The University of Manchester

The University of Manchester Research

Static Analysis of Taverna Workflows To Predict Provenance Patterns

DOI:
10.1016/j.future.2017.01.004

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Alper, P., Belhajjame, K., & Goble, C. (2017). Static Analysis of Taverna Workflows To Predict Provenance Patterns. *Future Generation Computer Systems*, *75*, 310-329. https://doi.org/10.1016/j.future.2017.01.004

Published in:
Future Generation Computer Systems

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

OPEN ACCESS

Download date:27. Apr. 2024

Static Analysis of Taverna Workflows To Predict Provenance Patterns

Pinar Alper[a], Khalid Belhajjame[b], Carole A. Goble[a]

[a]*University of Manchester, School of Computer Science, Oxford Rd, Manchester M13 9PL, United Kingdom*
[b]*Université Paris Dauphine, Place du Maréchal de Lattre de Tassigny, 75016 Paris, France*

Abstract

Workflows have found adoption in scientific domains particularly due to their automation and provenance features. Using workflows scientists can repeat analyses with different input parameters to later use provenance to access and compare results based on respective parameters. An assumption that is often made is that by designing an analysis as a workflow, we get parameter-to-result traceability for free with workflow provenance. This assumption holds for cases of coarse-grained traceability, where an entire workflow is subjected to repetition and all workflow parameters contribute to all results. On the other hand for cases requiring finer grained traceability, where a workflow is configured with collections of parameters and analyses within a workflow are repeated with combinations of parameters from collections, this assumption is not guaranteed to hold. In this paper we identify two dimensions that affect fine-grained traceability as: 1) the level of granularity supported by a workflow system in modelling parameters/data in workflows and in provenance, which we name as the level of support for *Factorial Design*, and 2) the practice of scientists in successfully encoding *Factorial Design* into workflows. Taverna is a workflow system that provides extensive features for factorial design, meanwhile it provides an uncontrolled approach to workflow design; meaning scientists may create workflows, which, when run, could break traceability in provenance. Using a real-world Taverna workflow we show how broken traceability manifests in provenance and how it can render provenance practically useless for accessing workflow outputs derived from particular input parameters. In order to prevent broken traceability from occurring, we describe a rule-based static analysis technique, which operates over workflow descriptions and anticipates patterns in provenance. Our rules exploit the well-defined execution behaviour in the Taverna system. In order to understand *Factorial Design* support in workflow systems in general, we provide a comparative survey. We conclude that other workflow systems also provide constructs for *Factorial Design*, and, similar to Taverna, they too are prone to broken traceability.

Keywords: scientific workflows; provenance; annotation; static analysis

1. Introduction

Workflows systems have enjoyed notable adoption in many scientific disciplines as a means for encoding computational analyses, offering three key benefits:

- They ease the exploitation of diverse scientific resources by providing a readily available resource-access infrastructure while still honouring the original codes and their native hosting environment. Workflow systems are often utilised as front-ends to access web services, command-line tools or community databases, which are often third party and are **black-box** analysis capabilities.
- They bring transparency into analyses by capturing them as explicit processes and monitoring and documenting process executions as workflow provenance [1]. Provenance is a trail of activity executions, their input/output, and the causal relations among activities and data. Such trails vary in their **granularity** - that is the detail that the processes and data are documented; for example, file level, record in file level or item in record level. To date such trails have been used chiefly in debugging workflows, verifying analytical processes or as proof of execution to support veracity of results for scientific audit and peer-review.
- They enable automation and repeatability by encoding analyses as executable processes. In particular workflows are a powerful means for exploring (also known as "sweeping") a parameter space for an analysis. Scientists (re)run activities while carefully permuting **key factors** that vary the workflow set-up, such as input datasets and/or parameters, expecting to conduct a meta-analysis by comparing and contrasting the results of each activity. *It is this class of workflows that concerns this paper.*

Adoption of workflows has proliferated computational analyses and data obtained through such analyses. Consequently, a need for result management has emerged [2] . Scientists require result data to be indexed with respective analysis parameters, configurations or input data, so that a number of post-execution activities such as result access, comparison, meta-analysis and reporting can occur. A recent survey [2] shows that although scientists receive ample tool support for the design and enact-

Email addresses: alperp@cs.manchester.ac.uk (Pinar Alper), Khalid.Belhajjame@dauphine.fr (Khalid Belhajjame), carole.goble@manchester.ac.uk (Carole A. Goble)

ment of analyses, they receive less support for post-execution activities. Hence, established but ad-hoc techniques of result management prevail; such as naming output files with parameter values, embedding parameters into data files or using folders to hold results from the sweep of a particular parameter.

Given the automation and execution trailing features of workflows, it is often assumed that by simply designing an analysis as a workflow, the **parameter-to-result traceability** comes for free with provenance. In a simplistic coarse-grained viewpoint, where a workflow's entire outputs dependent on its entire inputs, the above expectation can be met. However, the following features of scientific workflow systems complicate the picture:

- Workflow languages allow the modelling of parameter collections, where each collection represents the changeable factors of a scientific analysis and the various possible combination of parameters. Workflow languages also allow the modelling analysis repetitions over input parameter combinations. In this paper we refer to these capabilities as a workflow system's constructs for **Factorial Design (FD)**
- Workflow systems collect provenance via add-on components, which can lead to a discrepancy of the granularities with which parameters/data are represented in workflows and in provenance. For example, in Kepler [3] an array structure is a single one blob in provenance, and in Vistrails [4] activities can be repeated, but it appears as if the repetitions consume one piece of data, while in fact they consume individual elements of an array.
- In order to trace diverse scientific computations, workflow systems make no assumptions on the inner workings of activities and simply record activities as black-boxes with inputs and outputs in provenance. As a consequence, once a black-box activity is recorded provenance lends itself minimally to inference of more refined or finer-grained causal relations among that activitys inputs and outputs.

As a result, it becomes non-trivial to assess whether for a particular workflow, with multiple input factors and blackbox steps, its execution provenance will yield parameter-to-result traceability.

In this paper we observe that such traceability is in fact not a guaranteed side-effect of having workflows, specifically for those executing factor and data sweeps. The way workflows are designed are key in shaping their provenance and therefore the starting point to achieve traceability is to ensure that the workflow design process is informed with anticipated provenance characteristics. We also observe that the workflow system itself, such as Taverna [5], can be exploited to support a static analysis to identify pre-hoc potential concerns in parameter-to-result traceability.

In this paper we make the following contributions:

- Using a real-world Taverna workflow we illustrate Factorial Design (that is, workflows designed to examine specific factors) and show how this can break. We show how a particular configuration in the workflow design leads to the *n-by-m* pattern in provenance, where all *n* input parameters of the workflow are traceable to all *m* results.
- We show how Taverna's well-defined behaviour in preparing parameter combinations and task repetition allows us to anticipate patterns in provenance without running workflows. Based on a formal representation of Taverna workflows and execution behaviour we provide a set of rules that are predicated on the elements of a workflow to predict the provenance characteristics of that workflow.
- We provide a survey of scientific workflow systems to understand their level of support for Factorial Design and whether provenance from these systems could potentially lead to have the *n-by-m* pattern.

Parameter-to-result traceability is a prerequisite for building tools that can exploit workflow provenance for result management. The value proposition of our approach is that for a prevalent class of parameter-sweep type workflows, we provide a technique that targets this traceability requirement.

The paper is organised as follows. We begin in Section 2 by introducing an example Taverna workflow and its provenance. Using the same example, in Section 3 we present a high level characterisation factorial design and how it can be broken, we also show the negative impact of broken factorial design on provenance queries. In Section 4 we provide an abstract model for static analysis of Taverna workflows, and in Section 5 we discuss the model's implementation in Datalog. We review related work on workflow and programme analysis in Section 7. In Section 6 we present the comparative survey of workflow systems. We conclude and outline future work in Sections 8 and 9 respectively.

2. Taverna Workflows and Provenance

In this section we illustrate some of the constructs enabling FD in the Taverna workflow system. We also observe the granularity characteristics of Taverna provenance.

Taverna workflows are comprised of *tasks*, *input/output ports* of tasks, and the *dataflow* dependencies among ports. Figure 1 displays a workflow [6] that has been developed by the AMIGA research group [7] that studies the interstellar medium of galaxies to understand its impact on galaxy shapes (morphologies). The workflow has been developed as part of the Wf4Ever project [8], which has devised tools to support scientific workflow preservation. Our example workflow accepts as input a set of galaxy names (*list_ cig_ name*), and generates extinction values of galaxies as output (*data_ internal_ extinction*). The workflow contains tasks for data retrieval from the remote Sesame and Leda data repositories (Steps 1& 2), calculation of galaxy properties with local tools (Step 3), and data adaptation (unnumbered steps in between). In addition to the core constructs, Taverna supports the following:

- Simple and Collection data types for ports (of tasks and workflows). Collection types allow the modelling of (nested) data/parameter collections. In our example the *list_cig_name* workflow input is defined to be a *collection* of *string* values.

- Iteration (or looping) constructs to model task repetition. Tasks in Taverna are accompanied with *iteration strategies* that prescribe how to perform repetition in case the task encounters a collection of inputs (to be swept over) rather than a single input. The *Sesame* task (Step-1) has a single input port, hence it will simply be repeated per item for an incoming input collection. In case a task has multiple inputs as illustrated with *Leda* task (Step-2) in Figure 1, which has two inputs, then the task's iteration strategy outlines how to pick input combinations from incoming collections. Iteration strategies, which we formally describe in Section 4, are recipes based on *Dot* and *Cross* product operations over collections. Briefly *Cross* product creates input combinations through a cartesian pairing of items in input lists, whereas *Dot* product pairs items from equal sized lists on a one-to-one basis.

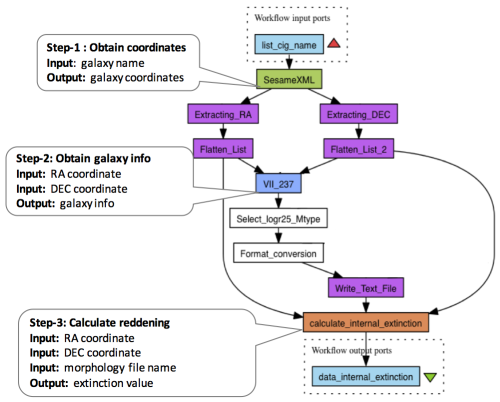


Figure 1: Sample workflow from Astronomy developed by the Wf4Ever project.

In Figure 2 we illustrate a particular execution of the workflow of Figure 1 with two input galaxies. During execution some tasks are exposed to collections rather than single inputs of defined type, therefore they are iterated as per respective strategies. The iteration begins at the *Sesame* task, which is invoked two times, once for each galaxy. The iterated execution of *Sesame* results in a list of xml-based coordinate information regarding each galaxies. The iteration of *Sesame* creates a ripple effect and causes the follow-on adapter tasks *Extracting_RA*, and *Extracting_DEC* to also iterate. By definition these tasks extract lists of coordinate fragments from input XML data. As extractors iterate, they result in a nested list, rather than a single list of coordinate fragments. The workflow contains two flattening tasks that by definition coalesce a given nested list (of strings) into a single-depth list. The scientist has included these flatteners to bundle coordinate fragments for all galaxies into single lists of coordinate fragments. Note that the flattener tasks do not iterate as they encounter a single input that is of their expected input type (a nested list). The *Leda* task iterates and at each iteration consumes same indexed coordinate fragments from the two incoming lists. Galaxy information returned from the *Leda* task is then subjected to filtering to be used in the follow-on steps of the workflow.

The information illustrated in Figure 2 makes up the core

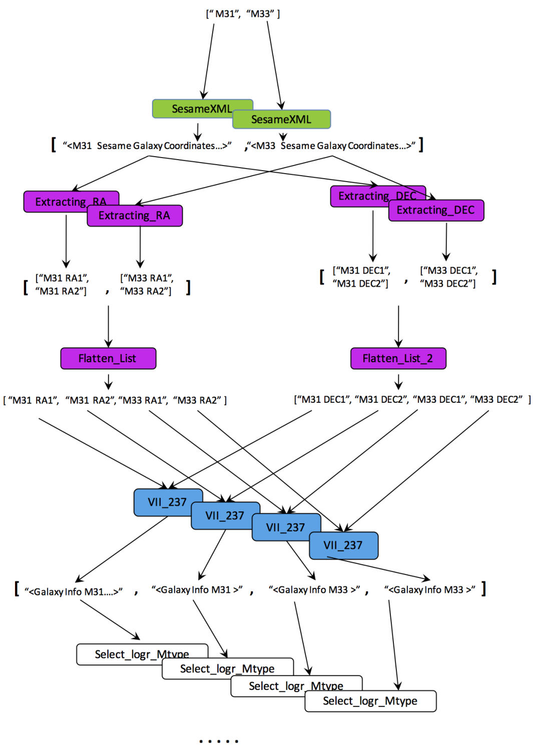

[" M31", "M33"]
SesameXML
SesameXML
["<M31 Sesame Galaxy Coordinates...>" ,"<M33 Sesame Galaxy Coordinates...>"]
Extracting_RA
Extracting_RA
Extracting_DEC
Extracting_DEC
[["M31 RA1", "M31 RA2"] , ["M33 RA1", "M33 RA2"]]
[["M31 DEC1", "M31 DEC2"] , ["M33 DEC1", "M33 DEC2"]]
Flatten_List
Flatten_List_2
["M31 RA1", "M31 RA2", "M33 RA1", "M33 RA2"]
["M31 DEC1", "M31 DEC2", "M33 DEC1", "M33 DEC2"]
VII_237
VII_237
VII_237
VII_237
["<Galaxy Info M31....>" , "<Galaxy Info M31 >" , "<Galaxy Info M33 >" , "<Galaxy Info M33 >"]
Select_logr_Mtype
Select_logr_Mtype
Select_logr_Mtype
Select_logr_Mtype
.....

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

of workflow provenance. Workflow systems utilise standard provenance vocabularies such as OPM [9] and PROV [10] to represent execution trails as graph structured information comprised of nodes denoting *activities* (task invocations) and *data*, and causal links among nodes denoting an activity's *consumption* of input and *generation* of output data and the *lineage* among an activity's inputs and outputs. In Taverna, workflows and provenance are in consistent granularities. Meaning that for a port defined as a collection type in the workflow, the data appearing at that port will be recorded as a collection in provenance. Each item in this collection will be recorded as a distinct artefact in provenance. Similarly given a task with an iteration strategy in the workflow, each invocation of task will be documented distinctly in provenance.

We shall note that recent research on provenance have introduced the notion of "provenance patterns/templates" [11] [12]. Templates denote fragments of provenance that are expected to be collected by a system, and they are commonly represented in standard models such as OPM or PROV. Combined with conformance checking mechanisms, templates can act as an explicit schema for provenance and can be used to guarantee certain patterns. *In our case we do not have a template for the provenance to be collected, however, we have peripheral information: workflow description and workflow execution semantics, which can be used to infer anticipated provenance patterns*. We will return to the discussion on provenance templates in the conclusions of this paper, where we discuss alternatives to our approach.

3. Factorial Design in Taverna

The support for factorial design in a workflow system is the ability to define an input parameter/data space for an analysis, and the ability to run the analysis over each point in that space. In our example in Figure 1 the input list of galaxy names (*list_ cig_ name*) defines a one dimensional parameter space, and Taverna's iteration capability allows performing the analyses involved in the workflow for each galaxy. Taverna supports process/data granularities that are consistent in both the workflow description and provenance. As a consequence factorial design as encoded in a Taverna workflow can be reflected to the workflow's execution provenance. This makes Taverna provenance a potential index to access result data by specifying input parameters used to generate that data. However, in order for this index to be effective, provenance should link-up parameters and results discretely. Let I denote a collection of input (parameter) nodes in a workflow provenance graph P, and let $descendants(i)$ be a function that returns the set of data nodes that descend from the given node. We state that *the trace P has factorial design for I if for each $i \in I$ $descendants(i)$ are mutually disjoint sets. We refer to the case otherwise as broken-factorial design, i.e. if for $i, j \in I$ the set $descendants(i) \cap descendants(j)$ is non-empty.*

The trace given in Figure 2 illustrates broken factorial design. Through iteration the workflow execution generates coordinates (with the *Sesame* activities), and coordinate fragments (with the *Extracting_RA*, *Extracting_DEC* activities) exclusively for each input galaxy. At the *Flatten_List_2* and *Flatten_ List*, activities however, factorial design breaks, as a single activity invocation bundles all coordinate fragments for all galaxies into a single list, which becomes a descendant of both input galaxy ids $M31$ and $M33$. By inspecting the follow-on structure of execution we understand that the workflow designer has inadvertently created this design mishap as the follow on *Leda* is iterated over these bundled up lists, so are all remaining steps of the workflow. So if the designer did not include the list flattening steps and iterated *Leda* directly over the outputs of extraction steps, mutual disjointness of galaxy id's would have been preserved.

Note that factorial design is defined in terms of a particular input collection. Therefore, for a workflow with multiple input collections, the provenance trace may have factorial design for some of the input collections, whereas have broken factorial design for others. This is illustrated with the trace in Figure 3. The workflow is comprised of a *Pair-up* task that performs string concatenation, followed by a *List_To_String* task that coalesces lists of strings into a single string. The workflow is ran with two input collections one a list of characters ("*A*", "*B*", and "*C*") and the other a list of input numbers ("1" and "2"). The iteration strategy for *Pair-Up* prescribes the the exploration of all possible input combinations. Consequently *Pair-Up* is invoked 6 times consuming input pairs created from a cartesian (cross) product of input lists. The output is a nested list, where the top level list contains 3 elements, each a list, containing items that commonly descend from the same indiced item in the list of characters (e.g. all items in list ["*A*-1 *A*-2"] descend from "*A*").

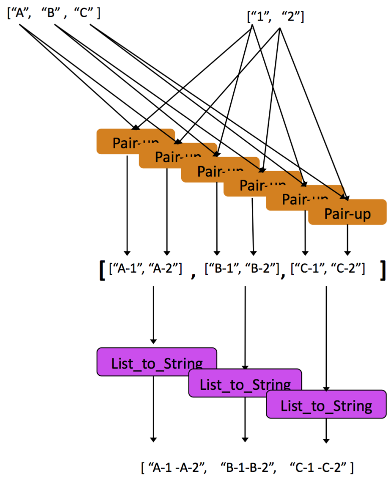


Figure 3: A trace that has broken factorial design for only one of its input collections (numbers).

Meanwhile items in those lists descend from different items in the list of numbers. The *List_To_String* task is invoked 3 times in each iteration consuming each one of three lists producing a single string representation of them. This trace has factorial design for the input list of characters, whereas it has broken factorial design for the input list of numbers, because each invocation of *List_To_String* consumes items that descend from distinct numbers.

Broken factorial design manifests in provenance with either the *n-by*-1 or *n-by-m* patterns denoting a single activity consuming n inputs (descending from n distinct parameters), and respectively producing a single output or a collection of m outputs. The trace in Figure 2 illustrates *n-by-m*, whereas the trace in Figure 3 illustrates *n-by*-1. What renders provenance an ineffective index for result access is the particular case of *n-by-m*, where the size of m is positively correlated with size of n as illustrated in the flattening activities in Figure 2, where the output list is based on the input list. To illustrate we ran a sample query to retrieve all outputs that belong to a particular galaxy ($M31$). We realise this query by identifying the galaxy name workflow input with designated value, and then obtaining all descendants of that input node in the provenance graph. Due to broken factorial design, all galaxy ids have common descendants beyond the flattening step. Hence, the precision of provenance-based result selection is reciprocal of the input size: $\frac{1}{n}$ (depicted in Figure 4).

Given the *black-box* nature of workflow provenance. Once broken factorial design occurs in provenance there is no actionable information to remedy it. Therefore one strategy, which we take in this paper, is to prevent this pattern from occurring in the first place through an analysis of workflows at design time. When designing workflows scientists may compromise factorial design:

- either intentionally, by integrating external analytical resources (databases, tools, web services) coarsely into workflows. Scientists may submit data from multiple parameters to an external resource all at once to reduce resource access overhead.

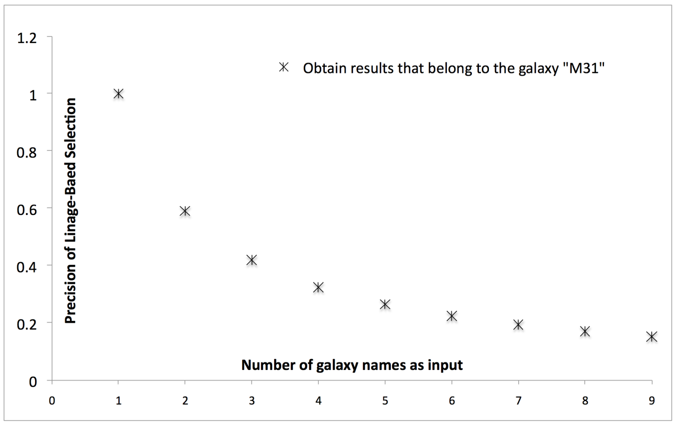


Figure 4: A trace that has broken factorial design for one of its input collections.

- or inadvertently, by error, as it was the case with the example in Figure 1.

A crucial requirement, then, is the ability to anticipate breaks in factorial design by inspecting workflow descriptions. Results from such an analysis can be provided as feedback to workflow designers in order for them to understand whether they will later be able to use provenance for result access. For instance, for the astronomy workflow the feedback would be "The extinction results from this workflow are not discretely traceable to the corresponding galaxy ids, discreteness is broken at the $Flatten\_List$ step". Such feedback could be useful in refactoring workflows designs (where possible).

4. Static Analysis Model for Taverna Workflows

We will now provide the core contribution of this paper, the Taverna workflow analysis model. This section is comprised of six sub-sections:

- In Section 4.1 we outline how Taverna workflow descriptions are formally represented.
- In Section 4.2 we discuss how task compositions drive iteration in Taverna. We discuss different types of compositions and iteration.
- In Section 4.3 we outline the information model that our analysis supports. We outline formally Taverna's execution behaviour for the most basic case of iteration (a single-input task) and provide a core set of rules that make predictions on the provenance characteristics of a given workflow.
- In Section 4.4 we illustrate two *dot* and *cross* product, which are key enablers of Taverna's support for factorial design. We show the role of these operations in the complex case of iteration (that of multi-input tasks).
- In Section 4.5 we formalise Broken Factorial Design in terms of the information model given in Section 4.3.
- In Section 4.6 we outline Taverna's behaviour for multi-input iteration and provide corresponding prediction rules.

Throughout Section 4 we utilise and extend a formalisation provided by Missier et al [13] for representing Taverna workflows and their execution behaviour.

4.1. Modelling Taverna Workflow Descriptions

Taverna allows for structuring of data artefacts as nested collections. More specifically;

Definition 1. (Taverna type designators) $\mathcal{T}_{name}$ denotes the set of possible type designators for data processed in Taverna workflows. $\mathcal{T}_{name}$ is comprised of derivations of the following syntax rule:

$$\tau ::= s|L(\tau) \quad where;$$

s denotes the basic $string$ type and $L(\tau)$ denotes list types. s, $L(s)$, $L(L(s)) \in \mathcal{T}_{name}$ are example type designators. We denote nesting of lists with a superscript numbered shorthand; $L(L(s))$ is denoted as $L^2(s)$. We use $\mathcal{T}$ to denote the set of data values that conform to any type in $\mathcal{T}_{name}$.

Definition 2. (Taverna workflow) is denoted with the triple $w = \langle PRO, POR, LINK\rangle$ and the functions in, out, src and snk where;
PRO is the set of processor names,
POR is the set of port names,
$LINK$ is the set of dataflow links among ports,
$in : PRO \rightarrow 2^{POR\times\mathbb{N}\times\mathcal{T}_{name}}$ and $out : PRO \rightarrow 2^{POR\times\mathcal{T}_{name}}$ are two interface functions that maps processors to their ordered inputs and to their outputs with type designators. Each port within a signature of a processor has a unique name. For a particular a input/output $port \in POR$ of a processor $proc \in PRO$ we use the combination of $proc.port$ to refer to it.
$src : LINK \rightarrow PRO \times POR$ and $snk : LINK \rightarrow PRO \times POR$ are two functions that map links to their source and sink ports.
We extend the core workflow model with the following functions:
$procFun : PRO \rightarrow S$, where $S \subseteq \mathcal{T}$ denotes the set of strings, is a function that maps processors to their underlying functions. Multiple processors in a workflow may be underpinned by the same function (recall multiple list fattening processors in the case study workflow).
$lhb : PRO \rightarrow E$ is a list handling behaviour function which maps processors to their corresponding *iteration strategy expressions* obeying the following syntactic rule:

$$\varepsilon ::= (\varepsilon \otimes \varepsilon)|(\varepsilon \odot \varepsilon)|\langle portname\rangle$$

Expressions can be specified using binary *Cross* ($\otimes$) and *Dot* ($\odot$) product operators and port names. Each port appears once in the expression and expressions can be comprised of subexpressions. Iteration strategy expressions encode crucial information on how a processor is to be executed with multiple input collections.

Example 1. (Workflow) In the left hand side of Figure 5 we depict two sample processors with their defined input and output types. The $concat4Str$ processor accepts four inputs each of

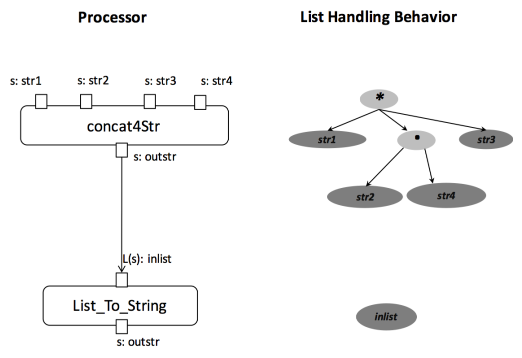


Figure 5: Two sample Taverna processors, their port types and list handling behaviour.

type s, and produces one output of type s. The *List_To_String* processor accepts an input of type $L(s)$ and returns s created by coalescing all items in the input list. The specification of the workflow fragment in Figure 5 would be:

$$\begin{aligned}
PRO &= \{concat4Str, List\_To\_String\} \\
POR &= \{str1, str2, str3, str4, outstr, inlist\} \\
LINK &= \{l1\} \\
\\
in(concat4Str) &= \{\langle str1,1,s\rangle, \langle str2,2,s\rangle, \langle str3,3,s\rangle, \langle str4,4,s\rangle\} \\
out(concat4Str) &= \{\langle outstr,s\rangle\} \\
lhb(concat4Str) &= \{(str1 \otimes (str2 \odot str4)) \otimes str3\} \\
\\
in(List\_To\_String) &= \{\langle inlist,1,L(s)\rangle\} \\
out(List\_To\_String) &= \{\langle outstr,s\rangle\} \\
lhb(List\_To\_String) &= (str1) \\
\\
src(l1) &= \{concat4Str, outstr\} \\
snk(l1) &= \{List\_To\_String, inlist\}
\end{aligned}$$

In right hand side of Figure 5 and throughout the rest of this paper we denote processor's *iteration strategy expressions* with an intuitive diagrammatic view using a List Handling Behaviour (LHB) formula tree. Note that the tree-based representation in Figure 5 is compact in the sense that operators are not binary, they have multiple parameters. This compact form can be expanded into a binary-operator-only tree through addition of nesting in a left associative manner.

4.2. Overview of Task Composition and Iteration Types

Dataflow links among processors is the mechanism of processor composition in Taverna. At the design interface Taverna allows the composition of processors with mismatching nesting-levels of input/output types. At the workflow specification backend Taverna automatically infers the dataflow adjustments required for a successful execution of the workflow. These adjustments can be informally described as follows:

- Simple Composition: This is the straightforward case where the input of a processor is of depth expected by the processor, requiring no adjustment.
- Wrapped Composition: Represents the case when the data supplied to a processor is of a lesser depth than expected. In this case Taverna infers that a *wrapping* dataflow adjustment is necessary, and prepends as many outer lists as necessary around a processor's inputs so that during execution inputs will be of depth acceptable by the processor.
- Iterated Composition: Represents the case where the input supplied is of depth greater than expected, here Taverna requires information in the LHB to determine how to consume inputs. There are two sub cases of iterated composition.
  - Single-Input Iterated Composition: This is the case where the processor has a single input. *List_To_String* processor in Figure 5 is an example. In this case Taverna is prescribed to traverse down the input list structure until the it encounters an item that is of depth acceptable by the processor.
  - Multi-Input Iterated Composition: When a processor accepts multiple collections as inputs, Taverna gives the possibility of creating input combinations configurable with list *Dot* and *Cross* product operations. *concat4Str* processor in Figure 5 is an example.

  In iteration because the processor is exposed to inputs with depth greater than expected, the output(s) will also be of greater depth than defined. The impact of depth differences of inputs to the output depends whether iteration is a single or multi input one, and also depends on the structure of the LHB formula. When one processor iterates it could potentially lead to other downstream processors to iterate.

Example 2. (Processor Composition) In Figure 6 we illustrate how the processors given earlier in Figure 5 are composed as part of a larger workflow. The workflow has 4 input ports which are connected with dataflows to inputs of the *concat4Str* processor. Note that three of workflow's input ports are of type $L(s)$, as a result dataflows from these ports to *concat4Str* correspond to iterated composition, with adjustment consequences for the output of this processor. The amount of depth adjustment for the output of *concat4Str* (the calculation of which we will show later) is 2. Hence while a single invocation of this processor results in a single string (as by its definition), in its composed occurrence in the workflow the overall (iterated) execution of this processor will result in a list of list of strings, $L^2(s)$ at output port *outstr*, designated with call-out in Figure 6. As a ripple effect the iteration of *concat4Str* will cause iteration of *List_To_String*, which will be exposed to an input of depth $L^2(s)$, greater than its defined type $L(s)$. As a result of iteration *List_To_String* will generate an output of type $L(s)$ instead of s. Note here that as a result of iteration of *List_To_String*, its downstream processor *List_To_String_2* will encounter an input that matches exactly its defined input type, henceforth this task will not be iterated and will be executed just once.

We will now present the abstract model for representing information generated through workflow analysis. We formally specify the iteration behaviour of Taverna (for the simple single-input case), and outline a set of rules for making predictions of iteration and the (list) depths of data generated from iterations.

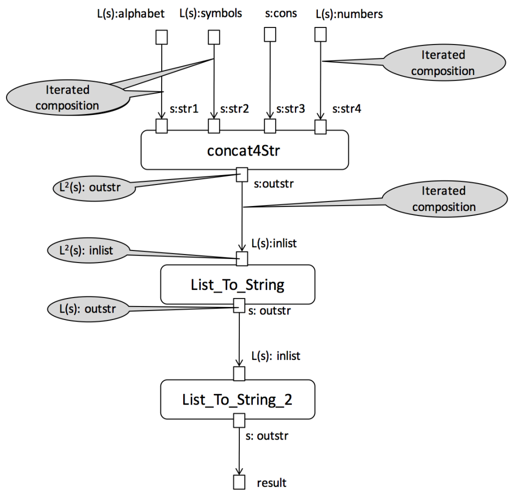


Figure 6: Depth adjustments and iterated compositions inferred for a workflow description.

4.3. Analysis Information Model and Predictions for Single-Input Iteration

Definition 3. (Predicted Provenance Model) Analysis-based predicted provenance PP for a workflow w is denoted is with the quadruple $PP = \langle w, D, M, R \rangle$ where;
D represent relations outlining iteration characteristics of processors and predictions on depths of processor input/outputs.
M represent relations on predicted patterns in provenance.
R denotes a set of rules that populate relations in D and M using relations in w, D and M.

Definition 4. *(Depth Predictions Model)* The information model D for representing depth predictions is comprised of the following relations:
$definedDepth : \langle PRO \times POR \rangle \to \mathbb{N}$ is a function that maps a processor port to the depth-wise size of the data structure that it is defined to consume/produce as per its type designator. Defined depth is a numerical characterisation of type designators of ports. Given $\langle p,t,n \rangle \in definedDepth$ we denote this with $dDep(p.t) = n$.
$predictedDepth : \langle PRO \times POR \rangle \to \mathbb{N}$ is a function that maps a processor's port to the depth-wise size of the data structure that the port is predicted to hold during workflow execution. Given $\langle p,t,n \rangle \in predictedDepth$ we denote this with $pDep(p.t) = n$.
$\Delta Depth : \langle PRO \times POR \rangle \to \mathbb{N}$ is a function that maps a processor's port to the difference (delta) between the predicted and defined depths. Given $\langle p,t,n \rangle \in \Delta Depth$ we denote this with $\Delta Dep(p.t) = n$.
$\Delta Proc : PRO \to \mathbb{N}$ is a function that designates whether a processor is predicted to iterate or not. Given $\langle p,v \rangle \in \Delta Proc$, $v = 0$ denotes that p is predicted to execute once, $v > 0$ denotes that p is predicted to iterate. We denote this tuple with $\Delta Proc(p) = v$.
$\Delta Link : LINK \to \mathbb{Z}$ is a function that designates the kind of composition that a dataflow link represents. Given $\langle l,v \rangle \in \Delta Link$, $v = 0$ denotes simple, $v < 0$ denotes iterated, and $v > 0$ denotes wrapped composition. We denote this tuple with $\Delta Link(l) = v$.

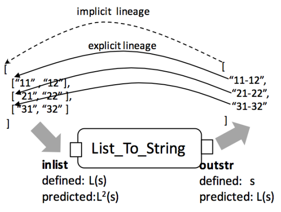


Figure 7: Explicit and Implicit provenance in an iterated processor execution.

Example 3. (Depth Predictions) Consider processor $List\_To\_String \in PRO$ the interface definition of which were given earlier in Example 1. Depth predictions for $List\_To\_String$ due to its composition with other processors in the workflow were illustrated in Figure 6. Using Definition 4 a subset of predictions (the computation of which we will discuss later) is denoted as follows:

$$\begin{array}{r} dDep(List\_To\_String.inlist) = 1 \\ pDep(List\_To\_String.inlist) = 2 \\ \Delta Dep(List\_To\_String.inlist) = 1 \end{array}$$

We will now move on to illustrating how the exposition of Taverna's execution behaviour allows for the definition of rules to populate PP. We denote Taverna execution behaviour (as exemplified in Equation 1) using the functional notation in [13]. Key functions are numbered 1 through to 5 in Section 4 and are also summarised in Table B.1 of the Appendix. The notation for these functions has the following characteristics:

- Behaviour is given in terms of recursive definition that are comprised of a conditional body. The conditions are comprised of 1) a base case and 2) a recursive case.
- Conditional test expressions of each case are given with the infix notation.
- Body of each case is a function call given with the prefix notation similar to most functional programming languages.
- The recursive case involves the use of map function, well-known from functional programming languages. For any other utility function used in the body we provide inline definitions.

Taverna achieves iteration by encapsulating processor functions within a recursive evaluator.

Definition 5. $eval : \mathbb{N} \times S \times \mathcal{T} \to \mathcal{T}$ is a function that applies a designated processor function over a given input data value. The specification of $eval$ is as follows:

$$(eval_l \quad f \quad v) = \begin{cases} (map \quad (eval_l \quad f) \quad v) & |v| > l \\ (f \quad v) & |v| = l \end{cases} \tag{1}$$

$eval$ uses the following utility function in its conditional expressions:

Definition 6. ($|\,|$) *depthwiseSize* : $\mathcal{T} \rightarrow \mathbb{N}$ is a function that returns the depth-wise size of a data value that is of type $\mathcal{T}$. We denote this function with "$|\,|$" as a shorthand. Example applications would be as follows:

$$\begin{aligned} |\, [\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}] \,| &= 1 \\ |\, [[\text{“}x\text{”}, \text{“}y\text{”},], [\text{“}z\text{”}, \text{“}t\text{”}]] \,| &= 2 \\ |\, \text{“}a\text{”} \,| &= 0 \end{aligned}$$

When executing a single input processor $p \in PRO$ with input $i \in POR$ Taverna instantiates the *eval* with l, f, and v, where $l = definedDepth(\langle p,i \rangle)$, $f = procFun(p)$ and v is the data value appearing at port i. *eval* traverses down the input list structure until it encounters an item that is of a depth that is acceptable by the processor, given with l. A sample execution of the *List_To_String* processor with *eval* would be as follows:

$$\begin{aligned} (eval_1\ List\_To\_String\ [[\text{“11”}, \text{“12”}], [\text{“21”}, \text{“22”}], [\text{“31”}, \text{“32”}]]) &= \\ (map(eval_1\ List\_To\_String)\ [[\text{“11”}, \text{“12”}], [\text{“21”}, \text{“22”}], [\text{“31”}, \text{“32”}]]) &= \\ [\ (eval_1\ List\_To\_String\ [\text{“11”}, \text{“12”}]), & \\ (eval_1\ List\_To\_String\ [\text{“21”}, \text{“22”}]), & \\ (eval_1\ List\_To\_String[\text{“31”}, \text{“32”}])\ ] &= \\ [(List\_To\_String\ [\text{“11”}, \text{“12”}]), & \\ (List\_To\_String\ [\text{“21”}, \text{“22”}]), & \\ (List\_To\_String\ [\text{“31”}, \text{“32”}])] &= \\ [\text{“11} - \text{12”}, \text{“21} - \text{22”}, \text{“31} - \text{32”}] &= \end{aligned}$$

During execution each invocation of the processor function (the base case of *eval* in Equation 1) fires an event, which gets caught by Taverna's provenance framework and gets logged. For the iterated *List_To_String* processor, there will be three such event logs, that record explicitly the lineage between the inputs and outputs of the processor function (depicted with solid lines in Figure 7). Note that Taverna does not explicitly record lineage among output lists created in response to traversing down input lists (the recursive case of eval in Equation 1). In Figure 7 we illustrate explicitly recorded lineage with solid lines and implicit lineage (among traversed lists) with dashed lines.

Observation on eval. The exposition of how a processor is executed (Equation 1) and how iteration occurs allows us to anticipate the structural characteristics of data and lineage before a workflow gets executed. Because the evaluator is recursive and exploits the *map* function, this gives us a rule of thumb stating that:

i for each enclosing list structure that we travel down (to find inputs that are of depth acceptable by a processor) a corresponding output list will be generated. As a result each extra (delta) depth for the input will become an extra (delta) depth for the output.

ii each enclosing output list (generated due to a corresponding one in input), will have the same size (number of items) as the corresponding input list.

We will now provide the formal encoding of these rules. Throughout Section 4 we denote rules in the form of "**if**... **then** ..." statements where each statement is a Horn clause (a conjunction of literals implying a single literal) [14]. Each literal represents information in relations of Predicted Provenance Model (Definition 3). In Section 5 we present how these rules are implemented in Datalog.

Definition 7. (Delta Depth Calculation Rules) Given a port r of processor p the following apply:

If $dDep(p.r) = n$, $pDep(p.r) = m$ **then** $\Delta Dep(p.r) = m - n$
If $dDep(p.r) = n$, $\Delta Dep(p.r) = d$ **then** $pDep(p.r) = d + n$.

The delta depth of a port is the difference between its predicted and defined depths. The delta depth calculated for an input port of a processor can be used to determine the iteration of that processor and its output depth, which is given in the following rules.

Definition 8. (Depth Prediction Rules - Single Input Processor) For a single input processor p, where $in(p) = \{\langle i,1,t \rangle\}$ executed with *eval* the following apply:

If $\Delta Dep(p.i) = n$ **then** $\Delta Proc(p) = n$.

A value $n > 0$ denotes that processor p will be iterated.

Definition 9. (Depth Prediction Rule - Processor Output) Using the delta depth of a processor we can predict the depth of its outputs. For each output of processor p, $\langle o,t \rangle \in out(p)$ the following holds:

If $\Delta Proc(p) = n$, $dDep(p.o) = m$ **then** $pDep(p.o) = m + n$.

A delta depth associated with a processor becomes a delta depth its outputs.

Definition 10. *(Depths, Depth Mappings)* The information model M for representing predicted patterns in execution provenance of a workflow w is comprised of the following relations:
Depths $D \subseteq PRO \times POR \times \mathbb{N}^+$ is the set of possible nesting levels for lists that appear at a designated port during workflow execution. We denote a particular depth $\langle p,r,m \rangle \in D$ as $d^{p.r}_m$, where m denotes a depth index. m can range over values in $[1..k]$ where $pDep(p.r) = k$. Depth indices start from 1, which corresponds to the nesting level of 0. A depth with index 1 refers to the top level collection at nesting level 0, whereas a depth with index 2 refers to all collections at nesting level 1.
Depth Mappings $DM \subseteq D \times D$ is the set of predictions on the existence of discrete lineage relations among list items within collection structured data to appear at designated ports. We denote a depth mapping $\langle d^{p1.r1}_{m1}, d^{p2.r2}_{m2} \rangle \in DM$ with $d^{p1.r1}_{m1} \rightarrow d^{p2.r2}_{m2}$. If there is such a mapping we can be assured that for any execution of workflow w there will be discrete lineage relations among list items at nesting levels designated by each depth.

Example 4. (Nested Lists) Intuitively, if we were to think of list structured data of type $L^n(s)$ as an n-dimensional array, a depth would correspond to a particular dimension of that array.

Imagine a processor p with input port *in* where $pDep(p.in) = 2$. During a particular execution of that workflow data of type $L^2(s)$ (as predicted) would appear at port *in*. An example such value could be:

$v1 = [\ [“11”, “12”], [“21”, “22”], [“31”, “32”]\ ]$

The depth $d_1^{p.in}$ would refer to the 1st dimension of this array. $d_1^{p.in}$ is an address for lists at nesting level 0, which for $v1$ is a single list comprised of the set of items {[“11”, “12”], [“21”, “22”], [“31”, “32”]}. Meanwhile $d_2^{p.in}$ is the second dimension, an address for all lists at nesting level 1, for $v1$ these are 3 particular lists, first comprised of items {“11”, “12”}, second comprised of items {“21”, “22”} and so on.

Definition 11. (Depth Definition Rule) For a processor p with input/output port r:

If $pDep(p.r) = n$, $i \in \mathbb{N}$, $1 < i \leq n$ **then** $d_i^{p.r}$.

The number of possible depths for a port is bound by the predicted depth of that port. Whenever the predicted depth of a port is calculated this is used to define individual depths with indices ranging from 1 up to the port's predicted depth.

Our earlier observation on *eval* allows us to model anticipated implicit lineage relations among collections in the form of mappings among depths.

Definition 12. (Depth Mapping Rule - Single Input Processor Evaluation) For a single input processor p executed with *eval*, where $in(p) = \{\langle i, 1, t1\rangle\}$ $out(p) = \{\langle o, t2\rangle\}$ the following apply:

If $\Delta Dep(p.i) = m$, $k \leq m$, $k \in \mathbb{N}^+$, $d_k^{p.i}$, $d_k^{p.o}$ **then** $d_k^{p.i} \rightarrow d_k^{p.o}$.

This rule creates mappings among same indiced depths of input and output ports from 1 up to delta depth of the input.

Intuitively this rule states that lists at depth n of the processor's output will be shaped by corresponding lists at depth n of the input. In other words, if there is such a mapping between two depths, we can be assured that there are discrete lineage relations among each item within lists at corresponding depths. **This assurance brought by the existence of depth mappings provides the foundation of our static analysis**.

Example 5. (Depth Mapping) Using rule in 12 a single depth mapping will be inferred for the *List_To_String* processor, which is $d_1^{in} \rightarrow d_1^{out}$ (we omit processor name prefix for brevity). We know that this mapping implies similar structure so if the workflow were to be run with an input list of (three) items, where each item is itself a list (of two). Then as per rules outlined thus far we can expect the output to be a list of (three) items, where each list item will be string with depth 0 equal to the defined depth of *List_To_String.outstr*.

Missier et al [13] have provided the pioneer insight into the impact of Taverna's iteration on provenance. This insight has allowed us to anticipate implicit lineage (among lists) with the notion of depth mappings. Meanwhile Missier et al [13] have used it to anticipate explicit lineage relations, with a mechanism they call *Index Projection*. Such that, for a given prospective location of an output data item, the location of the contributing input item can be calculated. Note that, both depth mapping and *Index Projection* calculations would be independent from any particular input data size, or data value.

To this end we have analysed processors with single inputs, where the processor did not have an associated LHB formula. We will now move onto multi-input processors with LHB formulas. We start by illustrative examples in the following section.

4.4. Overview of Multi-Input Iteration

As exemplified in Section 4.6 LHB formulas are built with *Dot* and *Cross* product operations. Both operations creates tuples out of items they encounter in lists, which could themselves be tuples. Note that in a repeated application of these operations, n-ary tuples will be generated. Note however in the formal representation of Taverna workflows there is no separate type for tuples. To adhere to this restriction we assume that *Dot* and *Cross* product operations are underpinned by functions that accept and return *string* types, where the result is a *string* based representation of a tuple.

Cross product is a *cartesian product* function that adheres to the nested structure of the lists, similar to the recursive evaluator given earlier, when the cross product encounter lists it will traverse down to find items of appropriate depth. So unlike the traditional cartesian product, which creates a list out of two lists, the Cross product will create an output of type $L^2(s)$ when applied over two inputs of type $L(s)$ as exemplified below:

$(cross\ \ [“a”, “b”, “c”]\ \ [“1”, “2”]) =$
$[[“a{\times}1”, “a{\times}2”], [“b{\times}1”, “b{\times}2”]], [“c{\times}1”, “c{\times}2”]]$

Dot product is similar to the *zip* function found in many programming languages. It will pair up items at same positions in each list. Similar to Cross product the Dot product also obeys the nested structure of lists. So the Dot product of two inputs of type $L^2(s)$ will result in an output of type $L^2(s)$ as exemplified below:

$(dot\ \ [[“x”, “y”], [“z”, “t”], [“u”, “v”]]\ \ [[“1”, “2”], [“3”, “4”], [“5”, “6”]]) =$
$[[“x{\times}1”, “y{\times}2”], [“z{\times}3, “t{\times}4”], [“u{\times}5”, “v{\times}6”]]$

In the previous section we saw that the driver of iteration is the depth difference, delta, between the expected and encountered depth of input i of a processor p. An intuitive way to think about delta depth is to think of it as a space of dimension $\Delta Depth(p.i)$ to be explored. In the case of multiple inputs, the application of the processor to input lists is preceded by operations (e.g. dot/cross product) that are responsible for creating an overall input space from individual spaces supplied for each input . This process of input space creation is informed by the LHB formula. We provide the intuition for parameter spaces with two examples. Later in Section 4.6 we will formally specify this process and corresponding prediction rules.

Example 6. (Cross Product Based Input Space) Consider the case in Figure 8. *concat3Str* is defined to be a processor that has three input ports, each accepting singleton strings (input port type *s*), and returns their concatenation. In this example each input port is instead exposed to a list of strings $L(s)$. Each such list represents a 1-dimensional space for respective input parameters. The LHB formulas are left-associative in Taverna the *Cross* product of three lists will be created by first creating the cross product of the first two lists and then crossing the result with the third. As a result a 3-dimensional input space (a nested list of depth 3), comprised of input triples, will get generated. These triples represent all possible combinations of inputs from respective input lists. Once the input space is prepared Taverna will explore this space by using the recursive evaluator given earlier in (1) and will generate an output per input in this space. The mappings between the dimensions of individual inputs and the output depends on the order of inputs in the LHB formula, and the operator that combines them. For *concat3Str* the mappings that will be inferred based on the analysis of workflow description will be : $d_1^{alphabet} \rightarrow d_1^{result}$, $d_1^{symbols} \rightarrow d_2^{result}$, $d_1^{numbers} \rightarrow d_3^{result}$.

Example 7. (Cross and Dot Product Based Input Space) We exemplify the use of Dot product in Figure 9. As discussed before, this operator expects its inputs to be of same depthwise size, and returns an output same size as the inputs. Here the LHB prescribes a Dot product of input spaces for $str2$ and $str4$. The delta for both of these inputs is 1, hence a Dot product is possible, which creates a single input space that is representative for both inputs. This space is then combined with the input space for $str1$ with a cross product. Note that the formula also prescribes how input spaces for $str3$ should contribute to the overall space. However, as the input for the ports is of expected depth, i.e. has no delta depth. There is no input space to be explored for $str3$. As a result each input quadruple in the overall input space will have the same value for str3. The inferred depth mappings will be as follows: $d_1^{str1} \rightarrow d_1^{result}$, $d_1^{str2} \rightarrow d_2^{result}$, $d_1^{str4} \rightarrow d_2^{result}$.

In both examples in Figures 8 and 9 the defined depth for the output of (string concatenation) processors is 0 (each invocation produces a single string). As a result the application of the processors will result in an output list structure that is of the same depth as the dimension of the overall input parameter space.

4.5. Modelling Broken Factorial Design

So far we outlined the elements of workflow analysis on a per processor basis. Our ultimate goal is to perform the analysis over a workflow comprised of multiple processors. As illustrated in Section 3 broken factorial design corresponds to an activity record in execution provenance where the activity consumes data descending from distinct workflow input parameters. In this section we illustrate and formalise this pattern as a discontinuation of depth mappings.

Example 8. (Broken Factorial Design) In Figure 10 there are two processors, composed by a dataflow link. *concatStr* pro-

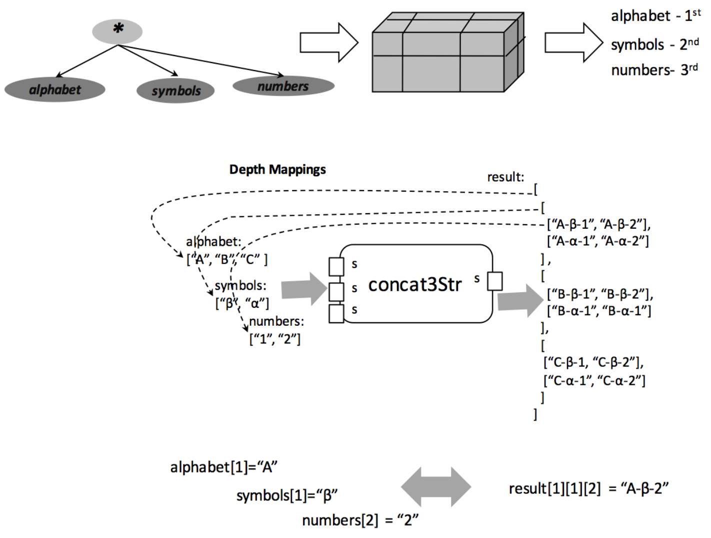


Figure 8: Illustration of input space resulting from Cross product of three parameters.

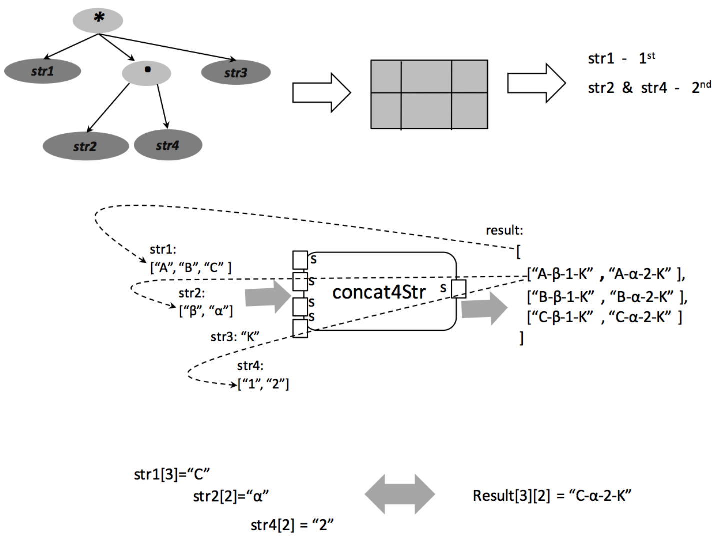


Figure 9: Illustration of input space using a combination of Cross and Dot over four parameters.

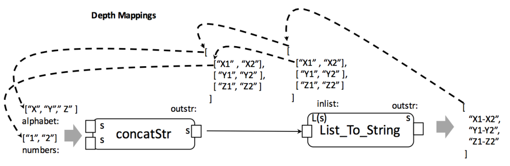


Figure 10: A broken factorial design illustrated with discontinued depth mapping at the *List_To_String* step.

cessor explores an input space of 2-dimension (based on the cross product of its inputs). As per rules that will be given in Section 4.6 $\Delta Proc(concatStr) = 2$. Two depth mappings will be created for this processor.

- For $concatStr$: $d_1^{alphabet} \rightarrow d_1^{outStr}$, $d_1^{numbers} \rightarrow d_2^{outStr}$

As per rules that will be given in Section 4.6 when the data from the output of one processor is transferred to the input port of a follow on processor via a dataflow link, depth mappings will be created among the link source and sink.

- For the dataflow link: $d_1^{outStr} \rightarrow d_1^{inList}$, $d_2^{outStr} \rightarrow d_2^{inList}$

Based on the predicted output depth of $concatStr$ processor, $List\_To\_String$ will be exposed to $L^2(s)$, greater than its defined input type $L(s)$, hence we'll have $\Delta Dep(List\_To\_String.inlist) = 1$ and $\Delta Proc(List\_To\_String) = 1$. Note that as the delta for this processor is 1, it will traverse down only one level to find data of acceptable depth. One such input item is [“$X1''$,“$X2''$], which results in the output “$X1 - X2''$. With such an invocation $List\_To\_String$ consume data all at once from the second dimension (depth) of the output of $concatStr$. As per the rule in Definition 12 the following mappings will be generated:

- For the $List\_To\_String$ processor: $d_1^{inlist} \rightarrow d_1^{outStr}$

Note here that we are unable to create a mapping for d_2^{inlist} as this depth is beyond the delta depth for $inlist$. In other words this depth is part of the data we consume, rather than a dimension we explore. This inability to map an input depth (which is the target of a prior mapping) to any output depth represents the break in factorial design, which can be defined as follows:

Definition 13. (Broken Factorial Design) Let $reachesDepths : D \rightarrow 2^D$ be a function that accepts an input depth and returns the set of depths that are reachable from the given depth by traversing the depth mapping relations in w. We state that workflow w has broken factorial design for depth $d_i^{p.t}$ if the set $\{d_j^{q.r} \in reachesDepths(d_i^{p.t}) | j > \Delta Depth(\langle q, r \rangle)\}$ is non-empty.

4.6. Predictions for Multi-Input Iteration

To this end we only illustrated how input spaces look like in several multi-input iteration scenarios, and what the depth mappings would be. We will now outline a high-level model for the process that Taverna follows to prepare input spaces, to apply processors onto these input spaces and to carry data among data

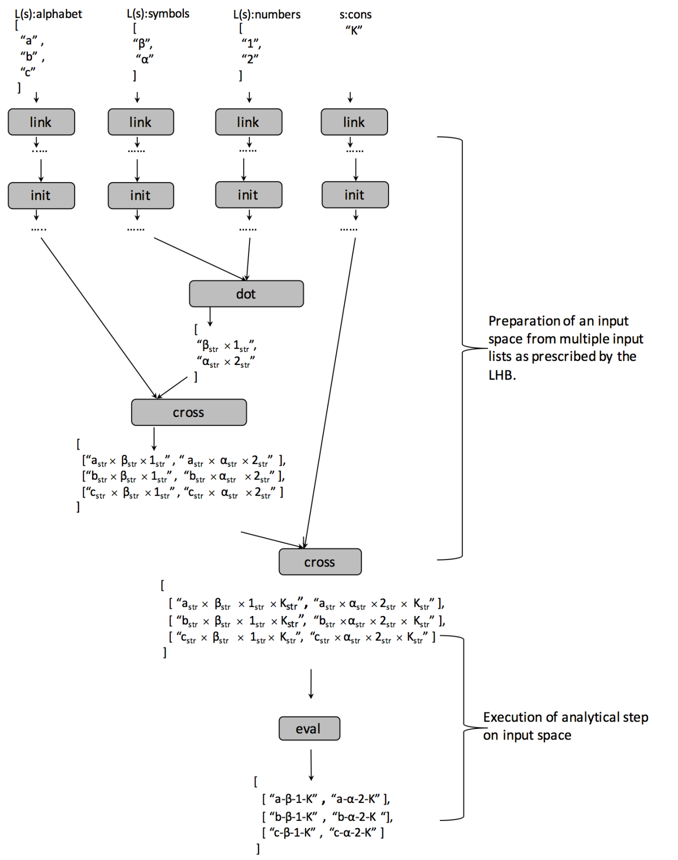

L(s):alphabet
L(s):symbols
L(s):numbers
s:cons
link
init
dot
cross
cross
eval
Preparation of an input space from multiple input lists as prescribed by the LHB.
Execution of analytical step on input space

Figure 11: Illustration of iterated execution of $concat4Str$ over multiple input lists with respect to its LHB definition.

links. After that we will provide (in functional terms) the formal specification of each step of this process. Based on these specifications we will provide depth and depth mapping prediction rules.

A convenient way to model the process that Taverna adopts in executing workflows is to break it down to a set of known computations. We illustrate these for the execution of $concat4Str$ processor in Figure 11. There are different four types of computations involved:

- the carrying of data through a dataflow link. Depicted with *link* boxes in Figure 11.

- the initialisation of a processor's (multiple) inputs and identifying whether there exists an input space to be explored or a single input. Depicted with *init* boxes in Figure 11.

- the build up of the space of input tuples from the individual input spaces using the LHB formula as a guideline. Depicted with *dot* and *cross* boxes in Figure 11.

- applying the processor to each input tuple in this space. Depicted with the *eval* box in Figure 11.

Taverna's execution behaviour for a dataflow link is given with the *link* function.

Definition 14. *link* : $\langle \mathbb{N} \times \mathcal{T} \rangle \rightarrow \mathcal{T}$ is a function accepts an input value and a target depth and returns an output data value with depth wise size equal to the target depth. *link* ensures that the nesting level of the result is equal to the target depth by wrapping the input with as many enclosing lists as necessary and returns. The specification of *link* is as follows (bracket [ ] represents list constructor):

$$(link_d \quad v) = \begin{cases} (v) & \text{if } d \leq |v| \\ (link_d \quad [v]) & \text{if } d > |v| \end{cases} \tag{2}$$

When realising a dataflow link $l \in LINK$, where $src(l) = \langle p1, o \rangle$ and $snk(l) = \langle p2, i \rangle$, Taverna instantiates the *link* function with the data value v appearing at the link source, port o of processor $p1$, and the defined depth of the link's target port $d = definedDepth(\langle p2, i \rangle)$. In Figure 11 each four links's targets correspond to input ports of the multi-input *concat4Str* processor.

Definition 15. (Depth Prediction and Mapping Rules - *link*) For a dataflow link $l \in LINK$, where $src(l) = \langle p1, o \rangle$ and $snk(l) = \langle p2, i \rangle$ realised with the *link* function (as per Definition 14) and the following apply:

If $pDep(p1.o) = n$, $dDep(p2.i) = m$ **then** $\Delta Link(l) = m - n$

Difference between the link target's defined and link source's predicted depth is the delta for the link. It's value determine the type of composition for the link.

If $\Delta Link(l) = d$, $dDep(p2.i) = m$, $d > 0$ **then** $pDep(p2.i) = m$.
If $\Delta Link(l) = d$, $pDep(p1.o) = n$, $d \leq 0$ **then** $pDep(p2.i) = n$.

A positive delta represents wrapped composition and the predicted depth of target would be the same as its defined depth. A negative delta represents iterated, a zero delta represents simple composition. In these latter cases no wrapping occurs hence the predicted depth of the link's target would be the same as its source.

If $\Delta Link(l) = d$, $d > 0$, $d_n^{p1.o}$, $d_{n+d}^{p2.i}$ **then** $d_n^{p1.o} \rightarrow d_{n+d}^{p2.i}$.
If $\Delta Link(l) = d$, $d \leq 0$, $d_n^{p1.o}$, $d_n^{p2.i}$ **then** $d_n^{p1.o} \rightarrow d_n^{p2.i}$.

If the link represents wrapped composition, then each depth of the source map to depths of the target with an index shift factor equal to delta of the link. For simple or iterated composition, no wrappings are made, so all depths of source map to same indiced depths of target.

Definition 16. *init* : $\mathbb{N} \times \mathcal{T} \rightarrow \mathcal{T}$ is a function that replaces items (that are of a designated depth) within a (nested) input collection with their string-based representation. *init* can be realised using *eval* by passing *makeStr* (defined next) as the processor function parameter.

$$(init_l \quad v) = (eval_l \quad makeStr \quad v) \tag{3}$$

Definition 17. *makeStr* : $\mathcal{T} \rightarrow S$ is a function that returns the string-based representation of a given input value. If the input is a string than the result is simply equal to the input. If the input is a (nested) collection than we assume the result is some string-based representation of this collection.

Note that *init* is single input processor and depth predictions and mappings for given *eval* in Definition 12 also apply to *init*.

The purpose of *init* is to truncate depths within data collection that correspond to data values that will be consumed by the follow-on processor evaluation step (*eval* in Figure 11), for which an input space is being prepared. We assume such a behaviour for our convenience as it simplifies depth mappings for follow-on (*Dot* and *Cross* product) steps. As a result of truncation, depth wise size of *init*'s output represents the dimension of the input space for an individual one of multiple inputs that will be consumed by the by the follow-on processor evaluation step.

Definition 18. *makeTuple* : $S \times S \rightarrow S$ is a function that accepts two strings representing data values or tuples of values and returns a string representation of a tuple of the two input values. Example tuples created while preparing the input space for the *concat4Str* are given in Figure 11.

Definition 19. *cross* : $\mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ is a cartesian product function for nested lists. The functional specification for *cross* given below is comprised of a two phased evaluation based on the use of an additional function *cross2* : $S \times \mathcal{T} \rightarrow \mathcal{T}$.

$$(cross \quad b \quad a) = \begin{cases} [(map \quad (cross \quad b) \quad a)] & \text{if } |a| > 0 \\ [(map \quad (cross2 \quad a) \quad b)] & \text{if } |a| = 0 \end{cases}$$

$$(cross2 \quad a \quad b) = \begin{cases} (makeTuple \quad a \quad b) & \text{if } |b| = 0 \\ [(map \quad (cross2 \quad a) \quad b)] & \text{if } |b| > 0 \end{cases} \tag{4}$$

In the first phase we traverse down the first input collection until a string item is reached. Afterwards the second phase (*cross2*) starts, where using the item located in the first phase we traverse down the second collection until a string item is reached. A tuple is created out of the two string items located.

Definition 20. (Depth Prediction and Mapping Rules - *cross* product) For a two-input processor p, realised with *cross* having $in(p) = \{\langle a, 1, s \rangle, \langle b, 2, s \rangle\}$ and $out(p) = \{\langle c, s \rangle\}$ the following apply:

If $\Delta Dep(p.a) = n$, $\Delta Dep(p.b) = m$ **then** $\Delta Proc(p) = n + m$.

The sum of the delta depths on the two inputs of *cross* becomes a delta for the *cross* product step.

If $d_j^{p.a}$, $d_j^{p.c}$ **then** $d_j^{p.a} \rightarrow d_j^{p.c}$.

The depths of the first input will map to same indiced depths of its output.

If $\Delta Dep(p.a) = n$, $d_j^{p.b}$, $d_{j+n}^{p.c}$ **then** $d_j^{p.b} \rightarrow d_{j+n}^{p.c}$.

The depths of the second input will map to depths of the output with an index shift factor equal to the delta depth of the first input.

Definition 21. $dot : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ is a zip function for nested lists. The functional specification for *dot* is as follows (note in the recursive case, we map *dot* onto two lists).

$$(dot \quad a \quad b) = \begin{cases} (makeTuple \quad a \quad b) & \text{if } |a| = |b| = 0 \\ [(map \quad dot \quad a \quad b)] & \text{if } |a| = |b| > 0 \end{cases} \tag{5}$$

Definition 22. (Depth Prediction and Mapping Rules - *dot* product) For a two-input processor p with $in(p) = \{\langle a, 1, s\rangle, \langle b, 2, s\rangle\}$ $out(p) = \{\langle c, s\rangle\}$ realised with *dot* function the following apply:
If $\Delta Dep(p.a) = n$, $\Delta Dep(p.b) = n$ **then** $\Delta Proc(p) = n$.

The delta depths on the two inputs of *dot* becomes a delta for the *dot* product step.

If $d_j^{p.a}, d_j^{p.c}$ **then** $d_j^{p.a} \to d_j^{p.c}$.
If $d_j^{p.b}, d_j^{p.c}$ **then** $d_j^{p.b} \to d_j^{p.c}$.

In terms of depth mappings depths of both the first and second input will map to same indiced depths of the output.

Once the input space is prepared then we apply the processor by using a single input recursive evaluator *eval*, for which depth predictions and mappings were given in Definition 12. This evaluator will simply traverse down each list until it finds non-list items (tuples), then invokes the function associated with the multi-input processor (e.g. *concat*4*Str*) using this tuple. As *eval* is preceded with the preparation of the overall input space. In terms of depth mappings each depth in the input space will be mapped to same indiced depth of the output of processor evaluation.

5. Implementing Analysis With Datalog

We adopt a declarative logic programming approach, more specifically Datalog [14], for implementing the static analysis of workflows. The analysis is *static*, as it is performed solely over the workflow description, without running the workflow. An analysis program is comprised of 1) the extensional database (EDB), which is a collection of facts, and 2) the intensional database (IDB), which is a set of rules used to deduce facts. A detailed description of rule implementations and illustration of their execution is given in Appendix A.

A modular overview of our Datalog implementation is given in Figure 12. We represent the **Workflow Description** as a set of EDB facts. For illustration Table 1 provides a fragment of the facts representing the workflow in Figure 6. With our facts we represent, workflows, processes (tasks) and their input/output ports. For each processor we also have facts that represent the LHB formula tree. The EDB also provides the defined depths for all ports. In order to kickstart depth prediction rules, the

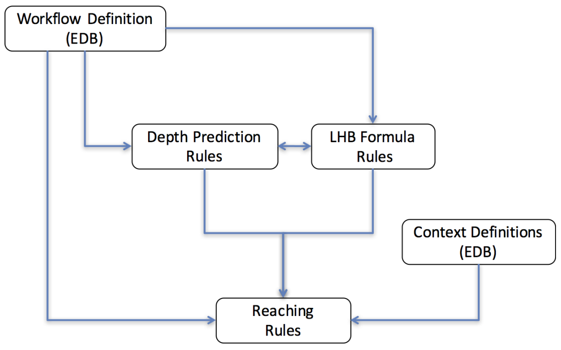


Figure 12: Modules of rules used for our analysis and their dependencies.

EDB also contains the predicted depth for the workflow input ports, which equal their defined depths.

We implement the rules given in Section 4 with three main modules ("Depth Prediction", "LHB Formula" and "Reaching" in Figure 12). Their functionalities are described briefly here with details given in Appendix A.1, Appendix A.2 and Appendix A.3 respectively.

Depth Prediction rules, which, starting from the workflow's input's, will make the following predictions:

- delta and predicted depths of ports,
- the kinds of composition each dataflow link represents and the implied depth mapping index shift factors for wrapped composition,
- the size of the input space of processors, and based on that, the predicted depths of a processor's outputs. In order to calculate the input space size Depth Prediction Rules utilise facts produced by the **LHB Formula** rule module.

The **LHB Formula** rules encapsulate size calculation for dot and cross product operations and a traversal logic for computing the overall space size of a given LHB formula tree. This module also calculates depth mapping shift factors for each depth of processor's input ports.

We allow users to define an input parameter space of a workflow based analysis as a **Context**. A context is an extensional fact, a named combination of a workflow input port and a depth, which represents an input data/parameter collection that drives iteration within a workflow. An example of a context is the list of galaxy names in the Astronomy workflow (*list_cig_name* in Figure 1). A context therefore represents those parameters collections, which are later to be used in provenance queries as an index to reach results of interest. We provide **Reaching Rules**, which, starting from a given context, compute depth mappings throughout the dataflow links and processors of a workflow. Reaching rules use the depth mapping adjustment factors computed by the Depth Prediction and LHB Formula modules. In this module we also broken factorial design, by flagging those processor input ports where depth mappings are discontinued.

We have developed a tool that converts Taverna workflow descriptions represented with the abstract Wfdesc [8] model

```
workflow(w1).
workflowInput(w1,alphabet). workflowInput(w1,symbols).
workflowInput(w1,cons). workflowInput(w1,numbers).

definedDepth(w1,alphabet,1). predictedDepth(w1, alphabet,1).
. . . . . . . . . . . . . . . . . . . . . . . . . . .
process(concat4Str).
processInput(concat4Str,str1). processInput(concat4Str,str2).
processInput(concat4Str,str3). processInput(concat4Str,str4).
processOutput(concat4Str,outstr).

dataLink(dl1,w1,alphabet,concat4Str,str1).
dataLink(dl2,w1,symbols,concat4Str,str2).
dataLink(dl3,w1,cons,concat4Str,str3),
dataLink(dl4,w1,numbers,concat4Str,str4).

%LHB FORMULA TREE STRUCTURE
hasLhbRoot(concat4Str,uid1).

lhbNode(uid1, cross, concat4Str).
lhbNode(uid21, str1, concat4Str).
lhbNode(uid22, dot, concat4Str).
lhbNode(uid23, str3, concat4Str).
lhbNode(uid31, str2, concat4Str).
lhbNode(uid32, str4, concat4Str).

hasChild(uid1, uid21, 0).
hasChild(uid1, uid22, 1).
hasChild(uid1, uid23, 2).
hasChild(uid22, uid31,0).
hasChild(uid22, uid32,0).
```

Table 1: EDB Facts for representing a workflow.

into a set of EDB facts in DLV Datalog syntax [15]. The source for the converter tool, the rule modules and the complete EDB for the Astronomy workflow in Section 2 can be found at [16].

6. Factorial Design in Workflow Systems

In this section we survey state of the art workflow systems to understand their level of support for factorial design. Survey includes Taverna, Wings/Pegasus, Galaxy, Vistrails, Kepler, as these systems have documented provenance capabilities. First we give brief overviews (except for Taverna as it was introduced earlier), then we provide a comparative discussion. (The narrative follows from Table 2[1].)

Wings [17] is a "semantic" workflow system that operates with resources (data and analytical components) made part of a catalogue prior to use in workflows. During cataloging Wings collects semantic descriptions and constraints about resources. A semantic description could state that a task performs a particular function, a constraint could state that the input and output data of some task should be about the same study subject. Wings utilises semantic descriptions to validate task compositions during workflow design, and to propagate attributes of inputs to results during workflow execution.

Galaxy [18] is a web/cloud-based data analysis platform used widely in biomedical disciplines. Galaxy combines of command line tools with data in workflows. Similar to Wings, Galaxy operates over controlled resources and is similarly backed by a catalogue. During catalogue introduction, Galaxy expects scientists to supply domain-specific, structured descriptions denoting data types or tool functions. Galaxy combines these descriptions with execution traces to publish them online as workflow execution (histories).

[1]We use a full circle to denote enhanced, half circle to denote partial and empty circle to denote minimal/no support for Factorial Design

System	Process Granularity	Data Granularity	FD. Support	Prone to n-by-m
Taverna	Processor Iteration Sub-workflow	Fine	●	Y
Kepler	Actor Ramp, Repeat, Loop Sub-workflow	Coarse Data, Coarse Parameter	○	N
Kepler (COMAD)	Actor Data Bindings	Coarse Parameter, Fine Data	◓	Y
Galaxy	Steps	Coarse Data, Coarse Parameter	○	N
Wings	Component Component Collection	Coarse parameter, Fine Data	◓	Y
Vistrails	Modules Parameter exploration, map/fold sub-workflow	Fine Parameter, Coarse Data	◓	N

Table 2: Comparative Table of Workflow Systems' Support for Factorial Design.

Vistrails [4] system is specialised for visualisation workflows. Vistrails embodies a comprehensive library of visualisation tools, which operate over a common information model. An inherent characteristic of visualisation workflows is that they are produced via a long and exploratory design process and Vistrails uses provenance to support this process. Vistrails keeps track of all intermediate workflow designs and allows scientists to compare and contrast execution results of design alternatives, as well as different parameter values.

Kepler [3] system, used in several scientific domains, is based on the Ptolemy II framework, which adopts a Dataflow Architecture of computing [19]. Each flavour of dataflow computation is manifested in Kepler as a Director, that dictates the mechanics of dataflow among analytical steps. Kepler utilises provenance to perform smart workflow re-runs [20], where subparts of a workflow can be selectively re-executed using cached outputs of upstream portions of a workflow.

6.1. Constructs for Factorial Design

A common aspect of provenance collection for all systems is that processes are recorded at the granularity of tasks in a workflow. Tasks are called *Processors* in Taverna, *Actors* in Kepler, *Modules* in Vistrails, *Components* in Wings and *Steps* in Galaxy. Furthermore, provenance collection in all systems can see into control constructs, be they iteration, or sub-workflows and record the provenance of tasks within those control constructs. Meanwhile, there is variation in the granularity of modelling data and this is a critical factor in determining a system's support for factorial design.

As discussed earlier in Section 2 **Taverna** supports fine-grained modelling of data consistently at the workflow and provenance level. Collections are the main driver of iteration in Taverna. Differences in the expected and defined cardinality of input determine whether a *Processor* is to iterate or not. Taverna gives the scientists the flexibility of defining multi-dimensional parameter spaces with *dot* and *cross* product and complex iteration strategies. Taverna does not differentiate between parameters and data, which means both data/parameter sweeps are built using the same language constructs. Henceforth Taverna fully supports factorial designs.

Wings permits fine-grained modelling of input data as *Files* and *FileCollections* and supports (coarse-grained) parameter definition. Granularity of representation for data is uniform across workflows and provenance. Wings provides *Component Collections* for task iteration and uses this feature to sweep a pre-configured analysis over multiple input datasets. Unlike Taverna, comprehensive parameter spaces cannot be implemented as parameters are coarse entities and iterations are not configurable. Hence Wings provides partial support for factorial design

Vistrails system has a fixed parameter exploration capability built-in. More specifically, given *n* input parameter collections, Vistrails builds an n-dimensional parameter space based on a cartesian product of these collections, upon which the entire workflow is iterated. Vistrails also supports structured data types with *lists*. Iteration (within a workflow) over lists is achieved with higher-order modules such as *map* and *fold* that encapsulate analytical modules and the lists that the analysis should be applied on. While parameter collections get distinctly represented in provenance list types appears as coarse data nodes. Due to lack of controllable parameter space build-up and coarse data modelling Vistrails is categorised to have partial support for factorial design.

Kepler's capabilities depend on the different Directors used.

- Kepler with its default set of Directors provides weak support for encoding factorial design. Kepler supports structured data types with *arrays*, however, similar to Vistrails, such structured data gets modelled coarsely in execution provenance. Kepler supports the modelling of parameters as distinct design elements, however collection-typed parameters are not supported. Kepler provides iterated analyses via *ramp*, *repeat* and $feedback - loop$ constructs, however as both data and parameters are modelled coarsely, this capability is insufficient on its own to represent factorial designs.

- The Collection-Oriented Modelling and Design (COMAD) director of Kepler [21] provides improved, yet partial support for factorial designs. COMAD allows finer-grained data modelling with nested collections of datasets to be recognised as first class design elements in workflows. COMAD lacks control constructs for task iteration, meanwhile alternative mechanisms, called *data bindings* are in place. In this approach inputs, intermediary and final output data is to kept in one single hierarchical data structure that is passed through every task in the workflow in an assembly line manner. Each task has a data binding specification (somewhat similar to an XPATH query over XML) that specifies the data of interest for that task. When a binding specification adresses multiple data sub-hierarchies this can be used to sweep a particular analysis over data collections. While data is modelled at a fine-granularity, parameters are coarsely represented, hence parameter explorations cannot defined and managed within workflows.

Galaxy system does not provide support for encoding factorial design into workflow descriptions. Here data/parameters are modelled coarsely both in the workflow description and in execution traces. Galaxy also lacks control-constructs such as iteration or sub-workflows.

Systems that support factorial design within workflows, such as Taverna, Wings, Kepler (COMAD) and Vistrails bear the potential of yielding parameter-to-result traceability in provenance. However, having support for factorial design can become a double-edged sword when factorial design is broken. In Wings, Taverna, Kepler (COMAD) the encoding of factorial design is left to the user, and there are no restrictions in place that would prevent the user from creating a design that would lead to the *n-by-*1 or *n-by-m* pattern in provenance. Meanwhile in Vistrails the control of factorial design is not left to the user, instead the workflow system is in charge of factorial design, albeit providing it in a restricted manner (only allowing cartesian (cross) product of inputs, and iterating the entire workflow).

6.2. Exploiting Factorial Design

In the introduction to this paper we discussed that workflows are beneficial to scientists in analysis automation; particularly in settings where analyses are swept over data/parameter sets with the intent of performing post-execution activities such as result comparison, parameter calibration or checking for correlations among parameters and results. We also mentioned the imbalance in the level of tool support; ample for the development and execution of workflows, weak for the post-execution phase. Among the systems surveyed earlier, the only one that exploits provenance for post-execution result access is Vistrails. Figure 13 displays Vistrails' result comparison (spreadsheet) interface, a typical example fine-grained workflow provenance in action, where Vistrails capitalises on its tightly controlled approach to factorial design and the resulting provenance with guaranteed parameter to result traceability. Note that this does not mean that scientists using other workflow systems are unable to perform the aforementioned post-execution activities; instead they resort to custom-built solutions that rely on adhoc (or self-collected) provenance (file names, data values, folder structures) rather than workflow execution provenance (with potentially broken factorial design).

Self provenance collection is illustrated in the workflow in Figure 14 . The workflow decorates a given set proteins (denoted with accession numbers *ENSP....*) with their universal identifiers pulled from a designated repository (*SWISSPROT*) under a designated taxonomy (9606). The workflow contains several tasks for accessing the repository for data retrieval as well as a final task, *build_mapping_table* for recording provenance in the form of a mapping table that traces collected uni-

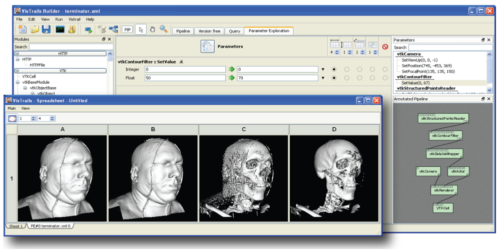
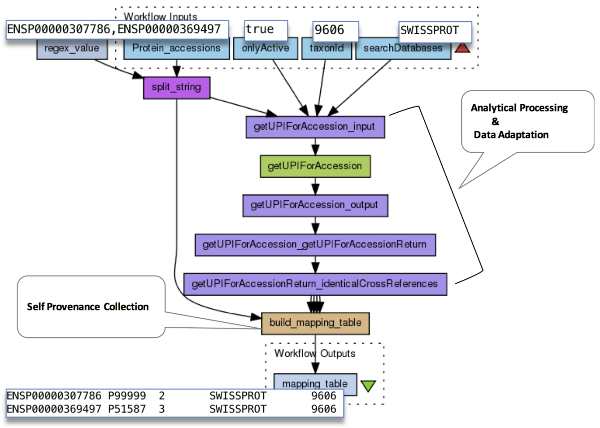


Figure 13: Comparing results derived from different parameter values in Vistrails.

versal identifiers to corresponding accession numbers and other parameter settings. Such a traceability table might as well have been generated by workflow tooling using provenance, under the condition that provenance reflects factorial design, which for this particular workflow is not the case as the scientist configures the workflow with a single coalesced string of accessing numbers, instead of a collection. One disadvantage of self provenance collection, as illustrated here, is that it obfuscates the ultimate analytical intent of the workflow by mixing up analytical and reporting tasks. On the other hand adhoc provenance as in file names etc., has the advantage of being easily portable as provenance is co-located with data. Moreover, once adhoc provenance is created, it can be read/written by tools that operate over data.



Workflow Inputs
ENSP00000307786,ENSP00000369497
true
9606
SWISSPROT
regex_value
Protein_accessions
onlyActive
taxonId
searchDatabases
split_string
getUPIForAccession_input
Analytical Processing & Data Adaptation
getUPIForAccession
getUPIForAccession_output
getUPIForAccession_getUPIForAccessionReturn
getUPIForAccessionReturn_identicalCrossReferences
Self Provenance Collection
build_mapping_table
Workflow Outputs
mapping_table

```
ENSP00000307786 P99999  2       SWISSPROT       9606
ENSP00000369497 P51587  3       SWISSPROT       9606
```

Figure 14: Combination of analytical processing and adhoc provenance collection in a Taverna workflow.

An immediate question that springs to mind is: “how common is broken factorial design in real-world datasets and how often scientists rely on self provenance collection”. In a previous empirical study [22] we surveyed 260 publicly shared workflows[2] to understand the high-level nature of data processing within them. The survey revealed that a minority 30% of tasks perform the scientific heavy lifting in a workflow (analyses, data collection, visualisation), whereas the remainder 70% are dedicated to data adaptation. Furthermore among those adapters, 20% perform a *Flattening/Merging* function, where collections of data, or data coming from different workflow branches are combined into single items, typically for the purpose of reporting (self provenance collection). On the other hand 5% of all adapters perform *Splitting*, which is the inverse of *Flattening*. These numbers tell us that, at a localised level, the $n-by-1$ or the $n-by-m$ pattern is commonplace in real-world provenance traces. We know, from examples, that some of those local patterns do indeed amount to broken factorial design at a global (whole-workflow) level. We do not state the exact rate of the co-occurrence of $n-by-m$ and broken factorial design in public workflows, as this would require running the static analysis over each workflow configuring each run with extensive domain-specific knowledge to determine input parameter(s) of interest, for which the health of factorial design shall be assessed.

In fairness, the difference in the level of tool support between Vistrails and other workflow systems is due to their operating assumptions. Vistrails targets a specific domain (visualisation) with a controlled resource environment (a local library of tools and a common information model). As a result Vistrails can afford to restrict users in implementing factorial design. On the other hand, other workflow systems, such as Taverna or Kepler, cater for a range of domains and therefore assume an uncontrolled/open resource environment. These systems have to 1) accommodate heterogeneous domain formats, which, for instance, may expect parameters and data sets to be bundled; or 2) integrate remote resources, which, due to access overhead, discourage scientists from a fine-grained integration, such as the iterated invocation of a web service. In short, it is more difficult to enforce factorial design in workflow systems with uncontrolled resource environments.

7. Related Work

As mentioned earlier Missier et al [13] have provided the initial insight that, for Taverna workflows, we can pre-compute anticipated lineage based on locations of data in nested input and output collections, as Taverna provides us with its iteration semantics. This work exploits the definitions of the cross product and the iterative processor evaluator to provide a location mapping formula among indices of prospective input/output data collections. Authors have shown that the cost of answering lineage queries using location-mappings would be resilient to increases in workflow size, when compared to the naive query answering based on traversal of lineage. In [23] Dey et al describe a similar approach for the Kepler workflow system’s Synchronous Dataflow (SDF) Director [3]. In Kepler’s SDF workflows activities consume/produce data tokens to/from containers (similar to ports) with defined rates. Dey et al provide a location-based lineage computation formula based on token consumption/production rates associated with each activity. Similar to Missier et al, Dey et al demonstrate that location-based provenance provides benefits in lineage query answering in settings where the workflow size increases.

The benefit of both Missier’s [13] and Dey’s [23] location-based provenance approaches is demonstrated in cases where

[2]Developed with Taverna, Wings, Galaxy and Vistrails systems; belonging to 10 domains, including Life Sciences, Astronomy and Earth Sciences amongst others.

workflows are very large (100+ steps) and all workflow steps are iterated over respective input collections. As a result authors demonstrate gains with synthetic datasets. In our approach we have focused on what can go wrong in querying the provenance of real-world workflows. Our case study has shown that when the iteration structure is not sustained it significantly effects the accuracy of provenance query results. Therefore we have focused on eliciting the case to the contrary of Dey and Missier.

Within a workflow the flow of data among modules is explicit. Recent research in provenance explores the use of static analysis and/or dynamic instrumentation techniques in cases where the flow is implicit in programmes or scripts. One motivation is obtaining a provenance abstraction capturing data's flow or the process's dependencies. In noWorkflow [24] authors analyse Python scripts to extract function-call hierarchies, which they use to create a view over provenance traces collected by run-time instrumentation of the functions' reads and writes to the file system. In [25] authors employ a taint tracking framework, which instruments programme executions and records which computations are affected by tainted data sources. Here the authors use names of files read by programmes as taint marks and show how such an approach can create fine grained lineage among files, where the programme during its execution writes to one file the data read from another file. In [26] authors apply source code analysis to programmes that underpin each analytical activity in workflows. The motivation here is obtaining provenance that is of finer-granularity than what is available in standard black-box workflow provenance. Through such analysis authors create a record of prospective provenance where each statement in a programme is modelled as a transformational process and the variables read and updated by the transformation are its inputs and outputs. The focus of these works is having some basic provenance for a system that has not been designed with such features. As the focus is on collecting provenance, these works do not focus how that provenance can be used or what the patterns or anti-patterns in provenance can be. Whereas in our work, we have a provenance capability, and we focus on patterns to increase the fitness of provenance for a particular querying scenario.

Several researchers have previously used Datalog for provenance representation and querying [27] and consistency checking [28]. In the field of workflow verification there exists research on modelling workflows with well-studied process formalisms such as Petri Nets [29] or π-calculus [30] and using temporal-logic based analyses over these process representations. We have opted for Datalog for modelling analysis rules as the broken factorial design pattern we were trying to detect does not have a temporal dimension.

A technique used in optimising compilers, named data-flow analysis [31], is method-wise similar to our approach. Dataflow analysis is a systematic way of collecting information about a program's states (possible variable values) for distinct points in the control-flow graph. Compilers utilise a number of well-know data-flow analysis types such as "Reaching definitions" or "live variable analysis". Here programme statements are associated with transfer functions that encapsulate if/how the statement alters programmes state information (e.g. variable definitions). A *join* operation is defined to compute state, when multiple programme branches come together. The outcome of data-flow analysis is used to optimise compilation, as in dead-code elimination, which excludes from compilation those statements that assign to a variable, which is never read afterwards.

8. Conclusion

In this paper we focused on a particular pattern that prevents workflow provenance from being useful for post execution activities, such as result comparison, parameter calibration, all of which rely on discrete traceability in provenance among parameters and corresponding results. Using Taverna as an example we showed workflow systems can sweep analyses over parameter/data collections using constructs for Factorial Design (FD). We observed, however, that having these constructs is not a guarantee for having parameter-to-result traceability in provenance. We identified the underlying reasons as the following:

1. granularity discrepancies between workflows and provenance in modelling parameters/data,
2. workflow design practices that may lead to broken parameter-to-result traceability in provenance.

In order to understand the level of support that workflow systems provide for factorial design and their modelling granularities, we provided a survey. We observed that workflow systems range over a spectrum in their support. There are systems that do not support FD constructs, thereby mandating scientists to handle the parameter sweeps and associated result organisation through external means [18]. There are those providing FD constructs yet fall short of modelling data and parameters with fine-granularity in provenance, thereby hampering its exploitation [3]. Finally, there are systems that support both the constructs and their fine-grained modelling in provenance, however they are prone to the broken factorial design problem [5] [17].

Due to black-box activities workflow provenance provides no additional information to add resolution to an existing activity execution trace. We therefore adopted a prehoc preventive approach to tackle the challenge of broken traceability. We provided a rule-based a static analysis technique operating over the Taverna language to detect whether a particular workflow design leads to broken traceability in provenance. Through an exposition of Taverna's operation we showed that that if Taverna iterates over a collection of input items, it creates a corresponding output collection and that this correspondence between collections is an assurance of discrete traceability among parameter and result collections.

A notable aspect of our approach is that it anticipates an *n-by-m* or *n-by-*1 pattern globally, at the workflow level, rather than locally at the individual activity level. If we are to inspect an individual activity in provenance by looking at the immediate inputs/outputs it may as well exhibit *n-by-m* . On the other hand such an activity does not pose a threat for factorial design if all its *n* inputs descend from the same workflow input parameter (factor).

Furthermore, broken factorial design may not always constitute a problem to be fixed. The *Flatten_List* processor in our case study workflow, which presented the break point (in factorial design) is a commonly used data adapter step in Taverna workflows. The nested collections created through iterations are represented as nested folder structures in the file system, Taverna's data storage layer. In order to reduce the complexity of this physical representation, scientists often create coarser grained data items by flattening nested lists into a single list or a single string to facilitate easier integration of data into textual reports or experimental bundles. In cases where the flattening is a terminal/final task in a workflow, which is intended for reporting, it would not have a derogatory affect on provenance queries. On the other hand, as discussed in our survey, such reporting tasks obfuscate the analytical process in the workflow.

Our work has shown that workflow design processes need to become provenance-aware, if provenance is later to be exploited for result access. As we're positioning our work in the workflow design process it relates to other design-focused techniques for collecting provenance. PrIME [32] is a methodology for "adapting" existing applications to make them provenance aware. This "adaptation" involves identification of input/output data of interest and possible provenance inquiries over that data, followed by iterative exposure and documentation of an application's actors (sub-modules) and their data interactions with "interaction graphs" (similar to workflows). Application modules identified in interaction graphs are then refactored (wrapped) to collect provenance from their executions. The PrIME methodology is concerned with revealing main data derivation paths in an application, and the granularity of exposure is determined with the requirements of provenance queries. On the other hand, PrIME does not specifically address collection-typed data and the fine-grained lineage among collections. Our work can be considered as complementary to PrIME as it focuses on the aspect of fine-grained provenance.

9. Future Work

We have identified two immediate directions for future work.

First one is the exploration of how the static analysis can assist the workflow design process. We plan to investigate how analysis outcomes can turn into design feedback as 1) highlights on problematic parts of the workflow and 2) suggested fixes. Our approach would be informed by empirical realities of workflows: how often does broken factorial design occur in workflows and whether it is the result of an intentional decision or an accidental design mishap. Note that suggestions of fixes would require more information than what is available in the workflow description. Consider the Astronomy workflow given earlier, where list flattening, which is a data adapter step was causing the break in factorial design. By augmenting the workflow description with semantic descriptions denoting the types of the inputs and outputs of activities, one may infer the input and output types of adapter steps, due to composition [33], and one can suggest that a step such as list flattening, which consumes and produces same typed data, while on the other hand breaks factorial design, may as well be eliminated.

The analysis rules presented in this paper are ultimately specific to Taverna. The second direction of future work is to understand 1) to what extent our work can be applied to other systems and 2) whether there can be alternative system-independent approaches. To proactively prepare for wider applicability, we designed our rules to operate over Wfdesc [8], an abstract model of workflows with extensions capturing Taverna's iteration features. At the time of writing of this paper, the Wfdesc model is being taken as a base for the development of a Common Workflow Language (CWL) [34] for the bioinformatics domain. Once the CWL specifications are released, the immediate next step for us would be to check whether our rules can be adapted to CWL, which potentially would have bilateral mappings with languages of different systems including Taverna and Galaxy. In our approach we started with information about a particular system's workflow specification language and execution semantics to infer resulting provenance patterns. An alternative approach could be taking the reverse direction. Starting with desired provenance patterns and deducing the workflow that would generate those patterns. This would require the specification of anticipated provenance patterns, a thread of research which is currently being explored with two research groups [12] [11]. Here a standard provenance language such as PROV is being enriched with variables to represent patterns abstracted from any particular data or process identity. There are already common patterns for describing scientific experiments in some domains (e.g. "crossover" or "parallel group" designs in "omics-based" studies [35]) or generic patterns such as the parameter-to-result traceability identified in this paper. In case such patterns can be represented as abstract provenance graphs then (workflow) system specific techniques can be devised in order to map patterns to workflows that can generate such patterns. We shall note that the current specifications for abstract provenance [11] [12] are still under development as this research is very recent. We conjecture that abstract provenance representations would potentially require extending with models to represent data collections and constraint languages to represent patterns, such as traceability among collections, or patterns that depend on data values.

[1] S. B. Davidson, J. Freire, Provenance and Scientific Workflows: Challenges and Opportunities, in: SIGMOD Conference, 2008, pp. 1345–1350. doi:http://doi.acm.org/10.1145/1376616.1376772.

[2] C. Tenopir, S. Allard, et al., Data sharing by scientists: Practices and perceptions, PLoS ONE 6 (6) (2011) e21101. doi:10.1371/journal.pone.0021101.

[3] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. Jones, E. A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the kepler system, Concurrency and Computation: Practice and Experience 18 (10) (2006) 1039–1065.

[4] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, H. T. Vo, Vistrails: Visualization meets data management, in: In ACM SIGMOD, ACM Press, 2006, pp. 745–747.

[5] P. Missier, S. Soiland-Reyes, S. Owen, et al., Taverna, reloaded, in: SSDBM, 2010, pp. 471–481. doi:10.1007/978-3-642-13818-8_33.

[6] S. S. Exposito, Workflow: Calculating the internal extinction with data from leda, myExperiment Repository, http://www.myexperiment.org/workflows/2920/versions/2.html (2012).

[7] A. R. Group, Analysis of the interstellar Medium of Isolated GAlaxies, http://amiga.iaa.es/p/1-homepage.htm (2012).

[8] K. Belhajjame, J. Zhao, D. Garijo, M. Gamble, K. Hettne, R. Palma, E. Mina, O. Corcho, J. M. Gomez-Perez, S. Bechhofer, G. Klyne,

C. Goble, Using a suite of ontologies for preserving workflow-centric research objects, Web Semantics: Science, Services and Agents on the World Wide Web 32 (0) (2015) 16 – 42.

[9] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, et al., The open provenance model core specification (v1.1), Future Gener. Comput. Syst. 27 (6) (2011) 743–756. doi:10.1016/j.future.2010.07.005.

[10] P. Groth, L. M. editors, PROV-Overview: An Overview of the PROV Family of Documents, W3C, http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/ (2013).

[11] D. Michaelides, T. D. Huynh, L. Moreau, PROV-TEMPLATE: A template system for PROV documents, Unofficial draft specification: https://provenance.ecs.soton.ac.uk/prov-template/ (June 2014).

[12] V. Curcin, S. Miles, R. Danger, Y. Chen, R. Bache, A. Taweel, Implementing Interoperable Provenance in Biomedical Research, Future Gener. Comput. Syst. 34 (2014) 1–16. doi:10.1016/j.future.2013.12.001.
URL http://dx.doi.org/10.1016/j.future.2013.12.001

[13] P. Missier, N. W. Paton, K. Belhajjame, Fine-grained and efficient lineage querying of collection-based workflow provenance, in: Proceedings of the 13th International Conference on Extending Database Technology, EDBT '10, ACM, New York, NY, USA, 2010, pp. 299–310. doi:10.1145/1739041.1739079.

[14] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases: The Logical Level, Addison-Wesley, 1995.

[15] N. Leone, G. Pfeifer, W. Faber, The DLV Project - A Disjunctive Datalog System (and more), http://www.dbai.tuwien.ac.at/proj/dlv/.
URL http://www.dbai.tuwien.ac.at/proj/dlv/

[16] P. Alper, Taverna Workflow Analysis with Datalog, Source Code, GitHub repository at: https://github.com/pinarpink/phd-sources/workflow2datalog (Dec. 2015).

[17] Y. Gil, V. Ratnakar, J. Kim, P. A. González-Calero, P. T. Groth, J. Moody, E. Deelman, Wings: Intelligent workflow-based design of computational experiments, IEEE Intelligent Systems 26 (1) (2011) 62–72.

[18] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, P. Shah, et al., Galaxy: A platform for interactive large-scale genome analysis, Genome Research 15 (2005) 1451–1455.

[19] W. A. Najjar, E. A. Lee, G. R. Gao, Advances in the dataflow computational model, Parallel Computing 25 (1314) (1999) 1907 – 1929. doi:http://dx.doi.org/10.1016/S0167-8191(99)00070-8.

[20] I. Altintas, O. Barney, E. Jaeger-Frank, Provenance Collection Support in the Kepler Scientific Workflow System, in: IPAW, 2006, pp. 118–132. doi:http://dx.doi.org/10.1007/11890850_14.

[21] S. Bowers, T. McPhillips, M. Wu, B. Ludäscher, Project Histories: Managing Data Provenance Across Collection-Oriented Scientific Workflow Runs, in: Data Integration in the Life Sciences, Vol. 4544 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 122–138. doi:10.1007/978-3-540-73255-6_12.

[22] D. Garijo, P. Alper, K. Belhajjame, Ó. Corcho, Y. Gil, C. A. Goble, Common motifs in scientific workflows: An empirical analysis, Future Generation Computer Systems 36 (2014) 338–351. doi:10.1016/j.future.2013.09.018.
URL http://dx.doi.org/10.1016/j.future.2013.09.018

[23] S. Dey, S. Koehler, S. Bowers, B. Ludaescher, Computing location-based lineage from workflow specifications to optimize provenance queries, in: Provenance and Annotation of Data and Processes, Vol. 8628 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 180–193. doi:10.1007/978-3-319-16462-5_14.

[24] L. Murta, V. Braganholo, F. Chirigati, D. Koop, J. Freire, noworkflow: Capturing and analyzing provenance of scripts, in: 5th International Provenance and Annotation Workshop (IPAW), Cologne, 2014.

[25] M. Stamatogiannakis, P. T. Groth, H. Bos, Looking inside the black-box: Capturing data provenance using dynamic instrumentation, in: Provenance and Annotation of Data and Processes - 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Revised Selected Papers, 2014, pp. 155–167. doi:10.1007/978-3-319-16462-5_12.

[26] D. Ghoshal, A. Chauhan, B. Plale, Static compiler analysis for workflow provenance, in: Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science, WORKS '13, ACM, New York, NY, USA, 2013, pp. 17–27. doi:10.1145/2534248.2534250.

[27] S. Dey, S. Kohler, S. Bowers, B. Ludäscher, Datalog as a Lingua Franca for Provenance Querying and Reasoning, in: Presented as part of the 4th USENIX Workshop on the Theory and Practice of Provenance, USENIX, Berkeley, CA, 2012.
URL https://www.usenix.org/conference/tapp12/workshop-program/presentation/Dey

[28] P. Missier, K. Belhajjame, A prov encoding for provenance analysis using deductive rules, in: Proceedings of the 4th International Conference on Provenance and Annotation of Data and Processes, IPAW'12, 2012, pp. 67–81. doi:10.1007/978-3-642-34222-6_6.

[29] W. M. P. VAN DER AALST, The application of petri nets to workflow management, Journal of Circuits, Systems and Computers 08 (01) (1998) 21–66. doi:10.1142/S0218126698000043.

[30] V. Curcin, M. M. Ghanem, Y. Guo, Analysing scientific workflows with computational tree logic, Cluster Computing 12 (4) (2009) 399–419. doi:10.1007/s10586-009-0099-6.
URL http://dx.doi.org/10.1007/s10586-009-0099-6

[31] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[32] S. Miles, P. Groth, S. Munroe, L. Moreau, Prime: A methodology for developing provenance-aware applications, ACM Trans. Softw. Eng. Methodol. 20 (3) (2011) 8:1–8:42. doi:10.1145/2000791.2000792.
URL http://doi.acm.org/10.1145/2000791.2000792

[33] K. Belhajjame, S. M. Embury, et al., Automatic annotation of web services based on workflow definitions, in: The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings, 2006, pp. 116–129. doi:10.1007/11926078_9.

[34] P. Amstutz, M. R. Crusoe, N. Tijanic, Common Workflow Language (CWL) Workflow Description, v1.0, https://w3id.org/cwl/v1.0/, v1.0 (2016).

[35] P. Rocca-Serra, S.-A. Sansone, M. Brand, et al., Specification documentation: release candidate 1, ISA-TAB 1.0, http://isatab.sourceforge.net/docs/ISA-TAB_release-candidate-1_v1.0_24nov08.pdf (November 2008).

Appendix A. Datalog Rule Modules

The rules given as part of our predicted provenance model *PP* (Definition 3) map to a number of rules in the Datalog implementation (Given in Table A.1). In the following section we discuss Datalog rules in rule blocks that make-up each module.

Datalog programs are collections of rules, which are Horn clauses or if-then expressions [14]. A Datalog program consists of a finite set clauses of the form:

$$A_0 \quad :- \quad A_1, \quad \ldots \quad , A_m \quad (m \geq 0)$$

where each A_i is a positive atom of the form $r(t_1, \ldots, t_k)$ where each t_i is a variable or a constant. "$: -$" is read as "if". There can be two forms of clauses:

- facts, that correspond to the case when $m = 0$
- rules, that correspond to the case when $m > 0$

A rule is comprised of a head and a body. The RHS of the rule clause is the *body*, the LHS is the *head*. Rule Head is an atom and the body is comprised of the conjunction (AND) of zero or more atoms. The rule implies that atom A_0 is *true* if atoms A_1 to A_k are *true*.

Appendix A.1. Depth Prediction Rules

Depth prediction, the rules of which is given in Table A.2 occurs incrementally, through dataflow links and processors.

Rule Definition in Section 4	Rule Block
7.(Delta Depth Calculation Rules)	DP3
8.(Depth Prediction Rules-Single Input Processor)	LHB1
9.(Depth Prediction Rules-Processor Outputs)	DP4
11.(Depth Definition Rule)	R1
12.(Depth Mapping Rule-Single Input Processor Eval.)	LHB2, R4
15.(Depth Prediction and Mapping Rules-link)	DP1, DP2, R2
20.(Depth Prediction and Mapping Rules-cross product)	LHB1
22.(Depth Prediction and Mapping Rules-dot product)	LHB1
13.(Broken Factorial Design)	R3

Table A.1: Mappings from Abstract Model Rules to Datalog Rule Blocks.

- Rule Block DP-1: As the predicted depths of workflow input ports have been provided as part of the EDB, this will initiate the rules that determine the composition types of links from these input ports to processors' input ports. As per depth prediction rules given earlier (Definition 15), what determines a composition (link) type is the difference between the defined depth of the link's target and the actual depth encountered at the source (i.e. the predicted depth of the source). The facts inferred for our example workflow are depicted in Figure A.1, the predicates *wrapped*, *iterated* and *smooth* are used to assert the composition type of each dataflow link.

- Rule Block DP-2: The kinds of composition for each dataflow link informs what the predicted depth of the target of the dataflow link will be. For iterated and simple composition the predicted depth of the target will equal the predicted depth of source. For wrapped composition a depth adjustment occurs so that the predicted depth of target is equal to the defined depth of target.

- Rule Block DP-3: When we have information on the predicted depths of a processor's input ports, then we can determine per rules outlined earlier (Definition 7) whether an input port is initialised with a single input or a space of inputs. The *deltaDepth* facts deduced for each port of *concat4Str* is given in Figure A.1.

- Rule Block DP-4: In order to make a prediction for the output port of a processor, we need to sum up two pieces of information (recall Definition 9). First is the defined depth of the processor's output port, this information is available in the EDB. The second is the size of the overall input space, this information is produced by rules in the LHB Formula module. There are two rules in Rule Block DP-4 (see Table A.2). One is designed to handle the cases with processors that have inputs, and the other is designed for processors without inputs. For the latter case the defined depth of the output becomes also the predicted depth. Rules in block DP-4 infer the predicted depth for output *outstr* of *concat4Str* as 2 (see Figure A.2) as the defined depth of *outstr* is 0, and the size of input space is 2 (we will present the rules for its calculation next).

Appendix A.2. List Handling Formula Rules

List Handling Behaviour (LHB) Formula rules, which are given in Table A.3 perform two computations, 1) the calculation of the overall size of the input space, and 2) the calculation of the mappings from depths in individual input parameter spaces to the depths in overall input space. We make use of the LHB Formula tree for each processor given in the EDB as guidelines to perform these computations. Equations (20) and

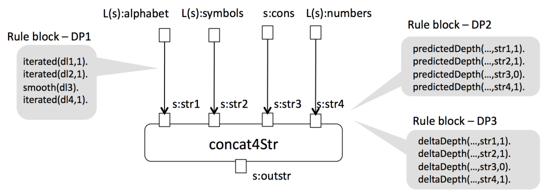


Figure A.1: Illustration of Deductions of the Depth Prediction rules.

DP-1: Determine the kind of composition for data links.

```
wrapped(ID,D):-
dataLink(ID,SRC_PRO, SRC_POR,SNK_PRO, SNK_POR),
predictedDepth(SRC_PRO,SRC_POR,Z),
definedDepth(SNK_PRO,SNK_POR,Y), Z<Y, D=Y-Z.

iterated(ID,D):-
dataLink(ID,SRC_PRO, SRC_POR,SNK_PRO, SNK_POR),
predictedDepth(SRC_PRO,SRC_POR,Z),
definedDepth(SNK_PRO,SNK_POR,Y), Z>Y, D=Z-Y.

smooth(ID):-
dataLink(ID,SRC_PRO, SRC_POR,SNK_PRO, SNK_POR),
predictedDepth(SRC_PRO,SRC_POR,Z),
definedDepth(SNK_PRO,SNKPOR,Y), Z=Y.
```

DP-2: Propagate predicted depth through a dataflow link, there is only an adjustment in the case of wrapped composition.

```
predictedDepth(SNK_PRO, SNK_POR,RES):-
dataLink(ID,SRC_PRO, SRC_POR,SNK_PRO, SNK_POR),
wrapped(ID,D),
predictedDepth(SRC_PRO,SRC_POR,Z),
RES=D+Z.

predictedDepth(SNK_PRO, SNK_POR,Z):-
dataLink(ID,SRC_PRO, SRC_POR,SNK_PRO,SNK_POR),
iterated(ID,_),
predictedDepth(SRC_PRO,SRC_POR,Z).

predictedDepth(SNK_PRO, SNK_POR,Z):-
dataLink(ID,SRC_PRO, SRC_POR,SNK_PRO,SNK_POR),
smooth(ID),
predictedDepth(SRC_PRO,SRC_POR,Z).
```

DP-3: Calculate the delta depth for a port.

```
deltaDepth(PRO, POR,Z):-
definedDepth(PRO, POR,DEFD),
predictedDepth(PRO, POR, PREDD),
Z= PREDD-DEFD.
```

DP-4: Calculate the predicted depths for outputs of an activity using the total size of the input space.

```
predictedDepth(PRO,O_POR,Z):-
hasLhbRoot(PRO,R),sizeCumulative(R,RT),
processOutput(PRO,O_POR),
definedDepth(PRO,O_POR,DD),Z=DD+RT.

predictedDepth(PRO,O_POR,DD):-
hasLhbRoot(PRO,null),
processOutput(PRO,O_POR),
definedDepth(PRO,O_POR,DD).
```

Table A.2: Rules predicting iterations and corresponding depth adjustments.

(22) given earlier for *cross* and *dot* product provide the formulas for what the size of their output will be depending on the size of input. Normally, if we apply these formulas bottom up to the LHB tree, the output size of the top node will be the size of the overall input space. Note that the equations also provide us with a formula to compute mappings from depths of inputs these operators to outputs. In order to compute the input space size and the mappings simultaneously for a formula tree, our rules (Rule Block LHB-1) in Table A.3 prescribe a depth-first pre-order traversal of the LHB tree as illustrated in Figure A.2.

We use two predicates to accumulate the input space size throughout the traversal, these are *sizeLhs* and *sizeCumulative* predicates. *sizeLhs* represents the size of the input space as defined by the portion of the formula to the left of the node. *sizeCumulative* represents the size of lefthand side combined with the size of space of the current node. The first child of each node inherits *sizeLhs* from their parent. Note that the (leaf) port nodes represent an individual input space that is of size *deltaDepth* computed for that port. When *sizeLhs* reaches a port node *sizeCumulative* is inferred by adding *sizeLhs* with the *deltaDepth* for the (port) node.

sizeLhs for a non-first child is computed with information from its left sibling depending on the parent operator node the siblings belong.

- Recall from Definition 20 that the size of output of binary cross product is the sum of the sizes of its inputs. Therefore, in our rules if the parent is a *cross* product then the *sizeCumulative* of the left sibling becomes the *sizeLhs* of the right sibling to reflect the additive nature of *cross* product.

- Recall from Definition 22 that the size of output of binary dot product is equal to the individual size of either input. Therefore in our rules if the parent is a *dot* product then the *sizeLhs* of the left sibling becomes the *sizeLhs* of the right sibling.

Finally the *sizeCumulative* of some parent (operator) is computed when sizes for all its children have been computed. The maximum size computed for a child becomes the *sizeCumulative* of parent. As a result of traversal, the *sizeCumulative* for the top operator node becomes the size of the overall input space.

The *sizeCumulative* that we have computed for each leaf (port) node is also an indicator of its depth mapping. With the rules in Rule Block LHB-2 we make this information more explicit using *depthMapping* predicate. Note that if the *deltaDepth* for a port is zero then the input is comprised of a single value and when a space of tuples is built up from multiple inputs the same value for this input will be used for the entire set of tuples. So this input maps to the entire input space. We denote this asserting that its *depthMapping* is 0. If *deltaDepth* is greater than zero then its dimensions (depths) will map to depths in the input space. The value of *sizeCumulative* for a port node denotes the size of the input space inclusive of that port node exclusive of others that come after it in a formula. Therefore we can deduce that the depth with indice *deltaDepth* for the individual input pace maps to the depth with indice *sizeCumulative* in the overall space. Depth mapping information are used in reaching rules, which we describe next.

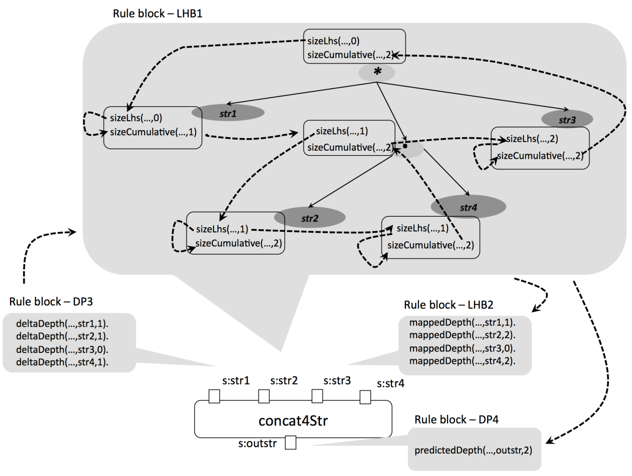


Figure A.2: Illustration of Deductions of the List Handling Formula rules.

Constant definitions.

```
 #const dotnode = dot.  #const crossnode = cross.
```

LHB-1: Computing the overall size of input space.

```
sizeLhs(R,0):-  hasLhbRoot(P,R), R != null.

sizeLhs(FIRSTCHILD,VAL) :-
lhbNode(PARENT,_,_),
hasChild(PARENT, FIRSTCHILD, 0),
sizeLhs(PARENT,VAL).

sizeCumulative(NODEID,Z) :-
lhbNode(NODEID, NAME, PROC),
NAME != dotnode,
NAME != crossnode,
sizeLhs(NODEID,VAL),
deltaDepth(PROC, NAME, NDEL),
Z=NDEL+VAL.

sizeLhs(SIBLING2, VAL):-
lhbNode(PARENT,dotnode,_),
hasChild(PARENT, SIBLING1, N),
hasChild(PARENT, SIBLING2, NEXT),
sizeLhs(SIBLING1,VAL), NEXT= N+1.

sizeLhs(SIBLING2, VAL):-
lhbNode(PARENT,crossnode,_),
hasChild(PARENT, SIBLING1, N),
hasChild(PARENT, SIBLING2, NEXT),
sizeCumulative(SIBLING1,VAL), NEXT= N+1.

sizeCumulative(PARENT, VAL):-
hasChild(PARENT, LASTCHILD, X),
#max{V : hasChild(PARENT,_,V)} =Y,
sizeCumulative(LASTCHILD,VAL), X==Y.
```

LHB-2: Computing depth mappings from each individual input to overall input space.

```
depthMapping(PROC, PORT, 0):-
lhbNode(NODEID,PORT,PROC),
sizeCumulative(NODEID, VAL), deltaDepth(PROC, PORT, 0).

depthMapping(PROC, PORT, VAL):-
lhbNode(NODEID,PORT,PROC), sizeCumulative(NODEID, VAL),
deltaDepth(PROC, PORT, ND), ND>0.
```

Table A.3: Rules calculating the overall size of input space and the depth adjustments per input based on LHB formula.

Appendix A.3. Context and Reaching Rules

We use a predicate named *context* to allow users define experimental contexts in the EDB. A context can be seen as a named depth definition. For our running example let us assume that we define each input item (parameter) in lists of alphabet and symbols to be contexts (Each item is of type *s*, therefore of depth 0). This represented with the ground facts $context(ctxA, w1, alphabet, 0)$. and $context(ctxS, w1, symbols, 0)$.

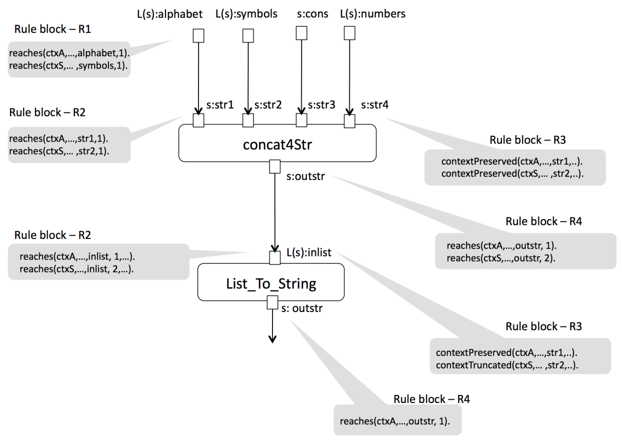

L(s):alphabet
L(s):symbols
s:cons
L(s):numbers
Rule block – R1
reaches(ctxA,...,alphabet,1).
reaches(ctxS,... ,symbols,1).
s:str1
s:str2
s:str3
s:str4
Rule block – R2
reaches(ctxA,...,str1,1).
reaches(ctxS,... ,str2,1).
concat4Str
Rule block – R3
contextPreserved(ctxA,...,str1,..).
contextPreserved(ctxS,... ,str2,..).
s:outstr
Rule block – R4
reaches(ctxA,...,outstr, 1).
reaches(ctxS,...,outstr, 2).
Rule block – R2
reaches(ctxA,...,inlist, 1,...).
reaches(ctxS,...,inlist, 2,...).
L(s):inlist
List_To_String
s: outstr
Rule block – R3
contextPreserved(ctxA,...,str1,..).
contextTruncated(ctxS,... ,str2,..).
Rule block – R4
reaches(ctxA,...,outstr, 1).

Figure A.3: Illustration of Deductions of the Reaching rules.

By building on such definitions the reaching rules presented in Table A.4 can be described as follows:

- Rule Block R-1: We use context definitions to kickstart the computation of the reach of contexts. By default a context reaches the port it is defined on. As illustrated in Figure A.3 context *ctxA* reaches depth indiced 1 at the workflow input port *alphabet* and similarly *ctxS* reaches depth indiced 1 at workflow input port *symbols*.

- Rule Block R-2: We propagate reaching from the source of a dataflow link to its target by taking into account the composition types of links. For simple and iterated composition, reaching propagates to the same indiced depths at the target port. For wrapped composition reaching propagates to depths of target with a positive indices shift equal to the amount of the wrapping adjustment made by the link.

- Rule Block R-3: When a context reaches a processor input port depending on the depth it reaches we can determine whether it will reach processor outputs. Recall that while performing depth prediction we calculated *depthMapping*s for each depth in the input spaces of individual input ports. So at this stage we need to check whether the depth that a context reaches is a depth within the input space. If it is within input space it will be preserved (or mapped to outputs), if it reaches a depth beyond the delta depth it will be truncated i.e. discontinued. In Figure A.3 both *ctxA* and *ctxS* reach respective processor input ports at depth indice 1, note that this is an indice within the boundary of the *deltaDepth* for the input ports, which is also 1, therefore both contexts will be preserved at the processor *concat4Str*, and they will be propagated to this processor's output ports.

- Rule Block R-4: We determine to which depth in a processor's output port a context reaches by using the *depthMapping* information associated with input ports. Note that for the input port *str*1 the *depthMapping* is 1, therefore *ctxA* reaches depth 1 in the port *outStr*. On the other hand the mapping for *str*2 is 2, hence *ctxS* is forwarded to depth 2 in *outStr*. Note now our example contexts are mapped to different depths at the same port.

R-1: Kickstart reaching definition.

```
reaches(CTX,PRO, PORT,Z):-
context(CTX,PRO,PORT,RMPOS),
predictedDepth(PRO,PORT,PRED_DEP),
Z=PRED_DEP - RMPOS.
```

R-2: Propagate the reach of a context through a dataflow link.

```
reaches(CTX,SNKPRO, SNKPOR,Z):-
reaches(CTX, SRCPRO, SRCPOR, Z),
smooth(DL1),
dataLink(DL1, SRCPRO,SRCPOR, SNKPRO, SNKPOR).

reaches(CTX,SNKPRO, SNKPOR,Z):-
reaches(CTX, SRCPRO, SRCPOR, Z),
iterated(DL1,_),
dataLink(DL1, SRCPRO,SRCPOR, SNKPRO, SNKPOR).

reaches(CTX,SNKPRO, SNKPOR,Y):- reaches(CTX, SRCPRO, SRCPOR, Z),
wrapped(DL1,X),
dataLink(DL1, SRCPRO,SRCPOR, SNKPRO, SNKPOR),
Y=X+Z.
```

R-3: Determine whether a context reaching a processor input will reach an output depth or is discontinued.

```
contextTruncated(CTX,PRO, PORT,Z):-
processInput(PRO,PORT),
reaches(CTX, PRO, PORT, CTX_POS),
deltaDepth(PRO, PORT, LMPOS),
CTX_POS > LMPOS, Z= CTX_POS - LMPOS.

contextPreserved(CTX,PRO, PORT,Z):-
processInput(PRO,PORT),
reaches(CTX, PRO, PORT, CTX_POS),
deltaDepth(PRO, PORT, LMPOS),
CTX_POS <= LMPOS, Z= LMPOS - CTX_POS.
```

R-4: Propagate the reach of a context from inputs ports of of an activity to its output ports.

```
reaches(CTX,P1, OUT1,Z):-
processInput(P1,IN1),
processOutput(P1,OUT1),
contextPreserved(CTX,P1,IN1,CDEL),
depthMapping(P1,IN1,FFAC),
Z= FFAC-CDEL.
```

Table A.4: Rules for inferring the reach of a context throughout workflow.

The case when reaching cannot be propagated occurs when a context reaches a depth that is beyond the input space, which denotes that it is not a driver of iteration but a part of data value to be consumed by one invocation of processor. At the *List_To_String* step in Figure A.3 *ctxS* reaches depth 2 of the input port. Meanwhile *deltaDepth* for this port is 1, in other words *List_To_String* will iterate over a list inputs it can consume. As a result *ctxS* get truncated at the *List_To_String* step.

Appendix B. Overview of Functions Used

Function	Description
(1) eval	Recursive evaluator that applies a designated named function (for a processor) over string values in a designated (nested) list structure, where strings represent tuples of processor inputs.
(2) link	Performs a wrapping adjustment on a given data value depending whether the data has the designated nesting level.
(3) init	Replaces data items that are of a designated nesting level in a designated input list with their string-based representation.
(4) cross	Creates a cartesian product of two designated (nested) lists by creating tuples out of string items in lists.
(5) dot	Creates a zip product of two designated (nested) lists by creating tuples out of same indiced string items in lists.

Table B.1: Brief descriptions of key functions that make up Taverna's execution behaviour.