (primarily) control (and optionally) data among steps.

An action/activity node (rounded rectangle) denotes a computational step. An action is an atomic step which is not further broken into sub-steps, whereas an activity is a group of actions or sub-activities.

Start node (solid circle) is a control node at which flow starts when an activity is invoked.

End node (hollow circle with solid circle inside) is a control node that stops all flows in an activity.

Fork node (thick line segment) is a control node that has one incoming edge and multiple outgoing edges and is used to split incoming flow into multiple concurrent flows. Join node is a control node that has multiple incoming edges and one outgoing edge and is used to synchronise incoming concurrent flows.

Control flow edge (arrow) is an edge denoting flow of control from one activity to another.

Object node (rectangle) is an edge denoting flow of data from one activity to another.

A data store nodes (rectangle) are stereotyped object nodes, which denote non-transient data that is persisted during the computational process.

Object flow edge (arrow) is an edge denoting flow of data during a computational process. An object flow edge is one that connects two nodes, where at least one is an object node. A value pin is special kind of input pin defined to provide constant values as input.

Pins (small rectangle at edge of rounded rectangle) are object nodes used to denote inputs/outputs to activities. A value pin is a special kind of input pin, which denotes constant-valued inputs to an activity.

## References

- Hey, T.; Tansley, S.; Tolle, K.M. (Eds.) *The Fourth Paradigm: Data-Intensive Scientific Discovery*; Microsoft Research: New York, NY, USA, 2009.
- Scientific Data, Open-Access Journal*; Nature Publishing Group: London, UK, 2015. Available online: <http://www.nature.com/sdata/> (accessed on 22 February 2018).
- Davenhall, C. *Curation Reference Manual, Chapter on Scientific Metadata*; The Digital Curation Centre (DCC): Edinburgh, UK, 2011. Available online: <http://www.dcc.ac.uk/resources/curation-reference-manual> (accessed on 22 February 2018).
- Taylor, C.F.; Field, D.; Sansone, S.A.; Aerts, J.; Apweiler, R.; Ashburner, M.; Ball, C.A.; Binz, P.; Bogue, M.; Booth, T.; et al. Promoting coherent minimum reporting guidelines for biological and biomedical investigations: The MIBBI project. *Nat. Biotechnol.* **2008**, *26*, 889–896.
- Sansone, S.A.; Rocca-Serra, P.; Field, D.; Maguire, E.; Taylor, C.; Hofmann, O.; Fang, H.; Neumann, S.; Tong, W.; Amaral-Zettler, L.; et al. Toward interoperable bioscience data. *Nat. Genet.* **2012**, *44*, 121–126.
- Ludaescher, B.; Altintas, I.; Berkley, C.; Higgins, D.; Altintas, I.; Berkley, C.; Higgins, D.; Jaeger, E.; Jones, M.; Lee, E.A.; et al. Scientific workflow management and the Kepler system. *Concurr. Comput. Pract. Exp.* **2006**, *18*, 1039–1065.
- Giardine, B.; Riemer, C.; Hardison, R.C.; Burhans, R.; Shah, P.; Elnitski, L.; Zhang, Y.; Blankenberg, D.; Albert, I.; Taylor, J.; et al. Galaxy: A platform for interactive large-scale genome analysis. *Genome Res.* **2005**, *15*, 1451–1455.
- Gil, Y.; Ratnakar, V.; Kim, J.; González-Calero, P.A.; Groth, P.; Moody, J.; Deelman, E. Wings: Intelligent Workflow-Based Design of Computational Experiments. *IEEE Intell. Syst.* **2011**, *26*, 62–72.

9. Callahan, S.P.; Freire, J.; Santos, E.; Scheidegger, C.E.; Silva, C.T.; Vo, H.T. Vistrails: Visualization meets data management. In *ACM SIGMOD*; ACM Press: New York, NY, USA, 2006; pp. 745–747.
10. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2014. Available online: <https://www.r-project.org> (accessed on 22 February 2018).
11. Rossum, G. *Python Reference Manual*; Technical Report; CWI (Centre for Mathematics and Computer Science): Amsterdam, The Netherlands, 1995.
12. Missier, P.; Paton, N.W.; Belhajjame, K. Fine-grained and Efficient Lineage Querying of Collection-based Workflow Provenance. In *Proceedings of the 13th International Conference on Extending Database Technology*, Lausanne, Switzerland, 22–26 March 2010; ACM: New York, NY, USA, 2010; pp. 299–310.
13. Chapman, A.; Jagadish, H.V. Understanding provenance black boxes. *Distrib. Parallel Databases* **2010**, *27*, 139–167.
14. Tenopir, C.; Allard, S.; Douglass, K.; Aydinoglu, A.U.; Wu, L.; Read, E.; Manoff, M.; Frame, M. Data Sharing by Scientists: Practices and Perceptions. *PLoS ONE* **2011**, *6*, e21101.
15. Missier, P.; Sahoo, S.S.; Zhao, J.; Goble, C.; Sheth, A. *Janus*: From Workflows to Semantic Provenance and Linked Open Data. In *Proceedings of the 3rd International Provenance and Annotation Workshop (IPAW 2010)*, Troy, NY, USA, 15–16 June 2010; pp. 129–141.
16. Cao, B.; Plale, B.; Subramanian, G.; Missier, P.; Goble, C.A.; Simmhan, Y. Semantically Annotated Provenance in the Life Science Grid. In *Proceedings of the 1st International Workshop on the role of Semantic Web in Provenance Management (SWPM 2009)*, Washington DC, USA, 25 October 2009.
17. Ailamaki, A.; Kantere, V.; Dash, D. Managing Scientific Data. *Commun. ACM* **2010**, *53*, 68–78.
18. Belhajjame, K.; Zhao, J.; Garijo, D.; Garrido, A.; Soiland-Reyes, S.; Alper, P.; Corcho, O. A Workflow PROV-corpus Based on Taverna and Wings. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, Genoa, Italy, 18–22 March 2013; ACM: New York, NY, USA, 2013; pp. 331–332.
19. Hull, D.; Stevens, R.; Lord, P.; Wroe, C.; Goble, C. Treating shimantic web syndrome with ontologies. In *Proceedings of the 1st Advanced Knowledge Technologies Workshop on Semantic Web Services (AKT-SWS04)* KMi, Milton Keynes, UK, 8 December 2004.
20. Alagiannis, I.; Borovica, R.; Branco, M.; Idreos, S.; Ailamaki, A. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, Scottsdale, AZ, USA, 20–24 May 2012; ACM: New York, NY, USA, 2012; pp. 241–252.
21. Wu, K. FastBit: An efficient indexing technology for accelerating data-intensive science. *J. Phys. Conf. Ser.* **2005**, *16*, 556.
22. Alawini, A.; Maier, D.; Tufte, K.; Howe, B.; Nandikur, R. Towards Automated Prediction of Relationships Among Scientific Datasets. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, La Jolla, CA, USA, 29 June–1 July 2015; ACM: New York, NY, USA, 2015.
23. Sousa, V.S.; de Oliveira, D.; Mattoso, M. Exploratory Analysis of Raw Data Files through Dataflows. In *Proceedings of the 2014 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, Paris, France, 22–24 October 2014; pp. 114–119.
24. Garijo, D.; Alper, P.; Belhajjame, K.; Corcho, O.; Gil, Y.; Goble, C. Common motifs in scientific workflows: An empirical analysis. *Future Gener. Comput. Syst.* **2014**, *36*, 338–351.
25. Zhao, J.; Sahoo, S.S.; Missier, P.; Sheth, A.P.; Goble, C.A. Extending Semantic Provenance into the Web of Data. *IEEE Internet Comput.* **2011**, *15*, 40–48.
26. Alper, P.; Goble, C.A.; Belhajjame, K. On assisting scientific data curation in collection-based dataflows using labels. In *Proceedings of the 8th Workshop On Workflows in Support of Large-Scale Science, (WORKS)*, Denver, CO, USA, 17 November 2013; pp. 7–16.
27. Alper, P.; Belhajjame, K.; Goble, C.A.; Karagoz, P. *LabelFlow*: Exploiting Workflow Provenance to Surface Scientific Data Provenance. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)*, Cologne, Germany, 9–13 June 2014; pp. 84–96.
28. Exposito, S.S. *Workflow: Calculating the Internal Extinction with Data from Leda*; myExperiment Repository; 2012. Available online: <http://www.myexperiment.org/workflows/2920/versions/2.html> (accessed on 22 February 2018).
29. Missier, P.; Soiland-Reyes, S.; Owen, S.; Tan, W.; Nenadic, A.; Dunlop, I.; Williams, A.; Oinn, T.; Goble, C. Taverna, Reloaded. In *Proceedings of Scientific and Statistical Database Management Conference (SSDBM)*, Lecture

- Notes in Computer Science, Heidelberg, Germany, 30 June–2 July 2010*; Gertz, M., Ludäscher, B., Eds.; Springer: Berlin, Germany, 2010; Volume 6187, pp. 471–481.
30. Moreau, L.; Ludäscher, B.; Altintas, I.; Barga, R.S.; Bowers, S.; Callahan, S.; Chin, G., Jr.; Clifford, B.; Cohen, S.; Cohen-Boulakia, S.; et al. The First Provenance Challenge. *CCPE* **2008**, *20*, 409–418.
  31. Alper, P.; Belhajjame, K.; Goble, C.A. Static analysis of Taverna workflows to predict provenance patterns. *Future Gener. Comput. Syst.* **2017**, *75*, 310–329.
  32. Belhajjame, K.; Zhao, J.; Garijo, D.; Gamble, M.; Hettne, K.; Palma, R.; Mina, E.; Corcho, O.; Gómez-Pérez, J.M.; Bechhofer, S.; et al. Using a suite of ontologies for preserving workflow-centric research objects. *Web Semant. Sci. Serv. Agents World Wide Web* **2015**, *32*, 16–42.
  33. Wood, D.; Lanthaler, M.; Cyganiak, R. *RDF 1.1 Concepts and Abstract Syntax*; W3C Recommendation; 2014. Available online: <https://www.w3.org/TR/rdf11-concepts/> (accessed on 22 February 2018).
  34. Groth, P.; Editors, L.M. *PROV-Overview: An Overview of the PROV Family of Documents*; W3C; 2013. Available online: <http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/> (accessed on 22 February 2018).
  35. Missier, P.; Dey, S.; Belhajjame, K.; Cuevas-Vicentín, V.; Ludäscher, B. D-PROV: Extending the PROV provenance model with workflow structure. In Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance, Lombard, IL, USA, 2–3 April 2013.
  36. Brandizi, M.; Melnichuk, O.; Bild, R.; Kohlmayer, F.; Rodriguez-Castro, B.; Spengler, H.; Kuhn, K.A.; Kuchinke, W.; Ohmann, C.; Mustonen, T.; et al. Orchestrating differential data access for translational research: A pilot implementation. *BMC Med. Inf. Decis. Mak.* **2017**, *17*, 30.
  37. Diaz, G.; Arenas, M.; Benedikt, M. SPARQLByE: Querying RDF Data by Example. *Proc. VLDB Endow.* **2016**, *9*, 1533–1536.
  38. Garijo, D.; Alper, P.; Belhajjame, K. *The Workflow Motif Ontology*; UPM Ontology Engineering Group, Revision 1.02; 2013. Available online: <http://vocab.linkeddata.es/motifs/> (accessed on 22 February 2018).
  39. Booch, G.; Rumbaugh, J.; Jacobson, I. *Unified Modeling Language User Guide*, 2nd ed.; Addison-Wesley Object Technology Series; Addison-Wesley Professional: Boston, MA, USA, 2005.
  40. Alper, P. *LabelFlow Evaluation Datasets*. 2015. Available online: <https://github.com/pinarpink/phd-sources/tree/master/labeling-workflow-generator> (accessed on 22 February 2018).
  41. Belhajjame, K.; Cheney, J.; Corsar, D.; Garijo, D.; Soiland-Reyes, S.; Zednik, S.; Zhao, J. *PROV-O: The PROV Ontology*; W3C; 2012. Available online: <http://www.w3.org/TR/prov-o/> (accessed on 22 February 2018).
  42. Group, P.W. *PROV Implementation Report*. 2013. Available online: <https://www.w3.org/TR/prov-implementations/> (accessed on 22 February 2018).
  43. Carroll, J.J.; Dickinson, I.; Dollin, C.; Reynolds, D.; Seaborne, A.; Wilkinson, K. Jena: Implementing the Semantic Web Recommendations. In Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Amp, New York, NY, USA, 17–20 May 2004; Posters; ACM: New York, NY, USA, 2004; pp. 74–83.
  44. Moreau, L.; Huynh, T.D.; Michaelides, D. An Online Validator for Provenance: Algorithmic Design, Testing, and API. In *Fundamental Approaches to Software Engineering*; Gnesi, S., Rensink, A., Eds.; Springer: Berlin, Germany, 2014; pp. 291–305.
  45. Missier, P.; Sahoo, S.S.; Zhao, J.; Goble, C.; Sheth, A. *Janus: From Workflows to Semantic Provenance and Linked Open Data*. In *Provenance and Annotation of Data and Processes*; Springer: Berlin, Germany, 2010; Volume 6378, pp. 129–141.
  46. Zhao, J.; Wroe, C.; Goble, C.; Stevens, R.; Quan, D.; Greenwood, M. Using Semantic Web Technologies for Representing e-Science Provenance. In Proceedings of the ISWC 2004, Hiroshima, Japan, 7–11 November 2004; Springer: Berlin, Germany, 2004; Volume 3298, pp. 92–106.
  47. Altschul, S.F.; Gish, W.; Miller, W.; Myers, E.W.; Lipman, D.J. Basic local alignment search tool. *J. Mol. Biol.* **1990**, *215*, 403–410.
  48. Sahoo, S.S.; Sheth, A.; Henson, C. Semantic provenance for escience: Managing the deluge of scientific data. *IEEE Internet Comput.* **2008**, *12*, 46–54.
  49. De Oliveira, D.; Silva, V.; Mattoso, M. How Much Domain Data Should Be in Provenance Databases? In Proceedings of the 7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15), Edinburgh, UK, 8–9 July 2015; USENIX Association: Edinburgh, UK, 2015.

50. Halper, M.; Geller, J.; Perl, Y. Value Propagation in Object-oriented Database Part Hierarchies. In Proceedings of the Second International Conference on Information and Knowledge Management, ACM, CIKM'93, Washington, DC, USA, 1–5 November 1993; pp. 606–614.
51. Artale, A.; Franconi, E.; Guarino, N.; Pazzi, L. Part-whole Relations in Object-centered Systems: An Overview. *Data Knowl. Eng.* **1996**, *20*, 347–383.
52. Greenberg, J. Theoretical Considerations of Lifecycle Modelling: An Analysis of the Dryad Repository Demonstrating Automatic Metadata Propagation, Inheritance, and Value System Adoption. *Cat. Classif. Q.* **2009**, *47*, 380–402.
53. Bhagwat, D.; Chiticariu, L.; Tan, W.-C.; Vijayvargiya, G. An Annotation Management System for Relational Databases. In Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, ON, Canada, 31 August–3 September 2004; Nascimento, M.A., Ozsu, M.T., Nascimento, M.A., Özsu, M.T., Kossmann, D., Miller, R.J., Blakeley, J.A., Schiefe, B., Eds.; VLDB Endowment Inc.: San Fransisco, CA, USA, 2004; pp. 900–911.
54. Bowers, S.; Ludäscher, B. A Calculus for Propagating Semantic Annotations Through Scientific Workflow Queries. In Proceedings of the 2006 International Conference on Current Trends in Database Technology, Munich, Germany, 26–31 March 2006; Springer: Berlin, Germany, 2006; pp. 712–723.
55. Bhagat, J.; Tanoh, F.; Nzuobontane, E.; Laurent, T.; Orlowski, J.; Roos, M.; Wolstencroft, K.; Aleksejevs, S.; Stevens, R.; Pettifer, S.; et al. BioCatalogue: A universal catalogue of web services for the life sciences. *Nucleic Acids Res.* **2010**, *38*, 689–694.
56. Hitzler, P.; Krötzsch, M.; Parsia, B.; Rudolph, S. (Eds.) *OWL 2 Web Ontology Language: Primer*; W3C Recommendation; 27 October 2009. Available online: <http://www.w3.org/TR/owl2-primer/> (accessed on 22 February 2018).
57. Bechhofer, S.; Buchan, I.; De Roure, D.; Missier, P.; Ainsworth, J.; Bhagat, J.; Couch, P.; Cruickshank, D.; Delderfield, M.; Dunlop, I.; et al. Why linked data is not enough for scientists. Special section: Recent advances in e-Science. *Future Gener. Comput. Syst.* **2013**, *29*, 599–611.



© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).