Provenance-Based Reproducibility in the Semantic Web

Luc Moreau

Electronics and Computer Science,
University of Southampton, Southampton, SO17 1BJ UK

Abstract

Reproducibility is a crucial property of data since it allows users to understand and verify how data was derived, and therefore allows them to put their trust in such data. Reproducibility is essential for science, because the reproducibility of experimental results is a tenet of the scientific method, but reproducibility is also beneficial in many other fields, including automated decision making, visualization, and automated data feeds. To achieve the vision of reproducibility, the workflow-based community has strongly advocated the use of provenance as an underpinning mechanism for reproducibility, since a rich representation of provenance allows steps to be reproduced and all intermediary and final results checked and validated. Concurrently, multiple ontology-based representations of provenance have been devised, to be able to describe past computations, uniformly across a variety of technologies. However, such Semantic Web representations of provenance *do not have any formal link* with execution. Even assuming a faithful and non-malicious environment, *how can we claim that an ontology-based representation of provenance enables reproducibility, since it has not been given any execution semantics*, and therefore has no formal way of expressing the reproduction of computations? This is the problem that this paper tackles by defining a denotational semantics for the Open Provenance Model, which is referred to as the *reproducibility semantics*. This semantics is used to implement a *reproducibility service*, leveraging multiple Semantic Web technologies, and offering a variety of reproducibility approaches, found in the literature. A series of empirical experiments were designed to exhibit the range of reproducibility capabilities of our approach; in particular, we demonstrate the ability to reproduce computations involving multiple technologies, as is commonly found on the Web.

Keywords: provenance, reproducibility, denotational semantics, primitive environment

1. Introduction

The envisaged applications of science and technology are far reaching, from Government personalised services for the citizen[1] to personalised medicine[2], from understanding climate change, to its tackling by smart energy usage [1]. Technology is revolutionising the way scientists undertake science, as illustrated by Grid Computing [2], e-Science [3], or the Fourth Paradigm [4]. While science is becoming computation and data intensive, the fundamental tenet of the scientific method remains unchanged: experimental results need to be *reproducible* [5].

The fundamental principle of reproducibility is particularly important given the importance of science in our life. This importance is illustrated by the scientific advice on climate change that has helped shape governmental policies. The mail controversy of the Climatic Research Unit "Climate-Gate"[3] highlights how global the impact of science has become. As a result, calls for more transparency in climate science have been issued; specifically, a parliament committee called for the release of both the raw data and the computer code used in research[4]. This is in no way limited to climate science: in social science, evidence-based policy refers to public policy that is informed by rigorously established objective evidence [6]. Similar calls exist for transparency in clinical trial results used in drug approval[5]. It is no surprise that novel initiatives [7] and recommendations [8] that encourage the publication of data sets are emerging. These are steps in the right direction, but by themselves, they do not ensure reproducibility of scientific results.

Provenance refers to the source or origin of something. In a computational setting, provenance of a data item is an explicit representation of the processes that led to that data item [9]. The workflow-based scientific community has strongly advocated the use of provenance as an underpinning mechanism for reproducibility: "*reproducibility requires rich provenance information, so that researchers can repeat techniques and analysis methods to obtain scientifically similar results . . . In order to support reproducibility, workflow management systems must capture and generate provenance information as a critical part of the workflow-generated data.*"[10]. The strong belief that

Email address: l.moreau@ecs.soton.ac.uk (Luc Moreau)

[1] http://www.telegraph.co.uk/technology/news/7484600/Every-citizen-to-have-personal-webpage.html

[2] http://www.futuremedicine.com/doi/abs/10.2217/17410541.5.1.55

[3] http://en.wikipedia.org/wiki/Climatic_Research_Unit_email_controversy

[4] http://blogs.nature.com/news/thegreatbeyond/2010/03/parliament_committee_calls_for.html

[5] http://www.sciencebasedmedicine.org/?p=215

provenance can support reproducibility is also echoed by the provenance community, with more than twenty papers cited by a recent survey on provenance [11] mentioning reproducibility in their abstract.

To relate reproducibility to provenance, various approaches have emerged in which provenance is defined in the context of specific execution semantics. Why-provenance [12] and lineage [13] identify source tuples that "contributed" to a result returned by a database query (within the relational and xml models). By applying the same query to such source tuples, the result could be reproduced. Souilah *et al.* [14] define a denotation of provenance in the context of a π-calculus variant: this denotation can replay the sending and receiving of data values across processes. Cheney *et al.* [15] define a notion of traces faithful to a program if they record enough information to recompute the program when the inputs change; this definition is proposed in the context of the Nested Relational Calculus, which is a core database query language, also capturing aspects of functional languages, and distributed programming systems such as MapReduce [16]. Similarly, Acar *et al.* [17] assume the existence of "a common language that can express both database queries and workflows", and in this context, define a provenance graph to be consistent with a program if it matches the evaluation traces generated by the program. In workflow systems, the Virtual Data Systems [18] views provenance as consisting of two components: all the aspects of the procedure or workflow for creating a data object (referred to as prospective provenance), and the information about the runtime environment in which these procedures were executed (retrospective provenance), the combination of both offering reproducibility. By means of a translation from provenance to its workflow language [19], Taverna is also able to reproduce past results.

All of these approaches have in common that they make strong assumptions on the execution environment(s) in which the application is executed: they assume it is a given database engine, workflow system, or distributed programming environment. But such assumptions are not aligned with the reality of Web applications, which typically involve multiple different technologies, with different underpinning semantics, hosted by different providers.

In a previous paper [11], the author articulated the Open Provenance Vision, consisting of architectural guidelines to support provenance inter-operability on the Web, by means of open models, open serialization formats and open APIs. As envisaged in the Open Provenance Vision, the provenance from individual systems or components can be expressed, connected in a coherent fashion, and queried seamlessly. Several models for provenance have emerged to tackle this vision, including Provenir [20, 21], the Provenance Vocabulary [22], PASOA [23], OPMV [24], PML [25] and the Open Provenance Model [26]. All rely on Semantic Web definitions, consisting of ontologies, vocabularies or abstract models. Assuming that provenance based on these models was generated faithfully and non-maliciously, *how can it be claimed that provenance enables reproducibility, given that there is no execution semantics attached to these models?* This is precisely the problem that this paper tackles, offering a definition of reproducibility for the Open Provenance Model (OPM) [26].

Several provenance approaches (by Cheney *et al.* [15], Souilah *et al.* [14], Acar *et al.* [17], and Missier and Goble [19]) have in common the idea that provenance can be seen as a program, for which an executable semantics can be defined. Hence, using provenance as a program, one can re-execute past computations, and reproduce results. In this paper, we leverage this idea and formalize it for OPM, while still ensuring that OPM remains independent of any technology used in the application execution environment.

To address this problem, this paper offers the following contributions:

1. Adopting a Semantic Web perspective to the problem, we extend the Open Provenance Model with minimum executional information and assumptions related to execution. In particular, we introduce the class of primitive procedures and the notion of primitive environment that maps such primitive procedures to something that can be executed. OPM processes are themselves caused by primitive procedure invocations.
2. We present the reproducibility semantics, a denotational semantics for OPM graphs. This mathematical (and therefore technology independent) definition formulates how an OPM graph can be seen as a mathematical function, taking some inputs and a primitive environment, and resulting in another OPM graph. This semantics is novel because it tackles a substantial subset of OPM, and in particular, its notion of account. With this semantics, we also identify a class of OPM graphs that are reproducible, and at the same time recognize that not all OPM graphs are by default reproducible. Specifically, we define a provenance graph as "reproducible", if combined with a primitive environment, it contains enough information to be interpreted as a program (or workflow) whose execution can yield an isomorphic provenance graph. Furthermore, this mathematical formulation also allows us to specify variants of reproducibility, which help us provide formal groundings for other reproducibility proposals found in the literature.
3. We propose a Semantic Web based architecture for a *reproducibility service*, which can take OPM graphs and check their reproducibility. This architecture, which relies on an extended OWL ontology for OPM, SWRL rules, and a set of SPARQL queries, is intended to act as a reference implementation for the reproducibility service.
4. We provide an evaluation of the approach by demonstrating: *(i)* how the reproducibility service is capable of reproducing the results of the first provenance challenge [27]; *(ii)* how the reproducibility service can easily be customized to invoke multiple execution technologies, such as command line and web services, simply by changing its primitive environment; *(iii)* how it can be used with different inputs.

This work is significant for several reasons. First, Semantic Web provenance languages are defined in terms of an ontology, but do not have an execution semantics; with this work, we establish that such provenance data models can and should also

be interpreted as executable programs. Second, from a provenance perspective, approaches such as OPM have been criticised as lacking formal foundations, and therefore, leading to incompatibility and misinterpretation [28]; the formal semantics presented in this paper addresses these concerns head on. Third, OPM, itself, contains original features, such as accounts, which have never been formalised. This paper provides an original way of formalizing them; in particular, it identifies a set of conditions required to be met for provenance claims in different accounts to be consistent.

There is no strong consensus on what reproducibility means, and how it can be achieved. Hence, we start by surveying related work, by providing several definitions of reproducibility, and explain how this work compares against them (Section 3). We then introduce the reproducibility semantics in Section 4, first considering account-less OPM graphs, and then multi-account OPM graphs. Next, we study the idea of a reproducibility service in the context of the Semantic Web, overview its architecture, and discuss the Semantic Web techniques that we leveraged for its implementation (Section 5). We then undertake a range of empirical evaluations aiming to demonstrate the capability of the reproducibility service, and its suitability for deployment in a multi-technology environment such as the Web (Section 6). This is followed by a discussion of the approach (Section 7) before we conclude the paper and summarize possible future work (Section 8). Beforehand, we summarize the OPM terminology.

2. OPM Terminology in One Paragraph and Figure

We assume that the reader is familiar with the OPM specification [26]; a tutorial on OPM is also available from `openprovenance.org/tutorial`. The following example acts as a reminder for the OPM terminology. Figure 1 illustrates an OPM graph describing the evaluation of a numeric expression $(10+20)\times 30/9$ resulting in value 100. Ovals represent artifacts and are here associated with numeric values; black rectangles denote processes. Plain edges represent data derivations (referred to as was-derived-from dependencies); dotted edges represent the dependencies between processes and artifacts, denoting the consumption of the latter by the former (used edges) or the generation of the latter by the former (was-generated-by edges); they are annotated by their roles in bracket. Gray "post-it" rectangles are annotations. The OPM specification [26] also introduces a notion of inference by which novel edges can be inferred from existing edges of an OPM graph. For instance, p_3 used a_6, which itself was derived from a_5, itself also derived from a_1 and a_2. So, we can infer that p_3 used a_1 and a_2 indirectly. Likewise, it can be inferred that a_6 was derived from a_1 and a_2 indirectly. Finally, OPM edges that have not been inferred are said to be asserted.

3. Related Work

In this section, we first review several definitions of reproducibility. We then discuss work on reproducibility that is not provenance specific, before focusing on provenance based approaches. Finally, given that our work consists of a novel formalization of OPM, we review extant efforts in that field.

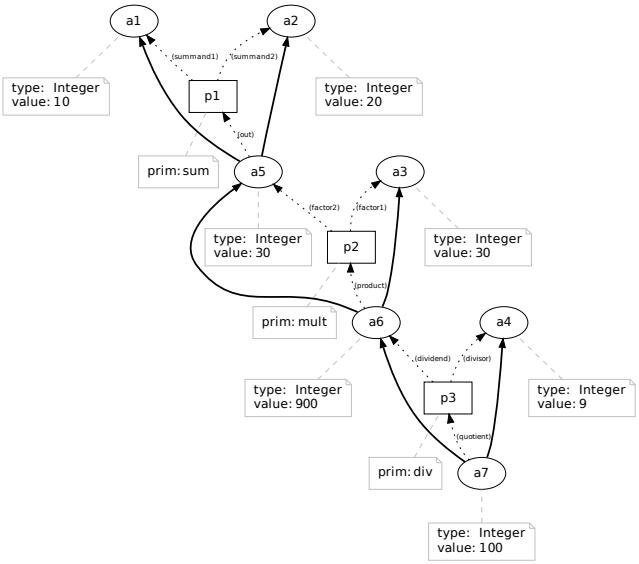


Figure 1: OPM graph for Numeric Expression

3.1. What is reproducibility?

There is no strong consensus on what reproducibility means in the context of computational science or computer-based systems. Wikipedia defines reproducibility[6] generally as follows: "*Reproducibility* is one of the main principles of the scientific method, and refers to the ability of a test or experiment to be accurately reproduced, or replicated, by someone else working independently to see if the reproduced experiments gives similar results to those originally reported". Wikipedia further contrasts reproducibility from *repeatability*[7], which measures the success rate in successive experiments, possibly conducted by the same experimenters. Reproducibility relates to the agreement of test results with different operators, test apparatus, and laboratory locations.

In computer systems, experiments are encoded as programs or workflows [10] that, like recipes, describe the various steps of execution, and can be executed time and time again. In computer systems, however, extensive logs of past activities, which we will refer to as provenance, can provide an accurate description of what occurred in the past, and can be used to reproduce experiments: the difference is that reproduction can be based on the logs, rather than the recipe.

Bechhofer *et al.* [29] see the need for a framework that facilitates the reuse and exchange of digital knowledge. They put

[6] `http://en.wikipedia.org/wiki/Reproducibility`
[7] `http://en.wikipedia.org/wiki/Repeatability`

forward the idea of Research Objects as containers for a principled aggregation of resources, produced and consumed by common services and shareable between scientists. In this context, they distinguish the following terms:

- *Repeatability*: relies on sufficient information for the original researchers or others to be able to repeat the study. This may involve access to data or execution of services.
- *Reproducibility* of a result consists of starting with the same materials and methods and checking if a prior result can be confirmed. It is a special case of repeatability, since it contains complete information such that a final or intermediate result can be verified.
- *Replayability* allows the investigator to "go back and see what happened". It does not necessarily involve execution or enactment of processes and services. It places a requirement on provenance of data.

From the above definitions, it is not entirely clear whether Bechhofer's repeatability and reproducibility draw on the original recipe or provenance of a past execution, or a combination of both. On the other hand, replayability seems to rely explicitly on provenance.

In their classification of provenance requirements, Miles *et al.* [30] identify a use case (Use Case 17) that distinguishes *reenactment*, i.e. performing the same experiment, but using contemporary data and services, from *repetition*, which means performing the same experiment with the same data and services as before, e.g. to test that the results can be reproduced. Whilst framed in the context of provenance, reenactment seems to apply equally to provenance and workflows.

So, reproducibility is a multi-dimensional problem, where several issues need to be taken into consideration: *(i) which scripts?*: is this workflow or provenance based reproducibility? *(ii) which inputs?*: is the experiment reproduced with the same inputs or others? (by inputs, we include not only experimental data, but also parameters) *(iii) which primitives?*: are the original primitives or services invoked? *(iv) which results?*: are intermediary and final results comparable to the original ones? In this paper, we fix the first dimension, focusing on provenance-based reproducibility, while considering all other dimensions of the problem.

3.2. Reproducibility without Provenance

Reproducibility has initially been researched without taking provenance into consideration. We review some salient outcomes, before focusing on provenance-based reproducibility.

Claerbout pioneered the concept of *really reproducible research*, " *An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.*" [31].

This pioneering approach has led to the more recent ReDoc [32], a system for reproducing scientific computations in electronic documents. It consists of three components, makefiles, make rules and naming conventions. Such an environment is akin to a workflow system, where the workflow script is a makefile, which can be executed over any input. It however does not describe a past execution and how a past result was achieved. But this shows that under the term "reproducibility", one can find two very different understandings: regenerate a result by applying a recipe on arbitrary inputs vs reproduce all the steps found in an evidence of a past execution.

In forensic investigations, a key factor is reproducibility, defined as the ability to achieve a consistent level of quality throughout the investigative process, no matter how many times it is repeated under the same conditions [33]. Pan and Batten [33] propose a model based on read and write operations, and associated timestamps, allowing them to be ordered in a linear time flow. This model is targeted to forensic investigations and is not aimed to generic computations. An alternative approach, which is provenance-based, is proposed by Levine and Liberatore [34] and discussed in Section 3.3.

Similar ideas, though not referred to as reproducibility, have already been developed in the context of digital preservation formalization [35]; for instance, rerunning past programs over past data over different machines, with potentially different hardware, can be achieved by means of emulators.

Stodden [36] defines reproducibility as the ability of others to recreate and verify computational results, given appropriate software and computing resources. Stodden investigates the legal impediment to scientific reproducibility, and proposes a reproducible research standard. We do not denigrate the importance of legal issues, but this article only focuses on the technical aspects of reproducibility.

3.3. Provenance-based Reproducibility

Davidson and Freire explain that a key benefit for maintaining provenance of computational results is reproducibility: a detailed record of the steps followed to produce a result allows others to reproduce and validate these results [37]. Specifically, with an explicit representation of provenance, so-called provenance queries can be expressed to identify all the data objects and the sequence of steps that have been used to produce a result [38].

Levine and Liberatore [34] seek to improve the reproducibility and comparison of digital forensic evidence. They propose a simple canonical description of digital evidence provenance that explicitly states the set of tools and transformations that led from acquired raw data to the resulting product. This provenance representation allows for the comparison and the reproduction of results. Inspired by the principle of N-version programming, their approach allows multiple tools/libraries to be used to reproduce a result, thereby increasing the confidence that can be put in investigations.

Silva *et al.* [39] argue about the importance of reproducibility in visualization. Leveraging the Vistrails systems, they exploit the provenance it generates to ensure that users will be able to reproduce the visualizations and let them easily navigate through the space of visualization pipelines created for a

given exploration task. Vistrails adopts an action-based provenance model, intended to help reproducibility. We conjecture that the reproducibility capability is based on replaying such actions, but the authors do not include an explicit description of how such functionality can be achieved outside the context of Vistrails itself. Koop *et al.* [40] discuss the problem of managing upgrades of tools and libraries, while still being able to run a previous computation in a new environment; this work is specifically focused on the Vistrails system and methods necessary to automatically update workflows and provenance.

Mesirov [5] proposes a Reproducible Research System (RRS), consisting of two components. The first is the Reproducible Research Environment (RRE), which provides computational tools together with the ability to automatically track the provenance of data, analyses, and results and to package them (or pointers to persistent versions of them) for redistribution. The second element is a Reproducible Research Publisher (RRP), which is a document-preparation system, such as standard word-processing software, that provides an easy link to the RRE.

Cheney *et al.* [15] provide an operational semantics for the Nested Relational Calculus (NRC) that generate *traces*, intented to capture the execution history of a query. Such a notion of trace is a representation of provenance which is directly inspired by the syntax of NRC constructs. They define properties of such traces and NRC programs, such as consistency and fidelity. A trace is said to be consistent to a program, if it is an explanation of what happened when the program was evaluated. Fidelity is the property that holds when the trace records enough information to recompute a program when the inputs change.

The provenance approaches that have been reviewed in this section are all grounded in a specific execution environment. Hence, because of their dependencies to a specific technology or execution semantics, they fail to meet a key requirement of the Open Provenance Vision [11] for provenance on the Web. Alternative provenance definitions are ontology-based and not specific to an execution technology: Provenir [20, 21], the Provenance Vocabulary [22], PASOA [23], OPMV [24], PML [25], and the Open Provenance Model [26]. None of them, however, has been given a semantics that is suitable for reproducibility purpose. This is a shortcoming that we address for OPM in this paper. This work complements other endeavours aiming to provide a formal underpinning to OPM, which we survey in the following section.

3.4. Formal definitions of OPM

Moreau *et al.* [41] provide a set-theoretic definition of OPM, as well as an illustration of how an abstract machine execution can generate OPM-based provenance traces. The mapping from execution to OPM graphs is not formally characterized, and no attempt is made to provide a converse mapping. In the subsequent version, Kwasnikowska *et al.* [42] provide a temporal interpretation of OPM graphs, defined as the set of temporal inequalities implied by its edges. OPM inferences combined with graph patterns are shown to be sound and complete with respect to inferences that can be made over graph interpretations. This temporal interpretation does not provide an understanding of OPM from an execution perspective either.

Cheney [43] investigates the use of structural causal models as a semantics for provenance graphs, and relates some OPM concepts to notions of actual cause and explanation proposed by Halpern and Pearl [44, 45]. At some level, the semantics we propose here bears some similarity to Cheney's since it is also a denotation of a provenance graph, i.e., it sees a graph as a mathematical function, resulting in a new provenance graph. In practice, they differ for several reasons: *(i)* our semantics conforms to OPM v1.1 and in particular handles OPM accounts, whereas Cheney's is account-less and regards single-step derivation edges as inferrable, when they can only be asserted in OPM; *(ii)* our semantics builds on the Semantic Web philosophy, where globally unique names are meant to capture well understood concepts (in this case, primitives), which are explicitly captured within a notion of primitive environment; *(iii)* Cheney's semantics attempts the more ambitious goal of providing a global approximation (using the predictive nature of causal models) for the program being executed (without having its explicit code), so that its behaviour can be repeated for any arbitrary input; *(iv)* our semantics allows us to explore and characterize variants of reproducibility.

Missier and Goble [19] address the question of whether, for any OPM graph, there exists a plausible workflow in the Taverna workflow language, which could have generated the graph. To this end, they identify the extra information that should be captured as part of an OPM graph so that the mapping from OPM to a workflow representation can be derived. Whilst this work focuses on some specificities of the Taverna workflow language, such as the implicit iterator semantics, it is similar in spirit to ours, since it derives an executable semantics for OPM. In their case, it is obtained by composing their translation to the Taverna semantics [46]. It however does not tackle OPM in full, ignoring accounts, and does not define reproducibility itself.

Various ontological definitions of the Open Provenance Model have emerged. The OPM toolbox[8] supports bidirectional conversions between XML and RDF serializations, respectively defined according to an XML schema and an OWL ontology [47]. This ontology was inspired by the OWL definition compatible with the OPM implementation of Tupelo[9] [48]. During the third provenance challenge, the Tetherless team also defined an OPM OWL ontology[10].

As part of the W3C Provenance Incubator activity [49], mappings of multiple provenance ontologies to OPM were defined [50]. These mappings showed that concepts such as processes, artifacts, and agents can be mapped quite naturally between the models. The mappings however do not characterize the computational implications associated with those models, and do establish whether reproducibility is preserved during translation across models.

[8] http://openprovenance.org

[9] http://twiki.ipaw.info/pub/Challenge/OpenProvenanceModelBindings/opm.owl

[10] http://twiki.ipaw.info/bin/view/Challenge/TetherlessPC3

4. Reproducibility Semantics

In this section, we specify the *reproducibility semantics* for OPM graphs, which we define as the mathematical meaning of an OPM graph, seen as a program and whose execution results in a new OPM graph. First, we study reproducibility in account-less OPM graphs (Sections 4.2, 4.3 and 4.4) and then in multi-account graphs (Sections 4.5 and 4.6).

4.1. Intuition of the Reproducibility Semantics

Before delving into the technical details of a denotational semantics, we provide some intuition of how we propose to reproduce the execution on an OPM graph. We assume that each process in an OPM graph is annotated with the name of a primitive, and that there is a primitive environment that maps primitive names to actual functions, which in a first approximation take some inputs and produce some outputs. Here, function arguments are not identified by their position in a sequence of arguments but by their roles; likewise, outputs can be multiple and are identified by their role.

The inputs of an OPM graph are all the artifacts for which there is no was-generated-by edge; its outputs are all the artifacts that are not adjacent to a used edge; intermediary artifacts are those that are not inputs or outputs. Given an acyclic OPM graph, we assume the existence of a function that returns a list of all its processes, sorted by order of execution: by this, we mean that a process in the sorted list does not use any artifact generated by a process that is subsequent in the list.

Reproducibility is formalized by a recursive function that traverses the sorted list of processes, executing each of them in turn. Execution of a process is achieved by invoking its associated primitive function on the values of artifacts (paired with roles) it uses, and results in values (also paired with roles), for which new artifacts are created. The reproducibility function constructs a new OPM graph at the same time, describing the re-execution of the graph. The new graph is initialized with the set of input artifacts (with the same value as in the original graphs, or different values, depending on the kind of reproducibility one wants to achieve). Each process execution adds the process and the output artifacts it generated, and all associated edges with roles, where appropriate. Each function of the primitive environment not only results in artifacts for given inputs, but also in a set of was-derived-from edges, which are added to the newly produced graph.

For comparing the original graph and the new graph, but also for book-keeping, a mapping from nodes of the original graph to those of the new graph is constructed. It allows us to check whether all nodes have been mapped, and whether they have the same associated values. Furthermore, in the case of multi-account OPM graphs, the mapping allows us to decide whether we are processing a node that has already been encountered.

So, in summary, the reproducibility semantics is expressed by a function that takes a set of input artifacts (with their associated values), a primitive environment, and an input graph, and returns a mapping and a new OPM graph, which describes the reexecution of the input graph, and a mapping. We now formalize this semantics.

4.2. Preliminary Definitions

First, an account-less OPM graph is defined in terms of its constituents in Figure 2, before its semantics is formalized in Section 4.3. An account-less OPM graph consists of a set of nodes and a set of edges. Nodes can be artifacts or processes[11], whereas edges are of four permitted types. Artifacts and processes respectively belong to primitive sets *Artifact* and *Process*. Figure 2 and following figures are a stylised representation of a Standard ML encoding of the reproducibility function available for download[12].

$$
\begin{array}{rcl}
\mathit{Artifact} & = & \text{primitive set of artifacts} \\
\mathit{Process} & = & \text{primitive set of processes} \\
\mathit{Role} & = & \text{primitive set of roles} \\
\mathit{Value} & = & \text{primitive set of values} \\
\mathit{PrimitiveName} & = & \text{primitive set of primitive names} \\
\mathit{Node} & = & \texttt{art of } \mathit{Artifact} \mid \texttt{proc of } \mathit{Process} \\
\mathit{Edge} & = & \texttt{used of } (\mathit{Process} \times \mathit{Role} \times \mathit{Artifact}) \\
 & & \mid \texttt{wgb of } (\mathit{Artifact} \times \mathit{Role} \times \mathit{Process}) \\
 & & \mid \texttt{wdf of } (\mathit{Artifact} \times \mathit{Artifact}) \\
 & & \mid \texttt{wtb of } (\mathit{Process} \times \mathit{Process}) \\
\mathit{OPMGraph} & = & \mathbb{P}(\mathit{Node}) \times \mathbb{P}(\mathit{Edge}) \\
\mathit{AResolver} & = & \mathit{Artifact} \rightarrow \mathit{Value} \\
\mathit{PResolver} & = & \mathit{Process} \rightarrow \mathit{PrimitiveName} \\
\mathit{FullOPMGraph} & = & \mathit{OPMGraph} \times \mathit{AResolver} \times \mathit{PResolver}
\end{array}
$$

Figure 2: Account-Less OPM Graph

Artifacts are associated with values (belonging to a primitive set of values), whereas each process is associated with the name of a primitive, whose invocation resulted in this process. The association is defined by the mappings *AResolver* and *PResolver*. A *FullOPMGraph* then refers to an *OPMGraph* accompanied by the artifact and the process resolvers.

The OPM graph of Figure 1 can be formalized by $G^v = (G, \mathcal{V}^a, \mathcal{V}^p)$ as follows:

$G = \langle \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, p_1, p_2, p_3\}$, $\{\texttt{used}(p_1, \mathsf{summand1}, a_1)$, $\texttt{used}(p_1, \mathsf{summand2}, a_2)$, $\texttt{wgb}(a_5, \mathsf{out}, p_1)$, $\texttt{used}(p_2, \mathsf{factor1}, a_3)$, $\texttt{used}(p_2, \mathsf{factor2}, a_5)$, $\texttt{wgb}(a_6, \mathsf{product}, p_2)$, $\texttt{used}(p_3, \mathsf{divisor}, a_4)$, $\texttt{used}(p_3, \mathsf{dividend}, a_6)$, $\texttt{wgb}(a_7, \mathsf{quotient}, p_3)$, $\texttt{wdf}(a_5, a_1)$, $\texttt{wdf}(a_5, a_2)$, $\texttt{wdf}(a_6, a_3)$, $\texttt{wdf}(a_6, a_5)$, $\texttt{wdf}(a_7, a_4)$, $\texttt{wdf}(a_7, a_6)\}\rangle$.

$\mathcal{V}^a = \{(a_1, 10), (a_2, 20), (a_3, 30), (a_4, 9), (a_5, 30), (a_6, 900), (a_7, 100)\}$

$\mathcal{V}^p = \{(p_1, \texttt{prim} : \texttt{sum}), (p_2, \texttt{prim} : \texttt{mult}), (p_3, \texttt{prim} : \texttt{div})\}$.

In this graph, a_1, a_2, a_3, a_4 are inputs and a_7 is an output, whereas the other artifacts a_5 and a_6 are intermediary artifacts.

In order to define a reproducibility function, some topological constraints are introduced on OPM graphs. Ways of relaxing these constraints are discussed further in Section 7.

Definition 1 (Reproducibility Graph Constraints).

1. *Well formed: OPM graphs are supposed to be well-formed as per OPM v1.1 [26]: an artifact can be generated by at most one process, and there exists no cycle formed of edges of type was-derived-from.*

[11] As in [41], we ignore agents in this paper, since their role as catalyst does not directly affect computational reproducibility.

[12] http://eprints.ecs.soton.ac.uk/21992/

2. *Acyclic: For the purpose of the reproducibility semantics, we assume that an OPM graph is fully acyclic, i.e., no cycle can be formed with any edge.*
3. *Sortable: From the acyclicity property, we can derive an ordered list of processes such that the invocation of a process in the list does not require the outputs of any subsequent processes in the list. Furthmore, we assume the existence of sortProcesses, a function*[13] *that can returns this ordered list of processes.*
4. *No was-triggered-by: We assume that there is no edge was-triggered-by.*
5. *Role unicity: We also assume that inputs and outputs are uniquely identified by a role, for a given process: for any edges* $\texttt{used}(p, r, a_1), \texttt{used}(p, r, a_2)$*, then* $a_1 = a_2$*; likewise, for any edges* $\texttt{wgb}(a_1, r, p), \texttt{wgb}(a_2, r, p)$*, then* $a_1 = a_2$*.*

□

Our formalization relies on partial maps, mapping roles to some set (e.g., role-values or role-artifacts). For instance $rv^* \in Role \rightarrow Value$. Given a role $r \in DOM(rv^*)$, then $rv^*(r)$ denotes the value associated with r. For convenience, our notation also allows for such a partial map to be seen as a finite sequence of role-value pairs $\mathbb{P}(RoleValue)$, over which we can perform a *map* operation. Finite sequences are represented with the SML list notation $x :: x^*$.

4.3. Reproducibility of an Account-Less OPM Graph

Key to reproducibility is a definition for each of the primitives referred to by processes of an OPM graph. To this end, we introduce the concept of a *primitive environment* associating primitive names with *primitives*, which essentially produce some output values for some input values. In OPM, values (whether input or output of a process) are associated with a role. Hence, primitives take sets of role-value pairs, and produce sets of role-value pairs (cf. Figure 3, line 2).

Furthermore, edges of type *was-derived-from* (`wdf`) need to be asserted and cannot be inferred (cf. OPM specification [26] for the definition of OPM inference and [42] for its characterization). The only component that has knowledge of such dependencies is the primitive itself. So we expect a primitive not only to return a set of role-value pairs, but also which *was-derived-from* dependency exists between which output (identified by its role) and which input (similarly identified). Such a pair of roles, an element of *EdgeSpec* (cf. Figure 3, line 4), can be used to reconstruct the appropriate *was-derived-from* edge in the resulting graph. The intent of the type *InvocationResult* is similar, except that it refers to role-artifact pairs rather than role-value pairs (cf. Figure 3, line 5).

Reproducing an OPM graph results in a new OPM graph. There is some mundane activity involved in constructing such a new OPM graph: how should artifacts and processes be created? Hence, we assume the presence of factories: given a node from the old graph and the current new graph that we are in the process of building, such a factory results in a new node and a new OPM graph containing that node (line 8–10). Parameterizing the reproducibility function by such factories allows us to consider a range of options, such as the resulting graph has the same nodes as the original or the resulting graph has fresh nodes.

Finally, to be able to compare the results of the reproduced computation and the original results (whether final or intermediary), we introduce a mapping function that maps nodes of the original graph to nodes of the resulting graph. The pair of mappers *AMapper* and *PMapper* is conveniently referred to as *Mappers* (cf. Figure 3, lines 1–3).

The reproducibility function has a signature of type *Reproduce* (cf. Figure 3, lines 16–18). Given a graph factory (i.e. how we construct nodes), a primitive environment, an input OPM graph and some input artifacts, a reproducibility function must produce a resulting OPM graph and mappings from the input graph to the output graph. Such a reproducibility function recursively traverses the input graph, reproducing the invocation of every primitive; to this end, it relies on an auxiliary function of type *Execute*, reproducing the invocation of a single process (cf. Figure 3, lines 12–14).

Having defined all the necessary sets, the reproducibility-semantics can be expressed as in Figure 4. The reproducibility-semantics is captured by function $reproduce_1$ of type *Reproduce*, which relies on the auxiliary function $reproduce_2$ to recursively execute each process of its input graph G_1^v.

The auxiliary function $reproduce_2$ relies on *execute* to invoke primitives, and for each such invocation, ensures that new edges of the appropriated type are accumulated in the graph G_2^v.

The auxiliary function *execute* of type *Execute* invokes a primitive, as per defined in the primitive environment $\mathcal{E}$ and extends its input graph G^v with new artifacts and processes, created with the respective factories. It also ensures that the mappers are suitably extended with new mappings and valuations for the process and its output artifacts.

4.4. A Definition of Reproducibility

In this section, we show that the reproducibility function can be applied to any OPM graph, but it does not always terminate, or it results in a different graph. Hence, we define here what we mean by a reproducible graph. First, we define role-value map equality.

Definition 2 (Role-Value Map Equality). *Two role-value maps* rv_1^* *and* rv_2^* *are equal if* $DOM(rv_1^*) = DOM(rv_2^*)$ *and* $rv_1^*(r) = rv_2^*(r)$ *for any* $r \in DOM(rv_1^*)$*.* □

Graph equality "up to mapping" is satisfied if graphs are isomorphic and have the same values for corresponding nodes. For completeness, formalization is provided as follows.

Definition 3 (Graph Equality "up to mapping"). *Let* $G_1^v = (G_1, \mathcal{V}_1^a, \mathcal{V}_1^p)$ *and* $G_2^v = (G_2, \mathcal{V}_2^a, \mathcal{V}_2^p)$ *be two full OPM graphs. Let* $\mathcal{M}$ *be a pair* $\langle \mathcal{M}^a, \mathcal{M}^p \rangle$ *of bijections such that their domains are the nodes in* G_1 *and their range the nodes in* G_2*. Two graphs* G_1^v, G_2^v *are equal "up to mapping* $\mathcal{M}$*", noted*

$$G_1^v \approx^{\mathcal{M}} G_2^v,$$

[13] The function *sortProcesses* selects a deterministic order of processes, when it does not exist in the graph.

$$
\begin{array}{rrl}
1 & \mathit{PrimitiveEnv} & = \mathit{PrimitiveName} \to \mathit{Primitive} \\
2 & \mathit{Primitive} & = \mathbb{P}(\mathit{RoleValue}) \to (\mathbb{P}(\mathit{RoleValue}) \times \mathbb{P}(\mathit{EdgeSpec})) \\
3 & \mathit{RoleValue} & = \mathit{Role} \times \mathit{Value} \\
4 & \mathit{EdgeSpec} & = \mathit{Role} \times \mathit{Role} \\
5 & \mathit{InvocationResult} & = \mathbb{P}(\mathit{Role} \times \mathit{Artifact}) \times \mathbb{P}(\mathit{EdgeSpec}) \\
6 & \mathit{Invocation} & = \mathit{PrimitiveName} \times \mathbb{P}(\mathit{Role} \times \mathit{Artifact}) \times \mathbb{P}(\mathit{Role} \times \mathit{Artifact}) \\
7 & & \\
8 & \mathit{ArtifactFactory} & = \mathit{Artifact} \to \mathit{OPMGraph} \to \mathit{Artifact} \times \mathit{OPMGraph} \\
9 & \mathit{ProcessFactory} & = \mathit{Process} \to \mathit{OPMGraph} \to \mathit{Process} \times \mathit{OPMGraph} \\
10 & \mathit{GraphFactory} & = \mathit{ArtifactFactory} \times \mathit{ProcessFactory} \\
11 & & \\
12 & \mathit{Execute} & = \mathit{GraphFactory} \to \mathit{PrimitiveEnv} \\
13 & & \to \mathit{Invocation} \to \mathit{Process} \to \mathit{FullOPMGraph} \to \mathit{Mappers} \\
14 & & \to \mathit{InvocationResult} \times \mathit{FullOPMGraph} \times \mathit{Mappers} \\
15 & & \\
16 & \mathit{Reproduce} & = \mathit{GraphFactory} \to \mathit{PrimitiveEnv} \\
17 & & \to \mathit{FullOPMGraph} \to \mathit{InputArtifacts} \\
18 & & \to \mathit{Mappers} \times \mathit{FullOPMGraph}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{AMapper} & = & \mathit{Artifact} \to \mathit{Artifact} \\
\mathit{PMapper} & = & \mathit{Process} \to \mathit{Process} \\
\mathit{Mappers} & = & (\mathit{AMapper} \times \mathit{PMapper}) \\
 & & \\
\mathit{InputArtifacts} & = & \mathit{AResolver}
\end{array}
$$

Figure 3: Reproducibility in a single Account

if the following conditions hold:

- *For any* $a \in G_1, \mathcal{V}_1^a(a) = \mathcal{V}_2^a(\mathcal{M}^a(a))$.
- *For any* $p \in G_1, \mathcal{V}_1^p(p) = \mathcal{V}_2^p(\mathcal{M}^p(p))$.
- *For any edge* $\langle n_1, n_2\rangle \in G_1$ *(or* $\langle n_1, r, n_2\rangle \in G_1$*),* $\langle \mathcal{M}(n_1), \mathcal{M}(n_2)\rangle \in G_2$ *(or* $\langle \mathcal{M}(n_1), r, \mathcal{M}(n_2)\rangle \in G_2$*).*
- *For every edge* $\langle n_1', n_2'\rangle \in G_2$ *, there exists* $\langle n_1, n_2\rangle \in G_1$ *such that* $\langle \mathcal{M}(n_1), \mathcal{M}(n_2)\rangle = \langle n_1', n_2'\rangle$ *(and likewise, for edges with roles).*

□

Graph equality up to mapping implies that graphs have the same artifacts and processes (up to naming), the same used and was-generated-by edges connecting processes and artifacts, and similar was-derived-from edges linking artifacts.

Having specified a reproducibility function in Figure 4, we can now define the notion of a reproducible graph as follows. A graph is reproducible if, given the same inputs, the reproducibility function produces another graph that is equal "up to the mapping" between nodes, for a given primitive environment. In other words, we define a provenance graph as reproducible, if combined with a primitive environment, it contains enough information to be interpreted as a program whose execution can yield an isomorphic provenance graph.

Definition 4 (Reproducible Graph). *Let* G_1^v *be an OPM graph. Let* $(\mathcal{F}^a, \mathcal{F}^p)$ *be artifact/process factories; let* $\mathcal{E}$ *be a primitive environment, let* $in(G_1^v)$ *be the values of input artifacts in* G_1^v*. Let* $(\mathcal{M}_2, G_2^v) = reproduce_1\ (\mathcal{F}^a, \mathcal{F}^p)\ \mathcal{E}\ G_1^v\ in(G_1^v)$*.*

The graph G_1^v *is reproducible in* $\mathcal{E}$*, if the following holds:*

$$G_1^v \approx^{\mathcal{M}_2} G_2^v.$$

□

We observe that the reproducibility function may not be defined, for different reasons, which we now discuss and illustrate with simple examples pertaining to Figure 1. *(i)* An incomplete set of inputs is provided: e.g., $\{(a_1, 10), (a_2, 20), (a_3, 30)\}$. *(ii)* Inputs or intermediary results are not in the domain of some primitives: e.g., $\{(a_1, 10), (a_2, 20), (a_3, 30), (a_4, 0)\}$. *(iii)* Incorrect number of inputs, incorrect roles or incorrect types are provided to a primitive; e.g., for primitive environment mapping `prim:sum` to the unary log function. We note that some of these failure reasons can be checked statically without re-executing primitives, if primitive signature and arity are available.

Based on Definition 4, for given factories and primitive environment, we can identify the class of reproducible OPM graphs. We note that not all graphs are reproducible. For instance, the OPM graph of Figure 1 is no longer reproducible with a primitive environment associating p_3 to the addition operation, since the graph output would be 909 instead of 100. Likewise, if a primitive returns different was-derived-from edges, the graph is not reproducible either. If the primitive environment maps p_1 to `prim:sum` defined as $\langle\ \lambda\ rv^*.\langle out, rv^*(summand_1) + rv^*(summand_2)\rangle, \{\langle out, summand_1\rangle, \langle out, summand_2\rangle\}\ \rangle$, or to the primitive ignoring its arguments $\langle\ \lambda\ rv^*.\langle out, 30\ \rangle, \{\}\rangle$, generated artifacts will have the same values as their original counterparts, but graph topologies will differ. Hence, with the latter primitive environment, the graph is not reproducible for such primitive definition.

Definition 4 is related to Cheney's pointwise approximation [43]; a pointwise approximation of a graph is a function that returns the same outputs (and intermediary results) when provided with the same inputs. Definition 4 generalizes this notion by mandating that the graph topology be preserved. It assumes that a same primitive environment is used to compute the original graph and the reproduced graph (hence, performing a consistency check [15]). The reproducibility function however allows other primitive environments to be used to reproduce

$$
\begin{array}{l}
\mathit{reproduce}_1 : \mathit{Reproduce}\\
\mathit{reproduce}_1\ (\mathcal{F}^a,\ \mathcal{F}^p)\ \mathcal{E}\ G^v_1\ \theta =\\
\mathit{let}\ (\mathcal{M}_2, G^v_2)\ =\ \mathit{initGraphForInputs}\ \mathcal{F}^a\ (\mathit{graphInputs}(G^v_1))\ \mathcal{M}_{Init}\ \theta\\
\mathit{in}\ \mathit{reproduce}_2\ (\mathcal{F}^a,\ \mathcal{F}^p)\ \mathcal{E}\ (\mathit{sortProcesses}\ G^v_1)\ G^v_1\ \mathcal{M}_2\ G^v_2\\
\mathit{end}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{reproduce}_2\ \mathcal{F}\ \mathcal{E}\ []\ G^v_1\ \mathcal{M}_1\ G^v_2 = (\mathcal{M}_1, G^v_2)}\\
\multicolumn{3}{l}{\mathit{reproduce}_2\ \mathcal{F}\ \mathcal{E}\ (p :: l)\ G^v_1\ \mathcal{M}_1\ G^v_2 =}\\
\mathit{let}\ \ \mathit{inv} & = & \mathit{extractInvocation}\ p\ G^v_1\ \mathcal{M}_1\\
\quad (\_, \mathit{ru}^*, \_) & = & \mathit{inv}\\
\quad ((\mathit{rg}^*, \mathit{sp}^*), G^v_3, \mathcal{M}_2) & = & \mathit{execute}\ \mathcal{F}\ \mathcal{E}\ \mathit{inv}\ p\ G^v_2\ \mathcal{M}_1\\
\quad (G_3,\ \mathcal{V}^a_3,\ \mathcal{V}^p_3) & = & G^v_3\\
\quad p_2 & = & \mathcal{M}_2\ p\\
\quad \mathit{used}^* & = & \mathit{map}\ (\lambda(r, a).\ \mathtt{used}(p_2, r, a))\ \mathit{ru}^*\\
\quad \mathit{wgb}^* & = & \mathit{map}\ (\lambda(r, a).\ \mathtt{wgb}(a, r, p_2))\ \mathit{rg}^*\\
\quad \mathit{wdf}^* & = & \mathit{map}\ (\lambda(r_1, r_2).\ \mathtt{wdf}(\mathit{rg}^*(r_1),\ \mathit{ru}^*(r_2)))\ \mathit{sp}^*\\
\quad G^v_4 & = & (G_3\ \cup\ \mathit{wdf}^*\ \cup\ \mathit{wgb}^*\ \cup\ \mathit{used}^*,\ \mathcal{V}^a_3,\ \mathcal{V}^p_3)\\
\multicolumn{3}{l}{\mathit{in}\ \mathit{reproduce}_2\ \mathcal{F}\ \mathcal{E}\ l\ G^v_1\ \mathcal{M}_2\ G^v_4}\\
\multicolumn{3}{l}{\mathit{end}}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{execute} : \mathit{Execute}}\\
\multicolumn{3}{l}{\mathit{execute}\ (\mathcal{F}^a, \mathcal{F}^p)\ \mathcal{E}\ \mathit{inv}\ p_0\ (G, \mathcal{V}^a, \mathcal{V}^p)\ (\mathcal{M}^a, \mathcal{M}^p) =}\\
\mathit{let}\ \ (n, \mathit{ru}^*, \mathit{rg}^*) & = & \mathit{inv}\\
\quad \mathit{pr} & = & \mathcal{E}(n)\\
\quad (\mathit{rv}^*, \mathit{sp}^*) & = & \mathit{pr}\ (\mathit{map}\ (\lambda(r, a).\ (r, \mathcal{V}^a(a)))\ \mathit{ru}^*)\\
\quad (\mathit{rg}^*_2, G_1) & = & \mathit{mapWithGraph}\ \ (\lambda(r, v).\lambda G.\ \ \mathit{let}\ (a, G) = \mathcal{F}^a\ \mathit{rg}^*(r)\ G\\
 & & \qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ ((r, a), G)\\
 & & \qquad\qquad\qquad\qquad\qquad\quad \mathit{end})\\
 & & \qquad\qquad\qquad\qquad \mathit{rv}^*\ G\\
\quad (p_2, G_2) & = & \mathcal{F}^p\ p_0\ G_1\\
\quad \mathcal{V}^a_2 & = & \mathcal{V}^a[\mathit{rg}^*_2 \rightarrow^{r^*} \mathit{rv}^*]\\
\quad \mathcal{V}^p_2 & = & \mathcal{V}^p\ [p_2 \rightarrow n]\\
\quad \mathcal{M}^a_2 & = & \mathcal{M}^a[\mathit{rg}^* \rightarrow^{r^*} \mathit{rg}^*_2]\\
\quad \mathcal{M}^p_2 & = & \mathcal{M}^p\ [p_0 \rightarrow p_2]\\
\multicolumn{3}{l}{\mathit{in}\ \ ((\mathit{rg}^*_2, \mathit{sp}^*), (G_2, \mathcal{V}^a_2, \mathcal{V}^p_2), (\mathcal{M}^a_2, \mathcal{M}^p_2))}\\
\multicolumn{3}{l}{\mathit{end}}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{initGraphForInputs}\ \mathcal{F}^a\ a^*\ (\mathcal{M}^a, \mathcal{M}^p)\ \mathcal{V}^a =}\\
\mathit{let}\ \ (a^*_2, G) & = & \mathit{mapWithGraph}\ \mathcal{F}^a\ a^*\ G_{Init}\\
\quad \mathcal{M}^a_2 & = & \mathcal{M}^a[a^* \rightarrow^* a^*_2]\\
\quad \mathcal{V}^a_2 & = & \mathcal{V}^a_{Init}[a^*_2 \rightarrow^* (\mathit{map}\ \mathcal{V}^a\ a^*)]\\
\multicolumn{3}{l}{\mathit{in}\ ((\mathcal{M}^a_2, \mathcal{M}^p), (G, \mathcal{V}^a_2, \mathcal{V}^p_{Init}))}\\
\multicolumn{3}{l}{\mathit{end}}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{extractInvocation}\ p\ (G, \mathcal{V}^a, \mathcal{V}^p)\ (\mathcal{M}^a, \mathcal{M}^p) : \mathit{Invocation} =}\\
\mathit{let}\ \ \mathit{used}^* & = & \mathit{getUsed}(p, G)\\
\quad a^* & = & \mathit{map}\ (\lambda\mathtt{used}(p, r, a).a)\ \mathit{used}^*\\
\quad a^*_2 & = & \mathit{map}\ \mathcal{M}^a\ a^*\\
\quad \mathit{ru}^*_2 & = & \mathit{map}_2\ (\lambda(\mathtt{used}(p, r, a), a_2).(r, a_2))\ \mathit{used}^*\ a^*_2\\
\quad \mathit{rg}^*_2 & = & \mathit{map}\ \ \ (\lambda(\mathtt{wgb}(a, r, p)).(r, a))\ \mathit{getGeneratedBy}(p, G)\\
\multicolumn{3}{l}{\mathit{in}\ (\mathcal{V}^p(p), \mathit{ru}^*_2, \mathit{rg}^*_2)}\\
\multicolumn{3}{l}{\mathit{end}}
\end{array}
$$

p	$\in$	$\mathit{Process}$	process
a	$\in$	$\mathit{Artifact}$	artifact
a^*	$\in$	$\mathbb{P}(\mathit{Artifact})$	artifact set
G	$\in$	$\mathit{OPMGraph}$	OPM graph
G^v	$\in$	$\mathit{FullOPMGraph}$	Full OPM graph
$\mathcal{E}$	$\in$	$\mathit{PrimitiveEnv}$	primitive env.
pr	$\in$	$\mathit{Primitive}$	primitive
$\mathcal{F}^a$	$\in$	$\mathit{ArtifactFactory}$	artifact factory
$\mathcal{F}^p$	$\in$	$\mathit{ProcessFactory}$	process factory
$\mathcal{F}$	$\in$	$\mathit{GraphFactory}$	graph factory
sp	$\in$	$\mathit{EdgeSpec}$	edge specification
rv	$\in$	$\mathit{RoleValue}$	role-value pair
rv^*	$\in$	$\mathit{Role} \rightarrow \mathit{Value}$	role value map
rg	$\in$	$\mathit{Role} \times \mathit{Artifact}$	role-generated artifact pair
rg^*	$\in$	$\mathit{Role} \rightarrow \mathit{Artifact}$	role generated artifact map
ru	$\in$	$\mathit{Role} \times \mathit{Artifact}$	role-used artifact pair
ru^*	$\in$	$\mathit{Role} \rightarrow \mathit{Artifact}$	role used artifact map
$\mathcal{V}^a$	$\in$	$\mathit{AResolver}$	artifact resolver
$\mathcal{V}^p$	$\in$	$\mathit{PResolver}$	process resolver
$\mathcal{M}^a$	$\in$	$\mathit{AMapper}$	artifact mapper
$\mathcal{M}^p$	$\in$	$\mathit{PMapper}$	process mapper
θ	$\in$	$\mathit{InputArtifacts}$	input artifacts

$$
\begin{array}{lcl}
f[x \rightarrow y] & = & \lambda v.\ \mathit{if}\ v = x\ \mathit{then}\ y\ \mathit{else}\ f(v)\\
 & & \\
f[x :: x^* \rightarrow^* y :: y^*] & = & f[x \rightarrow y][x^* \rightarrow^* y^*]\\
f[[] \rightarrow^* []] & = & f
\end{array}
$$

extension by role

$$
\begin{array}{lcl}
f[(r, x) :: x^* \rightarrow^{r^*} y^*] & = & f[x \rightarrow y^*(r)][x^* \rightarrow^{r^*} y^*]\\
f[[] \rightarrow^{r^*} y^*] & = & f
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapWithGraph} : (\alpha \rightarrow \beta \rightarrow \gamma * \beta) \rightarrow \mathbb{P}(\alpha) \rightarrow \beta \rightarrow \mathbb{P}(\gamma) \times \beta\\
\mathit{mapWithGraph}\ f\ []\ G\ = ([], G)\\
\mathit{mapWithGraph}\ f\ (x :: x^*)\ G =\\
\mathit{let}\ \ (y, G_2)\ \ \ =\ \ \ f\ x\ G\\
\quad (y^*, G_3)\ \ \ =\ \ \ \mathit{mapWithGraph}\ f\ x^*\ G_2\\
\mathit{in}\ (y :: y^*, G_3)\\
\mathit{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{getUsed}(p, G) = \{\mathtt{used}(p, r, a) \in G\}\\
\mathit{getGeneratedBy}(p, G) = \{\mathtt{wgb}(a, r, p) \in G\}\\
\mathit{sortProcesses} : \mathit{FullOPMGraph} \rightarrow \mathit{list}(\mathit{Process})
\end{array}
$$

$$
\begin{array}{l}
G_{Init} = \{\emptyset, \emptyset\}\\
\mathcal{V}^a_{Init} = \lambda a.\bot\\
\mathcal{V}^p_{Init} = \lambda p.\bot
\end{array}
$$

$$
\begin{array}{l}
\mathit{graphInputs}(G) = \{a \in G \mid \mathtt{wgb}(a, r, p) \notin G,\ \text{ for any } r, p\}\\
\mathit{graphOutputs}(G) = \{a \in G \mid \mathtt{used}(p, r, a) \notin G,\ \text{ for any } r, p\}\\
\mathit{graphInterm}(G) = \{a \in G\} \setminus\\
\quad (\mathit{graphInputs}(G) \cup \mathit{graphOutputs}(G))
\end{array}
$$

Figure 4: Reproducibility Semantics

computation (such as upgraded libraries [40]); we discuss the opportunities these present in Section 6.

In Section 5, we investigate the implementation of this semantics in a reproducibility service. Beforehand, we focus on reproducibility in the more general multi-account graphs. In this context, we establish a property of reproducible graphs.

4.5. Multi-Account OPM Graphs

So far, we have tackled account-less OPM graphs. In this section, we provide a definition of multi-account graphs with a view of defining the associated reproducibility semantics. Figure 5 displays a definition of a multi-account graph *MAccOPMGraph*. It is a triple consisting of a set of accounts, a partial function mapping an account to an OPM graph, and a refinement function.

$$
\begin{array}{rcl}
\mathit{Account} & = & \text{primitive set} \\
\mathit{Refinement} & = & \mathit{Account} \to \mathit{Process} \to \mathit{Account} \\
\mathit{MAccOPMGraph} & = & \mathbb{P}(\mathit{Account}) \\
 & & \times(\mathit{Account} \to \mathit{OPMGraph}) \\
 & & \times \mathit{Refinement} \\
\mathit{FullMAccOPMGraph} & = & \mathit{MAccOPMGraph} \times \mathit{AResolver} \\
 & & \times \mathit{PResolver}
\end{array}
$$

Figure 5: Multi-Account OPM Graph

With this definition, the set of accounts known to a *MAccOPMGraph* is given by the first component of the triple. For each account, the multi-account OPM graph provides us with one OPM graph.

The third component of a *MAccOPMGraph* is a refinement function, which is a partial function capturing a specific form of refinement, corresponding to process nesting due to *procedural abstraction*. A process is allowed to be refined into a subgraph (itself consisting of processes and artifact and associated edges); if p is the process, α the account in which p occurs, the subgraph must correspond to an account, say α'. In that case, the refinement function ρ is such that $\rho\ \alpha\ p = \alpha'$.

To be well-formed, a multi-account graph needs to satisfy some constraints, as follows:

Definition 5 (Well Formed Multi-Account Graph). *Let $\mathcal{G} \in$ MAccOPMGraph be the triple (α^*, Γ, ρ). The graph $\mathcal{G}$ is well-formed, if the following constraints hold:*

- *for any account $\alpha \in \alpha^*$, then $\Gamma(\alpha)$ is defined;*
- *for any account $\alpha \in \alpha^*$ and process p, if $\rho\ \alpha\ p$ is defined, then p is a process in graph $\Gamma(\alpha)$.*

We note that ρ is not necessarily defined for all processes in all accounts, meaning that some processes are not refined into subgraphs. □

A *FullMAccOPMGraph* includes artifact and process resolvers, mapping them to values and primitive names, respectively. Two OPM graphs in a multi-account OPM graph overlap if they share some artifact or process. We note that these resolvers are defined for a *FullMAccOPMGraph*, meaning that an artifact or a process is associated with a single value for all the accounts of the graph. The reason for this design decision is that we seek to define a reproducibility function, and we see accounts as a mechanism that provides multiple levels of details about a *same* execution.

To enable the definition of a reproducibility function, we set some strict constraints on the refinement function. To help their formulation, we introduce a relation between accounts.

Definition 6 (Descendant). *Let $\mathcal{G} \in$ MAccOPMGraph be the triple (α^*, Γ, ρ). Let α_1, α_2 be two accounts of $\mathcal{G}$, the account α_2 is said to be a descendant of α_1, noted $\alpha_1 \lhd \alpha_2$, if $\rho\ \alpha_1\ p = \alpha_2$ for some p belonging to $\Gamma(\alpha_1)$. We use $\trianglelefteq^*$ to denote the reflexive, transitive closure of $\lhd$.* □

Beyond well-formedness, a graph must satisfy constraints to ensure that it can be "evaluated" by the reproducibility function. These constraints do not exist in the original OPM specification. Instead, they reflect the view that each account can be interpreted like a detailed execution trace of a single process.

Definition 7 (Account Refinement Constraints). *Let $\mathcal{G} \in$ MAccOPMGraph be (α^*, Γ, ρ). Account refinements form a hierarchy, without sharing, that satisfies the following constraints:*

- *Acyclic: there is no sequence $\alpha_0, \ldots, \alpha_n$ where $\alpha_i \lhd \alpha_{i+1}$ (for $i = 0, \ldots, n-1$) and $\alpha_n = \alpha_0$, for any process p_i and $n \geq 1$.*
- *No Account Sharing: let $\rho\ \alpha_1\ p_1 = \alpha_2$ and $\rho\ \alpha_3\ p_2 = \alpha_4$. We have that $\alpha_2 = \alpha_4$ if and only if $\alpha_1 = \alpha_3$ and $p_1 = p_2$.*
- *No Process Sharing: for any distinct accounts α_1, α_2 in α^*, $\Gamma(\alpha_1)$ and $\Gamma(\alpha_2)$ do not have any common processes.*
- *Consistent generation: if there are edges $wgb(a, r_1, p_1)$ in $\Gamma(\alpha_1)$ and $wgb(a, r_2, p_2)$ in $\Gamma(\alpha_2)$, then $\alpha_1 \trianglelefteq^* \alpha_2$ or $\alpha_2 \trianglelefteq^* \alpha_1$.*
- *Sortable: for any account α in $\mathcal{G}$, let $\langle p_0, \ldots, p_{n-1} \rangle = sortProcesses(\Gamma(\alpha))$, then if $\rho\ \alpha\ p_i = \alpha_i$ and $\rho\ \alpha\ p_j = \alpha_j$, if $i < j$, then $(graphOutputs(\Gamma(\alpha_j)) \cup graphInterm(\Gamma(\alpha_j))) \cap graphInputs(\Gamma(\alpha_i)) = \emptyset$*
- *Input Preserving: for any account α_i in $\mathcal{G}$, an input artifact a in $graphInputs(\Gamma(\alpha_i))$ cannot be generated in an account α_j, where $\alpha_i \trianglelefteq^* \alpha_j$.*
- *Output preserving: outputs of a process that is refined in an account must be generated in a refinement of that process.*

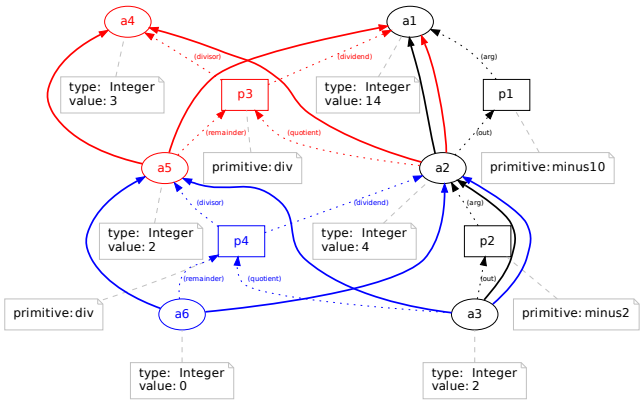
□

The above properties are purely syntactic, constraining the topology of a multi-account OPM graph. In addition, the following property refers to the primitive environment.

Definition 8 (Refinement consistency). *An artifact generated by two processes in different accounts requires the corresponding primitive to produce the same value.* □

Taken together, Definitions 7 and 8 provide a set of conditions that OPM graphs need to satisfy, for provenance claims in different accounts to be consistent, in the context of a given primitive environment.

Figure 6 contains a multi-account graph, with a toplevel account α_0 represented in black. Process p_1 is refined into account α_1, represented in red, whereas process p_2 is refined into account α_2 represented in blue. In other words, $\alpha_0 \lhd \alpha_1$ and $\alpha_0 \lhd \alpha_2$. This graph satisfies the account refinement constraints of Definition 7. Indeed, the descendant relationship is acyclic and forms a tree strictly. The graphs $\Gamma(\alpha_0)$, $\Gamma(\alpha_1)$, $\Gamma(\alpha_2)$ overlap over artifacts but not processes. Artifact a_2 was generated by p_1 in $\Gamma(\alpha_0)$ and by p_3 in $\Gamma(\alpha_1)$, with $\alpha_0 \lhd \alpha_1$ (and similarly for a_3). Processes in $\Gamma(\alpha_0)$ can be sorted as $\langle p_1, p_2 \rangle$, and inputs of $\Gamma(\alpha_1)$ (i.e., a_1, a_4) are not outputs or intermediaries of $\Gamma(\alpha_2)$. Artifact a_1 is an input in $\Gamma(\alpha_0)$ and in the refinement $\Gamma(\alpha_2)$. Finally, artifacts a_2 and a_3, generated respectively by p_1 and p_2 in $\Gamma(\alpha_0)$, are also generated in the refinements α_1 and α_2.



$\rho\ \alpha_0\ p_1 = \alpha_1, \rho\ \alpha_0\ p_2 = \alpha_2$ with α_0: black, α_1: red, α_2: blue.

Figure 6: A Multi-Account OPM Graph (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

In this graph, we see that the refinement $\Gamma(\alpha_1)$ can have more inputs than process p_1: this is the essence of abstraction, in $\Gamma(\alpha_1)$, details of the computation can be revealed, when they were hidden in α_0. Likewise, $\Gamma(\alpha_1)$ can have more outputs than process p_1. Furthemore, a_5, an output of $\Gamma(\alpha_1)$ is allowed to be used as input of $\Gamma(\alpha_2)$, despite being absent in graph $\Gamma(\alpha_0)$.

So far, our discussion of this graph has been independent of the primitive environment. We can see that the graph satisfies Definition 8: if p_1 and p_2 are associated with primitives subtracting 10 and 2 respectively, and p_3 and p_4 are both associated with the division primitive, then we see that for the input values ($a_1 = 14, a_4 = 3$), p_1 and p_3 both assign value 4 to a_2 (and similarly, for a_3).

4.6. Reproducibility of a Multi-Account OPM Graph

Figure 7 displays the reproducibility semantics for multi-account OPM graphs, expressed by a function of type *MAccReproduce*. Provided with the appropriate graph factories and a primitive environment, for a given account and input artifacts, it takes a FullMAccOPMGraph and returns another FullMAccOPMGraph and a set of mapper functions. The account provided as input is the account in which computation needs to be reproduced. Mappers can now map accounts of the original graph to accounts in the resulting graph; account mappings are noted $\mathcal{M}^\alpha$. Likewise, factories comprise a factory for accounts; account factories are noted $\mathcal{F}^\alpha$.

The reproducibility semantics is expressed by function $reproduce_4$, invoking the recursive function $reproduce_3$ after initializing a *FullMAccOPMGraph* acting as an accumulator. The function $reproduce_3$ uses a breadth-first strategy to reproduce a multi-account graph. Its third argument lists all accounts, at the current level, which need to be reproduced, whereas its fourth argument enumerates all direct descendant accounts, which will be iterated over, at the next level. The function $reproduce_3$ relies on $reproduce_2$ (defined in Figure 4) iteratively reproducing each account (and its refinements).

To ensure that nodes are properly shared across accounts in the resulting graph, we rely on the following operator, defining a factory function that creates a new artifact only for those that have not been mapped yet.

$$\mathcal{F}^a \setminus \mathcal{M}^a = \lambda x G.\ \begin{array}{ll} (\mathcal{M}^a x, \{\mathcal{M}^a x\} \cup G) & \text{if } \mathcal{M}^a x \neq \bot, \\ \mathcal{F}^a\ x\ G & \text{otherwise .} \end{array}$$

We also introduce a combining operator, that takes the "union" of two functions, preferring the first over the second.

$$\theta_1 \uplus \theta_2 = \lambda x.\ \begin{array}{ll} \theta_1 x & \text{if } \theta_1 x \neq \bot \\ \theta_2 x & \text{otherwise.} \end{array}$$

After reproducing the current account α, the function $reproduce_3$ ensures that all accounts that are direct descendant of α, noted α^*, are also given the current set of artifact values, $(\mathcal{V}_2^a \circ \mathcal{M}_2^a)$, as a way of resolving the refinements' inputs; likewise, any remaining accounts at the same level have their inputs extended in the same way. Referring to the example of Figure 6, this ensure that a_5 is a provided as an input to account α_2, after completion of evaluation of account α_1.

We note that the reproducibility function may be provided with inputs that differ from those that were used in the original execution. For instance, the graph of Figure 6 may be executed with inputs ($a_1 = 100, a_4 = 10$) and the same primitive environment, but would result in a_2 being assigned two different values. Hence, the reproducibility function is not defined for such a graph, these inputs and primitive environment. However, if p_1 (and likewise p_2) was assigned to primitive "div10", dividing its input by 10, then this inconsistency would not occur.

The following lemma summarizes a key property of the reproducibility function. If a graph $\mathcal{G}_2$ is the result of the reproducibility function for a given primitive environment, $\mathcal{G}_2$ is itself reproducible in that environment.

$$
\begin{aligned}
AccMapper &= Account \to Account \\
AccountFactory &= Account \to MAccOPMGraph \to Account \times MAccOPMGraph \\
MAccGraphFactory &= GraphFactory \times AccountFactory \\
MAccMapper &= Mappers \times AccMapper
\end{aligned}
$$

$$
\begin{aligned}
MAccReproduceIter = {} & MAccGraphFactory \to PrimitiveEnv \to \mathbb{P}(Account \times InputArtifacts) \to \mathbb{P}(Account \times InputArtifacts) \\
& \to FullMAccOPMGraph \to MAccMapper \to FullMAccOPMGraph \\
& \to MAccMapper \times FullMAccOPMGraph
\end{aligned}
$$

$$
\begin{aligned}
MAccReproduce = {} & MAccGraphFactory \to PrimitiveEnv \to (Account \times InputArtifacts) \to FullMAccOPMGraph \\
& \to MAccMapper \times FullMAccOPMGraph
\end{aligned}
$$

$$
\begin{aligned}
& reproduce_3 : MAccReproduceIter \\
& reproduce_3\ \mathcal{F}\ \mathcal{E}\ []\ []\ (\mathcal{G}, \mathcal{V}^a, \mathcal{V}^p)\ \mathcal{M}\ (\mathcal{G}_1, \mathcal{V}^a_1, \mathcal{V}^p_1) = (\mathcal{M},\ (\mathcal{G}_1, \mathcal{V}^a_1, \mathcal{V}^p_1))
\end{aligned}
$$

$$
reproduce_3\ \mathcal{F}\ \mathcal{E}\ []\ next\ (\mathcal{G}, \mathcal{V}^a, \mathcal{V}^p)\ \mathcal{M}\ (\mathcal{G}_1, \mathcal{V}^a_1, \mathcal{V}^p_1) = reproduce_3\ \mathcal{F}\ \mathcal{E}\ next\ []\ (\mathcal{G}, \mathcal{V}^a, \mathcal{V}^p)\ \mathcal{M}\ (\mathcal{G}_1, \mathcal{V}^a_1, \mathcal{V}^p_1)
$$

$$
\begin{aligned}
& reproduce_3\ \mathcal{F}\ \mathcal{E}\ ((\alpha, \theta) :: l)\ next\ (\mathcal{G}, \mathcal{V}^a, \mathcal{V}^p)\ \mathcal{M}\ (\mathcal{G}_1, \mathcal{V}^a_1, \mathcal{V}^p_1) = \\
& \quad let \\
& \qquad (\alpha^*_0, \Gamma, \rho) = \mathcal{G} \\
& \qquad G = \Gamma\ \alpha \\
& \qquad (\alpha_2, \mathcal{G}_2) = (\mathcal{F}^\alpha \setminus \mathcal{M}^\alpha)\ \alpha\ \mathcal{G}_1 \quad \textit{// map account} \\
& \qquad \mathcal{M}^\alpha_2 = \mathcal{M}^\alpha[\alpha \to \alpha_2] \quad \textit{// extend map} \\
& \qquad p^* = sortProcesses(G, \mathcal{V}^a, \mathcal{V}^p) \\
& \qquad ((\mathcal{M}^a_0, \mathcal{M}^p_0), G^v_0) = initGraphForInputs\ (\mathcal{F}^a \setminus \mathcal{M}^a)\ (graphInputs(G))\ (\mathcal{M}^a, \mathcal{M}^p)\ \theta \\
& \quad ((\mathcal{M}^a_2, \mathcal{M}^p_2), (G_2, \mathcal{V}^a_2, \mathcal{V}^p_2)) = reproduce_2\ (\mathcal{F}^a \setminus \mathcal{M}^a_0, \mathcal{F}^p)\ \mathcal{E}\ p^*\ (G, \mathcal{V}^a, \mathcal{V}^p)\ (\mathcal{M}^a_0, \mathcal{M}^p_0)\ G^v_0 \\
& \qquad \alpha p^* = map\ (\lambda p.(\rho\ \alpha, p))\ p^* \quad \textit{// get process-subaccount pairs} \\
& \qquad (\alpha^*_2, \mathcal{G}_3) = mapWithGraph\ \mathcal{F}^\alpha\ (map\ (\lambda(\alpha, p).\alpha)\ \alpha p^*)\ \mathcal{G}_2 \quad \textit{// map subaccounts} \\
& \qquad \mathcal{M}^\alpha_3 = \mathcal{M}^\alpha_2[map\ (\lambda(\alpha, p).\alpha)\ \alpha p^* \to^*\ \alpha^*_2] \quad \textit{// extend map} \\
& \qquad \theta_2 = \theta \uplus (\mathcal{V}^a_2 \circ \mathcal{M}^a_2) \quad \textit{// extend inputs with current resolver} \\
& \qquad l' = (map\ (\lambda(\alpha, \theta_3).\ (\alpha, \theta_3 \uplus (\mathcal{V}^a_2 \circ \mathcal{M}^a_2)))\ l) \quad \textit{// update current level} \\
& \qquad next' = (next@(map\ (\lambda\alpha.\ (\alpha, \theta_2))\ \alpha^*)) \quad \textit{// update next level} \\
& \qquad (\_, \_, \rho_1) = \mathcal{G}_3 \\
& \qquad \rho_2 = \rho_1 \cup \{(\alpha, p_i, \alpha_i) | \alpha_i p_i \in \alpha p^*\} \quad \textit{// extend refinement} \\
& \quad in\ reproduce_3\ \mathcal{F}\ \mathcal{E}\ l'\ next'\ (\mathcal{G}, \mathcal{V}^a, \mathcal{V}^p)\ ((\mathcal{M}^a_2, \mathcal{M}^p_2), \mathcal{M}^\alpha_3)\ (\mathcal{G}_3[\alpha_2 \to G_2][\to \rho_2], \mathcal{V}^a_1 \uplus \mathcal{V}^a_2, \mathcal{V}^p_1 \uplus \mathcal{V}^p_2) \\
& \quad end
\end{aligned}
$$

$$
\begin{aligned}
& reproduce_4 : MAccReproduce \\
& reproduce_4\ \mathcal{F}\ \mathcal{E}\ (\alpha, \theta)\ (\mathcal{G}, \mathcal{V}^a, \mathcal{V}^p) = \\
& \quad reproduce_3\ \mathcal{F}\ \mathcal{E}\ [(\alpha, \theta)]\ (\mathcal{G}, \mathcal{V}^a, \mathcal{V}^p)\ \mathcal{M}_{Init}\ (\mathcal{G}_{Init}, \mathcal{V}^a_{Init}, \mathcal{V}^p_{Init})
\end{aligned}
$$

$\mathcal{F}^a$	$\in$	$ArtifactFactory$	artifact factory	ρ	$\in$	$Refinement$	refinement
$\mathcal{F}^p$	$\in$	$ProcessFactory$	process factory	Γ	$\in$	$Account \to OPMGraph$	account graph map
$\mathcal{F}^\alpha$	$\in$	$AccountFactory$	account factory	θ	$\in$	$InputArtifacts$	input artifacts
$\mathcal{F}$	$\in$	$GraphFactory$	graph factory	$\mathcal{M}^\alpha$	$\in$	$AccMapper$	account mapping

Figure 7: Reproducibility of a Multi-Account OPM Graph

Lemma 1. *For any primitive environment $\mathcal{E}$, for any factory $\mathcal{F}$, for any multi-account graph $\mathcal{G}_1$, for any resolvers $\mathcal{V}_1^a$, $\mathcal{V}_1^p$, and for any input θ_1, if*

$$reproduce_4\ \mathcal{F}\ \mathcal{E}\ (\alpha, \theta_1)\ (\mathcal{G}_1, \mathcal{V}_1^a, \mathcal{V}_1^p)\ = (\mathcal{M}_1,\ (\mathcal{G}_2, \mathcal{V}_2^a, \mathcal{V}_2^p))$$

and

$$reproduce_4\ \mathcal{F}\ \mathcal{E}\ (\mathcal{M}_1^a(\alpha), \theta_2)\ (\mathcal{G}_2, \mathcal{V}_2^a, \mathcal{V}_2^p)\ = (\mathcal{M}_2,\ (\mathcal{G}_3, \mathcal{V}_3^a, \mathcal{V}_3^p)),$$

where $\theta_2(\mathcal{M}_1^a(x)) = \mathcal{V}_1^a(v)$ if $\theta_1(x, v)$, then

$$\mathcal{G}_2 \approx^{\mathcal{M}_2} \mathcal{G}_3.$$

Sketch of proof: we proceed by induction on the ordered list of processes in $\mathcal{G}_2$, and the current level of refinement. At each step, we can establish that the same primitive is applied to the same inputs, generating new output artifacts and edges $\mathcal{G}_3$ which correspond to those in $\mathcal{G}_2$. Furthermore, given that $\mathcal{G}_2$ was itself generated by $reproduce_4$, it does not contain any edge that was not produced by application of a primitive. Hence, the resulting graph $\mathcal{G}_3$ is equal to the original graph $\mathcal{G}_2$ "up to mapping".

5. Semantic Web and Reproducibility

In this section, we investigate how to leverage the reproducibility semantics of Section 4 in order to define a reproducibility service for the Semantic Web. First, we outline an architecture for a reference implementation of this service. Then, we survey the Semantic Web techniques that are exploited in this implementation. Finally, we summarize the assumptions underlying OPM graphs for reproducibility in the Semantic Web.

5.1. Reproducibility Service Architecture

Figure 8 displays the architecture of a reproducibility service. It consists of four key components: *(i)* the *reproducibility engine*, which implements the reproducibility semantics of Section 4; *(ii)* a *triple store* containing representations of the OPM graph to reproduce, and of the OPM graph being generated, and semantic declarations of primitives *(iii)* the *primitive environment*, which maps primitive names to actual primitives; *(iv)* the *execution engines*, which are implementations of the primitive. We now discuss them below.

The reproducibility engine implements the reproducibility semantics. Given an OPM graph, it extracts processes in their invocation order (as specified by dependencies). Each process is annotated with a primitive name.

For instance, in the First Provenance Challenge [27] workflow (discussed in Section 6), process `p1` results from the activation of the primitive `prim:align_warp`.

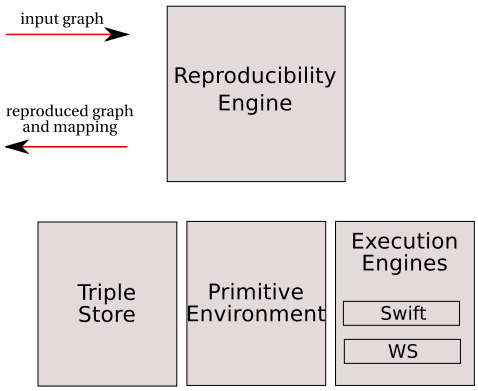


Figure 8: Reproducibility Architecture

```
@prefix opm:  http://openprovenance.org/ontology#
@prefix prim: http://openprovenance.org/primitives#
@prefix pc1:  http://www.ipaw.info/pc1/

pc1:p1 opm:annotation pc1:an1_p1 .

pc1:an1_p1 a opm:Annotation;
           opm:property pc1:pr_9 .

pc1:pr_9 a opm:Property ;
    opm:uri   prim:primitive;
    opm:value prim:align_warp.
```

Each primitive name is associated with an implementation of that primitive by the primitive environment. For instance, the default environment contains a mapping from `prim:align_warp` to the name `http://openprovenance.org/reproducibility/swift#align_warp`, which is itself mapped to an implementation procedure in the execution engine. We have developed two types of execution engines, inline in the Java virtual machine, or delegated to the command line by means of the Swift workflow engine [51].

Swift encompasses Swift script, a declarative language allowing the type of procedures to be declared, as well as the necessary "plumbing" to invoke the executable on the command line. Figure 9 illustrates a Swift declaration for `align_warp` specified as taking four inputs (`i1`, `i2`, `i3`, `i4`) and producing an output (`o`). Each input/output element is associated with its OPM role, unique for this primitive. For instance, the output `o` has role `out`, whereas the anatomy image `i1` has role `img`. We note that the actual invocation makes explicit use of the image files `i1` and `i2`. On the other hand, the header files are explicitly declared with roles `hdr` and `hdrRef`, but are not passed explicitly to the executable `align_warp` since it derives them automatically by replacing the extension `img` by `hdr`.

So far, within an OPM graph that we wish to reproduce, we find an explicit representation of processes and annotations indicating which primitives caused their executions. These primitives are represented by URIs that are mapped to procedure definitions executable by a specific execution engine. These executables need to be fed with the relevant inputs.

In order to illustrate how actual inputs are identified at re-execution time, let us examine an original OPM graph that

```
<procedure name="align_warp">

 <output name="o" type="WarpParameters"   opr:role="out"/>

 <input  name="i1" type="AnatomyImage"    opr:role="img"/>
 <input  name="i2" type="ReferenceImage"  opr:role="imgRef"/>
 <input  name="i3" type="AnatomyHeader"   opr:role="hdr"/>
 <input  name="i4" type="ReferenceHeader" opr:role="hdrRef"/>

 <binding>
   <application>
     <executable>align_warp</executable>
     <function name="filename">
        <variableReference>i1</variableReference>
     </function>
     <function name="filename">
        <variableReference>i2</variableReference>
     </function>
     <function name="filename">
        <variableReference>o</variableReference>
      </function>
     <stringConstant>-m</stringConstant>
     <stringConstant>12</stringConstant>
     <stringConstant>-q</stringConstant>
   </application>
 </binding>
</procedure>
```

Figure 9: Swift Script for `align_warp`

we wish to reexecute. For the sake of illustration, the graph contains `pc1:a1` an input artifact with type `File`, at location `/home/pc1/reference.img`, all of this expressed in RDF as follows.

```
pc1:a1 a opm:Artifact ;
opm:account pc1:black ;
opm:label "Reference Image" ;
opm:type prim:File.

pc1:a1 opm:annotation pc1:an1_a1 .

pc1:an1_a1 a opm:Annotation;
           opm:property pc1:pr_23 .

pc1:pr_23 a opm:Property ;
opm:uri prim:path ;
opm:value "/home/pc1/reference.img" .
```

The reproducibility engine identifies all input artifacts, and uses them to reproduce the original OPM graph. In practice, for each input artifact such as `pc1:a1`, we may want to use its exact file or another one, possibly at an alternative location (since the machine we reproduce may not have the same file system as the one in which the original graph was produced). The reproducibility engine allows for a novel location to be specified, by making use of engine-specific configuration files. For instance, with Swift, a configuration indicates the path where the file must be retrieved from.

```
<variable name="var_i1" type="ReferenceImage">
    <file name="//home/user/pc1/reference.img"/>
</variable>
```

Likewise, we do not necessarily want the reproducibility engine to write output files at the same location, because we do not want to overwrite existing files, or because we do not have the rights to write at that location. So, engine-specific configuration files also specify the actual location where output files must be stored into. In this example, they take place in the current directory.

```
<variable name="var_o" type="WarpParameters">
    <file name="./params1.warp"/>
</variable>
```

This kind of "plumbing" activity is taken care of automatically by pre-defined artifact factories.

5.2. *Semantic Web Techniques for Reproducibility Service*

Multiple Semantic Web technologies have been exploited to build a reference implementation of the reproducibility service. Our baseline is a pre-existing OWL ontology for OPM [47]. We discuss the technologies that have been used, the purpose for which they were used, and their limitations.

Queries. Having represented OPM graphs in RDF, it is natural to use SPARQL [52] to express provenance queries. This use of SPARQL is well documented in the literature (e.g., MINDSWAP [53] and Wings [54] in the first Provenance Challenge, Tetherless [55] in the third Challenge). Two issues are worth discussing. Identifying all inputs of an OPM graph requires negation by failure, which can be encoded in SPARQL as illustrated in Figure 10. Querying transitive properties is discussed next in this section.

```
SELECT ?a
WHERE
  ?a a opm:Artifact
  OPTIONAL {?a opm:_wasGeneratedBy ?p}
  FILTER (!bound(?p))
```

Figure 10: SPARQL query: Inputs to an OPM Graph

Transitive Closures. When multiple or disconnected OPM graphs co-exist in a triple store, and we wish to reproduce a specific result, the above query return all graph inputs, including some that may not have affected the result. Instead, transitive closures are useful to identify the inputs that indirectly cause some specific outputs. Figure 11 displays a possible definition of the multi-step edge `WasDerivedFrom*` as a transitive property in OWL. In the baseline OPM ontology [47], a `WasDerivedFrom` OPM edge is expressed as an OWL class and not an OWL property, since such an encoding facilitates the expressiveness of other OPM properties, such as account membership, time information, and OPM annotations.

We then define an OWL property, `_wasDerivedFrom` with `Artifact` as domain and range. Such a property can be inferred[14] by OWL by means of a property chain: if there is

[14] In the process of defining this ontology, it was necessary to sacrifice elegance for consistency. Indeed, instead of defining properties `effect` and `cause` from `Edge` to `Node`, we had to define `effectWasDerivedFrom` and `causeWasDerivedFrom` from `Wasderivedfrom` to `Node`; similar properties were also defined for other OPM edges. Such a kind of definition ensured the ontology was consistent.

```
// Class: http://openprovenance.org/ontology#WasDerivedFrom

SubClassOf(WasDerivedFrom Edge)
SubClassOf(WasDerivedFrom ObjectSomeValuesFrom(effectWasDerivedFrom Artifact))
SubClassOf(WasDerivedFrom ObjectSomeValuesFrom(causeWasDerivedFrom Artifact))
SubClassOf(WasDerivedFrom ObjectAllValuesFrom(effectWasDerivedFrom Artifact))
SubClassOf(WasDerivedFrom ObjectAllValuesFrom(causeWasDerivedFrom Artifact))

// Object property: http://openprovenance.org/ontology#effectWasDerivedFrom-1

InverseObjectProperties(effectWasDerivedFrom-1 effectWasDerivedFrom)
InverseFunctionalObjectProperty(effectWasDerivedFrom-1)
ObjectPropertyDomain(effectWasDerivedFrom-1 Artifact)
ObjectPropertyRange(effectWasDerivedFrom-1 WasDerivedFrom)

// Object property: http://openprovenance.org/ontology#_wasDerivedFrom

SubObjectPropertyOf(_wasDerivedFrom _wasDerivedFrom_star)
ObjectPropertyDomain(_wasDerivedFrom Artifact)
ObjectPropertyRange(_wasDerivedFrom Artifact)

// Sub property chain axiom

SubObjectPropertyOf(SubObjectPropertyChain(effectWasDerivedFrom-1 causeWasDerivedFrom) _wasDerivedFrom)

// Object property: http://openprovenance.org/ontology#_wasDerivedFrom_star

TransitiveObjectProperty(_wasDerivedFrom_star)
ObjectPropertyDomain(_wasDerivedFrom_star Artifact)
ObjectPropertyRange(_wasDerivedFrom_star Artifact)
```

Figure 11: Transitive `_wasDerivedFrom_star`

an artifact that is the effect of a `WasDerivedFrom` edge, itself with another artifact as its cause, then we can infer a property `_wasDerivedFrom` between these two artifacts.

Then, property `_wasDerivedFrom_star` is defined as transitive, with `_wasDerivedFrom` declared as a subproperty of `_wasDerivedFrom_star`. Similar definitions can be adopted for all multi-step inferences permitted by OPM.

Using OWL to encode OPM inferences (as described in [26]) presents some further challenges. First, we note that OPM completion rules (artifact and process introductions) are not expressible[15] in OWL 2 since they require the inference of novel individuals and properties between novel and existing individuals; encoding these would require us to declare a property as a subproperty of a property chain, which is explicitly forbidden by OWL 2 [56]. Second, the transitive closure defined in Figure 11 ignores accounts: it could infer a `_wasDerivedFrom_star` property by composing (properties inferred from) edges declared in two separate accounts, which is not a legal inference in OPM. A solution to this problem is to consider named graphs [57] to capture assertions related to an account, and ensure that OWL inferences are limited to a graph [58].

Ontological Definitions. We crafted an ontology for reproducibility that is being used at *design* time. Here, design time refers to the moment a system with reproducibility capabilities is being designed; it is to be contrasted with runtime, which denotes the moment when the reproducibility function is being executed.

[15]This problem was also observed by McGrath and Futrelle [48].

The ontology allows us to express core concepts, such as common artifacts (files with their path, numbers, collections), processes (with a reference to a primitive name), and kinds of primitives. This ontology (with prefix `prim`) extends the OPM OWL ontology, by subclassing its core classes artifacts and processes. It allows designers to check for consistency of the various concepts and to express primitive signatures.

Furthermore, still at design time, the ontology allows us to define common *derivations*, corresponding to *EdgeSpec* in the reproducibility semantics, and corresponding subtypes of the `WasDerivedFrom` relation, and associate them with the corresponding primitives. For instance, the addition primitive `PrimitivePlus` has two derivations `Summand0Derivation` and `Summand1Derivation` from its output (identified by role `out0`) to its respective inputs identified by roles `summand0` and `summand1`. Figure 12 illustrate an excerpt of the Primitive ontology.

Hence, the use of ontologies facilitates the typing of primitives (seen as functions operating over typed role-value pairs) and their associated typed derivations. The kind of static type checking of OPM graphs discussed in Section 4.4 can be implemented by means of this ontology, and was referred to as semantic validity by Miles *et al.* [59]. We note that execution of primitives is not modelled by the ontology, but instead relies on the execution engines (cf. Figure 8).

Runtime Rules. The reproducibility semantics expresses how OPM edges can be constructed for every enacted process. Such

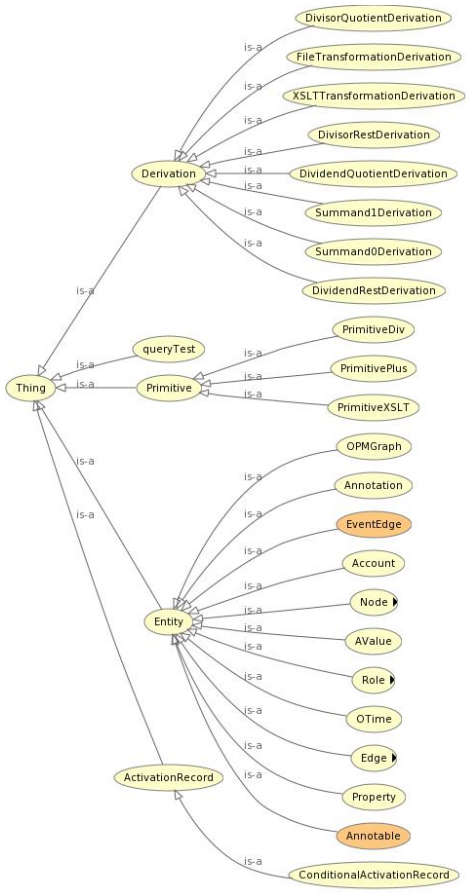


Figure 12: Primitive Ontology

edges can be asserted by new SWRL rules[16]. For instance, in Figure 13, a new edge `_wasDerivedFrom` is added to an OPM graph by means of a SWRL rule that checks the existence of an input artifact `a2` and an output artifact `a1`, respectively used and generated by a process under some roles; this process is associated with a primitive, declared to produce a derivation between those roles.

5.3. Semantic Web Assumptions for Reproducibility

In this section, we summarize the assumption that underpin OPM graphs for reproducibility in the Semantic Web.

- *Well defined global names*: Primitive names should be defined by a global unique name, with a precise meaning. Likewise, their implementations must be uniquely identified. In the Semantic Web tradition, such primitive names and implementations can be identified by URIs. For instance, we previously named the primitive `align_warp` with the URI `prim:align_warp`.

- *Explicit representation of information*: Each execution engine may itself be configurable, and such configurations need to be made explicit. For instance, Figure 9 displays the Swift script for the implementation of `align_warp`. Furthermore, the Swift engine relies on a file `tc.data` to map executable name to a specific executable in the file system; this file must also be made explicit.

[16]It is important to note that OPM provides no inference rule to infer `WasDerivedFrom` edges; instead, they must be asserted [26]. By constructing `WasDerivedFrom` edges by SWRL rules, we are not misinterpreting OPM semantics. Indeed, these edges are constructed according to an ontological definition of primitives, characterizing their signatures and their dependencies. We note that the current version of this ontology only supports static `WasDerivedFrom` edges, always between the same outputs and inputs (identified by their roles), for whatever invocation of the primitive. An alternative design supports conditional edges, according to the inputs to the primitive.

```
Artifact(?a1),
Artifact(?a2),
_wasGeneratedBy(?a1,?p),
_used(?p,?a2),
effectWasGeneratedBy-1(?a1,?g),
cause(?g,?p),
role(?g,?r1),
isKindOf(?r1,?rt1),
effectUsed-1(?p,?u),
cause(?u,?a2),
role(?u,?r2),
isKindOf(?r2,?rt2),
isKindOf(?p,?pt),
hasDerivation(?pt,?d),
effects(?d,?rt1),
causedBy(?d,?rt2)
 -> _wasDerivedFrom(?a1,?a2)
```

Figure 13: SWRL rule to construct derivations according to the semantics of primitives

6. Evaluation

We have undertaken three empirical evaluations aiming to demonstrate the capability of the reproducibility service, and its suitability for deployment in a multi-technology environment such as the Web. We discuss them in turn.

1. Feasibility: First Provenance Challenge. Our first experience is designed to demonstrate that the reproducibility service is capable of reproducing the results of a significant computation. We used the First Provenance Challenge workflow [27], which has become a de-facto benchmark in the provenance community.

By hand, we constructed `pc1trace`[17], an OPM provenance trace of the First Provenance Challenge (see Figure A.18, in Appendix). The trace `pc1trace` is minimal in the sense that it is not annotated or decorated with any information that is not required for reproducibility. We applied the reproducibility function to `pc1trace`, and obtained the following:

- an output trace with the same structure as `pc1trace`;

- a precise mapping of `pc1trace` node ids to the output trace ids;

[17]The OPM graph had to be created after the facts since the first Provenance Challenge predates OPM.

- a set of intermediary and final output files.

We have checked that the resulting trace is identical up to the node ids, the locations of produced files, and the file contents.

Figure 14 summarizes the outcome of this experiment. The input artifacts were chosen to be the same files (at a location of our choice). The intermediary artifacts `a11` to `a24` were shown to be the same. We however observe that the artifacts following the `slicer` stage differed. For instance, the artifact `atlas-x.pgm` had 8 bytes that differed by one unit. This difference is due to a more recent version of the `fsl` library[18] (version `4.1.6.` which was prevailing in 2010), whereas the original artifact was generated by version `3.3.1` of `fsl` in 2006.

	artifact id	success?	comments
inputs	`a1`, ..., `a10`	✔	copies of originals dated 2006-05-31
intermediaries	`a11`, ..., `24`	✔	checked to be the same as originals of 2006-05-31
intermediaries/ outputs	`a25`, ..., `30`	✘	different

Figure 14: Experiment 1: Result Comparability

The outcome of this experiment highlights the benefits of this approach. By keeping explicit provenance and intermediary results (which in this example were *four years old*), we were able to rerun the experiment and compare results. Reproducing the experiment allowed us to identify a specific library as the cause of the divergent results. The installed version was found to be more recent than in the original execution; reverting to the older version of the library, we were then able to reproduce all artifacts.

2. Reproducibility Variants. In the second experiment, we establish that the reproducibility service can easily be customized to invoke multiple execution technologies, such as command line and Java code, simply by changing its primitive environment. Furthermore, the reproducibility service can be parameterized with different graph factories, to customize execution or to ensure the uniqueness of the generated graph.

Java Inline Execution	✔	demonstrated with numeric wflow
WebService Execution	✘	implementation in progress
Swift Execution	✔	demonstrated with PC1 workflow
Replayability [29]	✔	PC1 mock up with dummy primitives
Multi Technology	✔	demonstrated with PC1 workflow
Graph factory variant	✔	output graph with different artifact ids
Graph factory variant	✔	output with different file locations

Figure 15: Experiment 2: Reproducibility Variants

The results of this experiment are summarized in Figure 15. The "Java Inline" execution refers to the ability to invoke Java code directly. This was achieved by reproducing the OPM graph of Figure 1, where the invoked primitives were implemented in Java directly. Alternatively, the PC1 OPM graph (Figure A.18) was reproduced using primitives implemented in Swift, invoking command lines. Work is in progress to support Web Services implementations of primitives; to this end, we are planning to make use of the D-Profile [60] to minimize the size of the OPM graph. We have also demonstrated that the reproducibility service can be configured with mock-up implementation of primitives, which are hardwired and return specific outputs for specific inputs (similarly to Bechhofer's [29] replayability). Such replayability was demonstrated for the PC1 OPM graph, with primitives returning directly the URLs as per specified in the First Challenge; such primitives were then described as "dummies" [27].

A driver for this paper is to provide a reproducibility semantics for an ontology-based representation of provenance, allowing a uniform representation of provenance, despite multiple execution technologies being involved in executions across the Web. To demonstrate this capability, we have configured the reproducibility service, with implementation of PC1 primitives using different technologies, e.g. Swift and Java, and have successfully reproduced the experiment.

The second part of Figure 15 demonstrates how the reproducibility service can be configured with various graph factories. The graph factory can be used to generate new ids for nodes (and edges), but also to change the location of files to be generated by the command line executables, so that they do not overwrite previously existing files. Both were successfully demonstrated using the PC1 OPM graph.

3. Other Inputs. In the third experiment, we establish that the reproducibility service can be used to reproduce experiments with inputs that differ from those used in the original experiment.

New inputs	✔	demonstrated with numeric wflow
Differently encoded inputs	✔	demonstrated with numeric wflow
Change of parameters	✔	Provenance Challenge 1

Figure 16: Experiment 3: Other Inputs

One should note that in this experiment we do not have the original workflow but just a trace of its past execution. Given the numeric expression OPM graph (Figure 1), one can recompute the expression with alternate inputs. When the original process makes decisions on its inputs, the outcome of such decision-making may differ when new inputs are provided. In that case, the provenance trace may not contain enough information to reproduce the original process (essentially alternate branches may be missing). This is an issue that Cheney *et al.* tackle under their "fidelity property" [15], which relies on a form of "continuation" [61], a data structure that combines computational state and program structure, to allow computations to be resumed and continued.

OPM requires the encoding of artifact values to be made explicit. Hence, alternate encodings of a same input can be sup-

[18] http://www.fmrib.ox.ac.uk/fsl/

ported (e.g., an integer passed by reference in a file, instead of by value). We note that this type of conversion, referred to a "shim" by Duncan *et al.* [62], can be handled automatically and systematically in a number of cases using appropriate type declarations [63].

Finally, parameter sweeps are possible by changing workflow parameters, considered as an "input" by the reproducibility service.

7. Discussion

7.1. OPM

This paper is the first to provide an executable semantics for a substantial subset of OPM, independently of a given execution technology. This formalization complements the ones discussed in Section 3.4: Cheney's causal perspective of OPM [43], Moreau *et al.*'s set-theoretic definition of OPM [41], Kwasnikowska *et al.*'s temporal interpretation of OPM [42], and Missier and Goble's translation of OPM to a workflow language [19]. The fact that each formalization covers a different subset of OPM, and that no equivalence between formalizations has been established yet, is indicative of a lack of a "grand theory of OPM".

OPM introduced interesting features, such as the notions of accounts and refinements. This paper has proposed a novel definition for these, which corresponds to the nested invocation of procedures in programming languages: a process can be refined into a subaccount. Alternate definitions have been proposed, and their implication for reproducibility need to be investigated. Kwasnikowska and Van Den Bussche [64] propose a methodology to accommodate hierarchical refinements in OPM. Their notion of refinement allows for an OPM subgraph to be refined into another OPM subgraph. Groth and Moreau [60] propose the D-profile, a profile to express details of execution in distributed systems, such as communication and messages; the D-profile introduces an alternative form of refinement, where an artifact is refined into a subgraph. Whilst the notions of refinement defined in these proposals are more general than the one presented here, no reproducibility semantics of such refinements has been proposed.

This paper has introduced constraints on the topology of OPM graphs to enable the definition of a reproducibility function (cf. Definition 1). It is our belief that the acyclicity constraint could be relaxed whilst still preserving reproducibility, for networks of processes exchanging artifacts. Indeed, provided that there is no cycle with *was-derived-from* edges, we can identify processes in subaccounts that exchange such artifacts. The current semantics would have to be extended in two different ways to support these: procedures would have to be called by "name" and no longer "by value", and processes would have to be ordered across multiple accounts. A number of edges have been ignored in the reproducibility semantics, because they hide execution "details", such as `wtb` and all multi-step edges. It would be interesting to investigate how their temporal interpretation [42] can be folded into the reproducibility semantics.

7.2. OPM and Semantic Web Technologies

McGrath and Futrelle [48] show limitations of SWRL and OWL in expressing OPM inferences. They did not consider property chains as we did in this paper. They propose a hybrid approach combining OWL, SWRL, RDF with extra tools to handle all OPM requirements. We are following a similar approach here. We note that our encoding of OPM graphs differs from the encoding of structured objects by description graphs, as described by Motik *et al.* [65] and Hastings *et al.* [66]. Indeed, as valid OPM graphs are assumed to be acyclic, we did not have to encode such topological constraints in the ontology. However, we share with these approaches the combined use of ontological descriptions and rules.

Zhao's Open Provenance Model Vocabulary (OPMV) [67] aims to encode OPM in RDF, attempting to leverage existing vocabularies and ontologies such as Dublin Core, FOAF, and the Provenance Vocabulary [68], its predecessor. OPMV is work in progress, and does not support the full expressivity of OPM yet. It may benefit from some of the encoding of relations introduced by this paper.

A challenge brought by this work was putting Semantic Web technologies into action in order to implement the reproducibility service. The challenge was both conceptual and implementational. First, there is not a single Semantic Web technology that allows us, today, to tackle all the issues we have encountered: *(i)* SPARQL does not support recursive queries over multi-step OPM edges; *(ii)* Multi-step edges can be inferred by SWRL rules or OWL property chains; *(iii)* OPM n-ary relations are not naturally encoded in RDF; *(iv)* RDF Named graphs go some way capturing OPM features [58] such as account; *(v)* OPM completion rules require the inference of individuals, which can only be supported by some non-standardized extensions. Adopting all these technologies together result in a framework, whose semantics are not clear, and good properties such as inference decidability are lost. From a practical point of view, at the time of writing, only a few reasoner could support the property chains described in this paper (TROWL and Pellet were successful, whilst FACT++ and HermiT failed). Pellet supported many of the above technologies, and was complemented by Java code, but performance of the overall approach remains a serious concern.

7.3. Reproducibility

In this paper, we have essentially regarded an OPM graph as a workflow, interpretable according to the reproducibility semantics. Therefore, this work bears relation with the workflow literature [69]. Techniques such a workflow abstraction and elaboration [70], scheduling [69], and collection-support are also applicable here [70].

The reproducibility semantics has been implemented using the OPM toolbox[19]. Its wrapping as a *reproducibility service* remains to be undertaken. We envisage this service of being capable of taking OPM graphs, and reproducing their execution, timestamping and signing the resulting provenance trace, hence

[19] http://github.com/lucmoreau/OpenProvenanceModel

confirming, in a non-forgeable way, that it is reproducible. Such a service would need to be scalable, and is obviously a good candidate for parallelization.

In Section 3.1, we introduced dimensions to the problem of reproducibility: inputs, primitives, and results. They are captured by θ (inputs), $\mathcal{E}$, $\mathcal{V}^p$ (primitives), and $\mathcal{V}^a$ (results) in the reproducibility semantics. Figure 17 categorizes the various kinds of provenance-based reproducibility found in the literature according to these dimensions.

Inputs	Primitives	Results	
same	same	same	repetition (Miles *et al.* [30])
different	different	different	reenactment (Miles *et al.* [30])
same	same	—	repeatability (Bechhofer *et al.* [29])
same	same	same	reproducibilty (Bechhofer *et al.* [29])
same	mockup	same	replayability (Bechhofer *et al.* [29])
same	multiple variants	same	N-version reproducibility (Levince *et al.* [34])
—	different	—	upgrades (Koop *et al.* [40])
same	same	same	consistency (Cheney *et al.* [15])
different	different	different	fidelity (Cheney *et al.* [15])

Figure 17: Classification of Reproducibility Approaches

In multiple publications [9, 23, 11], our preferred definition of provenance stated that it is an "explicit representation of the processes that *led* to that data item". In particular, we used the past tense to indicate that some processes *produced* a data item. In this paper, we looked at provenance as a *program*, which can be executed in the future. Hence, to accomodate this new perspective on provenance, we propose the following revised definition: provenance of a data item is an explicit representation of a computational activity, which in the past *led* to that data item, and which can be seen as a program and reexecuted in the future, possibly to derive similar new data items. We are not the first to consider an OPM graph as a program. Cheney [43] considers a subset of OPM as a series of nested `let` expressions, and Miles [71] introduces POEM, a textual notation to create OPM graphs.

Davidson *et al.* [72] study the problem of providing workflow data provenance without revealing the functionality of any module. To this end, they focus on the Secure View problem, which consists in ensuring privacy of all modules in a workflow, by hiding the smallest amount of data. The problem is established to be NP-hard, and they propose a polynomial-time approximation. We conjecture that there is a trade-off between full-reproducibility and full-privacy, since the reproducibility semantics expects primitive names (and implementations) to be shared. However, there may be a useful class of reproducibility behaviour, possibly similar to replayability [29], that can be performed on privacy-preserving provenance. Such an investigation has also to take into account the specific OPM graph structure, including *was-derived-from* edges, which partially reveal the private behaviour of processes.

Our assumption in this paper has been that we operate in a non-malicious environment, in which provenance is an authentic record of past execution. If this assumption no longer holds, one needs to identify the trusted base in the execution environment, and possibly exploit cryptographic techniques to be able to attribute provenance claims and check their integrity. Some of these techniques are reviewed elsewhere [11]. Furthermore, one may wonder how faithful a provenance record is to some original computation. Two different approaches should be considered to answer this question. First, with the reproducibility semantics, we have provided a precise meaning for OPM graphs; notions of fidelity [15] can now be adapted to OPM traces. Second, since the meaning of primitives still needs to be defined, the Semantic Web approach plays an important role in specifying ontologies, making their concepts globally referenceable by means of URIs, and standardizing them.

8. Conclusion

Results reproducibility is crucial in scientific and non-scientific contexts to gain confidence in results and ensure their quality. It is particularly important when such results are derived from computations that make use of third-party services across the Web. In this context, ontology-based representations of provenance offer a uniform description of past executions across such services. Provenance is usually considered as a strong foundation for ensuring reproducibility, since its rich representation encompasses the necessary details to reproduce execution steps, and check all results, whether intermediary or final. However, ontology-based representations of provenance lack any formal link with execution, which makes it unclear why provenance is a sound foundation for reproducibility. We have tackled this problem by providing the reproducibility semantics for the Open Provenance Model; this semantics takes the form of a denotational semantics, which assigns well-formed OPM graphs to a function, which for some inputs, produces an OPM graph describing the reproduction of the result.

The benefits of the reproducibility semantics are multifold.

1. It provides a strong, technology-neutral, understanding of provenance by defining the mathematical meaning of OPM graphs. It allows us to define reproducibility formally, and classes of reproducible graphs, for given primitive environments. It is therefore the basis of a theory of provenance-based reproducibility.
2. It is a specification of a *reproducibility service*, which we envision as deployable on the Web or on Intranets. It allows users who publish results and their provenance, to check that their results are reproducible, and users who discover data, to verify how they were produced. Hence, it permits users to increase their confidence in such data.
3. From a methodological viewpoint, one always wonders what should be included in provenance. The semantics provides an algorithmic way to decide what needs to be recorded in provenance to ensure past computation reproducibility.

Our future work will address several concerns. From a theoretical perspective, we will aim to relax the topological constraints that we set on OPM graphs, and define a broader class of reproducible OPM graphs. Better and more scalable Semantic Web reasoning techniques are required to support the OPM specific inferences, and the necessary inferences required for

reproducibility. Finally, we will seek to deploy a reproducibility service in the context of the Fourth Provenance Challenge, as a means to validate, automatically, the provenance traces produced by the participating teams.

9. Acknowledgement

A particular thanks to James Cheney for his constructive comments on the paper. Also, thanks to Jan Van den Bussche and Simon Miles for their feedback on an early draft of the paper, and to Jeff Pan and Nick Gibbins for discussions on ontologies.

References

[1] P. Vytelingum, T. D. Voice, S. D. Ramchurn, A. Rogers, N. R. Jennings, Agent-based micro-storage management for the smart grid, in: Autonomous Agents And MultiAgent Systems (AAMAS 2010), 2010.
URL http://eprints.ecs.soton.ac.uk/18360/

[2] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufman Publishers, 1998.

[3] T. Hey, A. Trefethen, Cyberinfrastructure for e-science, Science 308 (5723) (2005) 817–821.

[4] T. Hey, S. Tansley, K. Tolle (Eds.), The Fourth Paradigm: Data-Intensive Scientific Discovery, Microsoft Research, Redmond, Washington, 2009.
URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/

[5] J. P. Mesirov, Accessible reproducible research, Science 327 (5964) (2010) 415–416. doi:10.1126/science.1179653.

[6] L. Philip, A. Chorley, J. Farrington, P. Edwards, Data provenance, evidence-based policy assessment, and e-social science, in: Third International Conference on e-Social Science, 2007.
URL http://www.scientificcommons.org/40739576

[7] Nature Editorial, Illuminating the black box, Nature 6. doi:10.1038/442001a.

[8] J. Rees, Recommendations for independent scholarly publication of data sets, Tech. rep., Creative Commons Working Paper (Mar. 2010).
URL http://neurocommons.org/report/data-publication.pdf

[9] L. Moreau, P. Groth, S. Miles, J. Vazquez, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, L. Varga, The Provenance of Electronic Data, Communications of the ACM 51 (4) (2008) 52–58.
URL http://www.ecs.soton.ac.uk/~lavm/papers/cacm08.pdf

[10] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, J. Myers, Examining the challenges of scientific workflows, IEEE Computer 40 (12) (2007) 26–34. doi:10.1109/MC.2007.421.
URL http://www.ecs.soton.ac.uk/~lavm/papers/computer07.pdf

[11] L. Moreau, The foundations for provenance on the web, Foundations and Trends in Web Science. doi:10.1561/1800000010.
URL http://eprints.ecs.soton.ac.uk/21691/

[12] P. Buneman, S. Khanna, W.-C. Tan, Why and Where: A Characterization of Data Provenance, in: Proceedings of 8th International Conference on Database Theory (ICDT'01), Vol. 1973 of Lecture Notes in Computer Science, Springer, London, UK, 2001, pp. 316–330. doi:10.1007/3-540-44503-X_20.
URL http://db.cis.upenn.edu/DL/whywhere.pdf

[13] Y. Cui, J. Widom, Practical lineage tracing in data warehouses, in: Proceedings of the 16th International Conference on Data Engineering (ICDE'00), San Diego, California, 2000, pp. 367–378. doi:10.1109/ICDE.2000.839437.
URL http://www-db.stanford.edu/pub/papers/trace.ps

[14] I. Souilah, A. Francalanza, V. Sassone, A formal model of provenance in distributed systems, in: J. Cheney (Ed.), TAPP'09: First workshop on on Theory and practice of provenance, USENIX Association, San Francisco, CA, 2009.
URL http://www.usenix.org/event/tapp09/tech/full_papers/souilah/souilah.pdf

[15] J. Cheney, U. A. Acar, A. Ahmed, Provenance traces (extended report), Tech. Rep. http://arxiv.org/abs/0812.0564v1, University of Edinburgh (Dec. 2008).
URL http://homepages.inf.ed.ac.uk/jcheney/publications/drafts/provenance-traces-tr.pdf

[16] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6, USENIX Association, Berkeley, CA, USA, 2004, pp. 10–10.
URL http://portal.acm.org/citation.cfm?id=1251254.1251264

[17] U. Acar, P. Buneman, J. Cheney, J. Van Den Bussche, N. Kwasnikowska, S. Vansummeren, A graph model of data and workflow provenance, in: Proceedings of the 2nd conference on Theory and practice of provenance, TAPP'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 8–8.
URL http://portal.acm.org/citation.cfm?id=1855795.1855803

[18] Y. Zhao, M. Wilde, I. Foster, Applying the virtual data provenance model, in: L. Moreau, I. Foster (Eds.), Proceedings of the International Provenance and Annotation Workshop 2006 (IPAW'2006), Vol. 4145 of Lecture Notes in Computer Science, Springer, 2006, pp. 148–161. doi:10.1007/11890850_16.
URL http://www.springerlink.com/content/33p526mtu3025h01/?p=cd8c59e856bf4becb50bb7816dc56465&pi=15

[19] P. Missier, C. Goble, Workflows to open provenance graphs, round-trip, Future Generation Computer Systems. In Press. doi:10.1016/j.future.2010.10.012.

[20] S. S. Sahoo, R. S. Barga, J. Goldstein, A. P. Sheth, Provenance algebra and materialized view-based provenance management, Tech. Rep. 76523/tr-2008-170, Microsoft Research (2008).
URL http://research.microsoft.com/pubs/76523/tr-2008-170.pdf

[21] S. S. Sahoo, A. Sheth, C. Henson, Semantic provenance for escience: Managing the deluge of scientific data, Internet Computing, IEEE 12 (4) (2008) 46–54. doi:10.1109/MIC.2008.86.

[22] O. Hartig, Provenance information in the web of data, in: Proceedings of the Linked Data on the Web Workshop (LDOW'09), Madrid, Spain, 2009.
URL http://events.linkeddata.org/ldow2009/papers/ldow2009_paper18.pdf

[23] P. Groth, S. Miles, L. Moreau, A Model of Process Documentation to Determine Provenance in Mash-ups, Transactions on Internet Technology (TOIT) 9 (1) (2009) 1–31. doi:10.1145/1462159.1462162.
URL http://www.ecs.soton.ac.uk/~lavm/papers/toit09.pdf

[24] J. Zhao, Open provenance model vocabulary specification, Tech. rep., University of Oxford (2010).
URL http://open-biomed.sourceforge.net/opmv/ns.html

[25] P. P. da Silva, D. L. McGuinness, R. Fikes, A proof markup language for semantic web services, Inf. Syst. 31 (2006) 381–395. doi:10.1016/j.is.2005.02.003.

[26] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, J. Van den Bussche, The open provenance model core specification (v1.1), Future Generation Computer Systems doi:10.1016/j.future.2010.07.005.
URL http://eprints.ecs.soton.ac.uk/21449/

[27] L. Moreau, B. Ludaescher, I. Altintas, R. S. Barga, S. Bowers, S. Callahan, G. Chin Jr., B. Clifford, S. Cohen, S. Cohen-Boulakia, S. Davidson, E. Deelman, L. Digiampietri, I. Foster, J. Freire, J. Frew, J. Futrelle, T. Gibson, Y. Gil, C. Goble, J. Golbeck, P. Groth, D. A. Holland, S. Jiang, J. Kim, D. Koop, A. Krenek, T. McPhillips, G. Mehta, S. Miles, D. Metzger, S. Munroe, J. Myers, B. Plale, N. Podhorszki, V. Ratnakar, E. Santos, C. Scheidegger, K. Schuchardt, M. Seltzer, Y. L. Simmhan, C. Silva, P. Slaughter, E. Stephan, R. Stevens, D. Turi, H. Vo, M. Wilde, J. Zhao, Y. Zhao, The first provenance challenge, Concurrency and Computation: Practice and Experience 20 (5) (2008) 409–418. doi:10.1002/cpe.1233.
URL http://www.ecs.soton.ac.uk/~lavm/papers/

challenge-editorial.pdf
[28] J. Cheney, S. Chong, N. Foster, M. Seltzer, S. Vansummeren, Provenance: A future history, in: Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications: Onward! Session, 2009, pp. 957–964.
[29] S. Bechhofer, D. De Roure, M. Gamble, C. Goble, I. Buchan, Research objects: Towards exchange and reuse of digital knowledge, in: The Future of the Web for Collaborative Science, 2010. doi:10.1038/npre.2010.4626.1.
URL http://precedings.nature.com/documents/4626/version/1/files/npre20104626-1.pdf
[30] S. Miles, P. Groth, M. Branco, L. Moreau, The requirements of using provenance in e-science experiments, Journal of Grid Computing 5 (1) (2007) 1–25. doi:10.1007/s10723-006-9055-3.
URL http://eprints.ecs.soton.ac.uk/10269/
[31] J. B. Buckheit, D. L. Donoho, Wavelap and reproducibile research, Tech. rep., Stanford University (1995).
URL http://www-stat.stanford.edu/~wavelab/Wavelab_850/wavelab.pdf
[32] M. Schwab, M. Karrenbach, J. Claerbout, Making scientific computations reproducible, Computing in Science and Engineering 2 (6) (2000) 61–67. doi:10.1109/5992.881708.
[33] L. Pan, L. M. Batten, Reproducibility of digital evidence in forensic investigations, in: Digital Forensic Research Workshop (DFRWS'05), 2005.
URL http://www.dfrws.org/2005/proceedings/pan_reproducibility.pdf
[34] B. N. Levine, M. Liberatore, Dex: Digital evidence provenance supporting reproducibility and comparison, in: Proceedings of the Digital Forensic Research workshop(DFRWS'09), 2009. doi:10.1016/j.diin.2009.06.011.
URL http://www.dfrws.org/2009/proceedings/p48-levine.pdf
[35] J. Cheney, C. Lagoze, P. Botticelli, Towards a theory of information preservation, Tech. Rep. TR2001-1841, Cornell University (May 2001).
URL http://hdl.handle.net/1813/5828
[36] V. Stodden, Scientific reproducibility and software (Feb. 2010).
URL http://www.stanford.edu/~vcs/talks/VictoriaStoddenICES-Austin-Feb2010.pdf
[37] S. B. Davidson, J. Freire, Provenance and scientific workflows: challenges and opportunities, in: SIGMOD Conference, 2008, pp. 1345–1350. doi:10.1145/1376616.1376772.
URL http://www.cs.utah.edu/~juliana/pub/freire-tutorial-sigmod2008.pdf
[38] O. Biton, S. Cohen-Boulakia, S. B. Davidson, C. S. Hara, Querying and managing provenance through user views in scientific workflows, in: International Conference Data Engineering (ICDE'08), IEEE Computer Society, Los Alamitos, CA, USA, 2008, pp. 1072–1081. doi:10.1109/ICDE.2008.4497516.
URL http://www.inf.ufpr.br/carmem/pub/icde08.pdf
[39] C. Silva, J. Freire, S. P. Callahan, Provenance for visualizations: Reproducibility and beyond, Computing in Science and Engineering 9 (5) (2007) 82–89. doi:10.1109/MCSE.2007.106.
URL http://www.sci.utah.edu/publications/silva07/Silva_ProvenanceForVis.pdf
[40] D. Koop, C. E. Scheidegger, J. Freire, , C. T. Silva, The provenance of workflow upgrades, in: International Provenance and Annotation Workshop (IPAW '10), 2010.
[41] L. Moreau, N. Kwasnikowska, J. V. den Bussche, The foundations of the open provenance model, Tech. rep., University of Southampton (Apr. 2009).
URL http://eprints.ecs.soton.ac.uk/17282/
[42] N. Kwasnikowska, L. Moreau, J. Van den Bussche, A formal account of the open provenance modelSubmitted for publication.
URL http://eprints.ecs.soton.ac.uk/21819/
[43] J. Cheney, Causality and the semantics of provenance, in: S. B. Cooper, P. Panangaden, E. Kashefi (Eds.), Proceedings Sixth Workshop on Developments in Computational Models: Causality, Computation, and Physics, 2010, pp. 63–74. doi:10.4204/EPTCS.26.6.
URL http://arxiv.org/abs/1006.1429
[44] J. Y. Halpern, J. Pearl, Causes and Explanations: A Structural-Model Approach. Part I: Causes, Br J Philos Sci 56 (4) (2005) 843–887. doi:10.1093/bjps/axi147.
URL http://bjps.oxfordjournals.org/cgi/content/abstract/56/4/843
[45] J. Y. Halpern, J. Pearl, Causes and Explanations: A Structural-Model Approach. Part II: Explanations, Br J Philos Sci 56 (4) (2005) 889–911. doi:10.1093/bjps/axi148.
URL http://bjps.oxfordjournals.org/cgi/content/abstract/56/4/889
[46] P. Missier, D. Turi, C. Goble, T. Oinn, D. De Roure, Taverna workflows: Syntax and semantics, in: IEEE International Conference on e-Science and Grid Computing, IEEE Press, 2007, pp. 441–448.
URL http://eprints.ecs.soton.ac.uk/14700/
[47] L. Moreau, S. Miles, P. Missier, P. Groth. [link].
URL http://openprovenance.org/model/opm.owl
[48] R. E. McGrath, J. Futrelle, Reasoning about provenance with owl and swrl rules, in: AAAI Spring Symposium 2008 "AI Meets Business Rules and Process Management", 2008.
URL http://cet.ncsa.uiuc.edu/publications/mcgrath-futrelle-rules.pdf
[49] W3C Incubator Activity, Provenance incubator group charter (Sep. 2009).
URL http://www.w3.org/2005/Incubator/prov/charter
[50] S. Sahoo, P. Groth, O. Hartig, S. Miles, S. Coppens, J. Myers, Y. Gil, L. Moreau, J. Zhao, M. Panzer, D. Garijo, Provenance vocabulary mappings, Tech. rep., W3C (2010).
URL http://www.w3.org/2005/Incubator/prov/wiki/Provenance_Vocabulary_Mappings
[51] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, reliable, loosely coupled parallel computation, in: IEEE Congress on Services, 2007, pp. 199 –206. doi:10.1109/SERVICES.2007.63.
[52] E. Prud'hommeaux, A. Seaborne (Eds.), SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/ (Jan. 2008).
[53] J. Golbeck, J. Hendler, A semantic web approach to the provenance challenge, Concurrency and Computation: Practice and Experience 20 (5) (2008) 431–439. doi:10.1002/cpe.1238.
URL http://www.mindswap.org/~golbeck/downloads/pc.pdf
[54] J. Kim, E. Deelman, Y. Gil, G. Mehta, V. Ratnakar, Provenance trails in the wings/pegasus system, Concurrency and Computation: Practice and Experience 20 (5) (2008) 587–597. doi:10.1002/cpe.1228.
URL http://twiki.ipaw.info/pub/Challenge/FirstChallengeCCPEPapersPreview/CCPE-WingsPegasus-07.pdf
[55] L. Ding, J. Michaelis, J. McCusker, D. L. McGuinness, Linked provenance data: A semantic web-based approach to interoperable workflow traces, Future Generation Computer Systems. In Press. doi:10.1016/j.future.2010.10.011.
[56] B. Motik, P. F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, M. Smith, Owl 2 web ontology language structural specification and functional-style syntax, Tech. rep., W3C (2009).
URL http://www.w3.org/TR/owl2-syntax/
[57] J. J. Carroll, C. Bizer, P. Hayes, P. Stickler, Named graphs, provenance and trust, in: WWW '05: Proceedings of the 14th international conference on World Wide Web, ACM Press, New York, NY, USA, 2005, pp. 613–622. doi:10.1145/1060745.1060835.
URL http://www2005.org/cdrom/docs/p613.pdf
[58] T. Gibson, K. Schuchardt, E. Stephan, Application of named graphs towards custom provenance views, in: J. Cheney (Ed.), TAPP'09: First workshop on on Theory and practice of provenance, USENIX Association, San Francisco, CA, 2009.
URL http://www.usenix.org/event/tapp09/tech/full_papers/gibson/gibson.pdf
[59] S. Miles, S. C. Wong, W. Fang, P. Groth, K.-P. Zauner, L. Moreau, Provenance-based validation of e-science experiments, Web Semantics: Science, Services and Agents on the World Wide Web 5 (1) (2007) 28–38. doi:10.1016/j.websem.2006.11.003.
URL http://www.ecs.soton.ac.uk/~lavm/papers/WEBSEM07.pdf
[60] P. Groth, L. Moreau, Representing distributed systems using opm, Future Generation Computer Systems. In Press. doi:10.1016/j.future.

2010.10.001.
[61] J. C. Reynolds, The Discoveries of Continuations, Lisp and Symbolic and Computation, Special Issue on Continuations 6 (3/4) (1993) 233–248.
[62] D. Hull, R. Stevens, P. Lord, C. Wroe, C. Goble, Treating shimantic web syndrome with ontologies, in: University, Milton Keynes, UK, 2004.
URL http://www.cs.man.ac.uk/~hulld/papers/shimantic_web_syndrome.pdf
[63] Y. Zhao, J. Dobson, I. Foster, L. Moreau, M. Wilde, A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data, Sigmod Record 34 (3) (2005) 37–43.
URL http://www.ecs.soton.ac.uk/~lavm/papers/sigmod05.pdf
[64] N. Kwasnikowska, J. Van Den Bussche, Multiple and hierarchical refinement in opm (Aug. 2009).
URL http://twiki.ipaw.info/bin/view/OPM/ChangeProposalMultipleHierarchicalRefinement
[65] B. Motik, B. C. Grau, I. Horrocks, U. Sattler, Representing ontologies using description logics, description graphs, and rules, Artificial Intelligence 173 (14) (2009) 1275 – 1309. doi:10.1016/j.artint.2009.06.003.
[66] J. Hastings, M. Dumontier, D. Hull, M. Horridge, C. Steinbeck, U. Sattler, R. Stevens, T. Horne, K. Britz, Representing chemicals using owl, description graphs and rules, 2010.
URL http://hdl.handle.net/10204/4065
[67] J. Zhao, Open provenance model vocabulary specification (Oct. 2010).
URL http://open-biomed.sourceforge.net/opmv/ns.html
[68] O. Hartig, J. Zhao, Guide to the provenance vocabulary (Jul. 2010).
URL http://sourceforge.net/apps/mediawiki/trdf/index.php?title=Guide_to_the_Provenance_Vocabulary
[69] I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Vol. XXII, Springer, 2007.
[70] Y. Gil, P. Groth, V. Ratnakar, C. Fritz, Expressive reusable workflow templates, 2009, pp. 344 –351. doi:10.1109/e-Science.2009.55.
URL http://www.isi.edu/~gil/papers/gil-etal-escience09.pdf
[71] S. Miles, The poem format for opm (Sep. 2010).
URL http://twiki.ipaw.info/bin/view/OPM/POEMFormat
[72] S. B. Davidson, S. Khanna, D. Panigrahi, S. Roy, Preserving Module Privacy in Workflow Provenance, ArXiv e-prints arXiv:1005.5543.

Appendix A. Figure

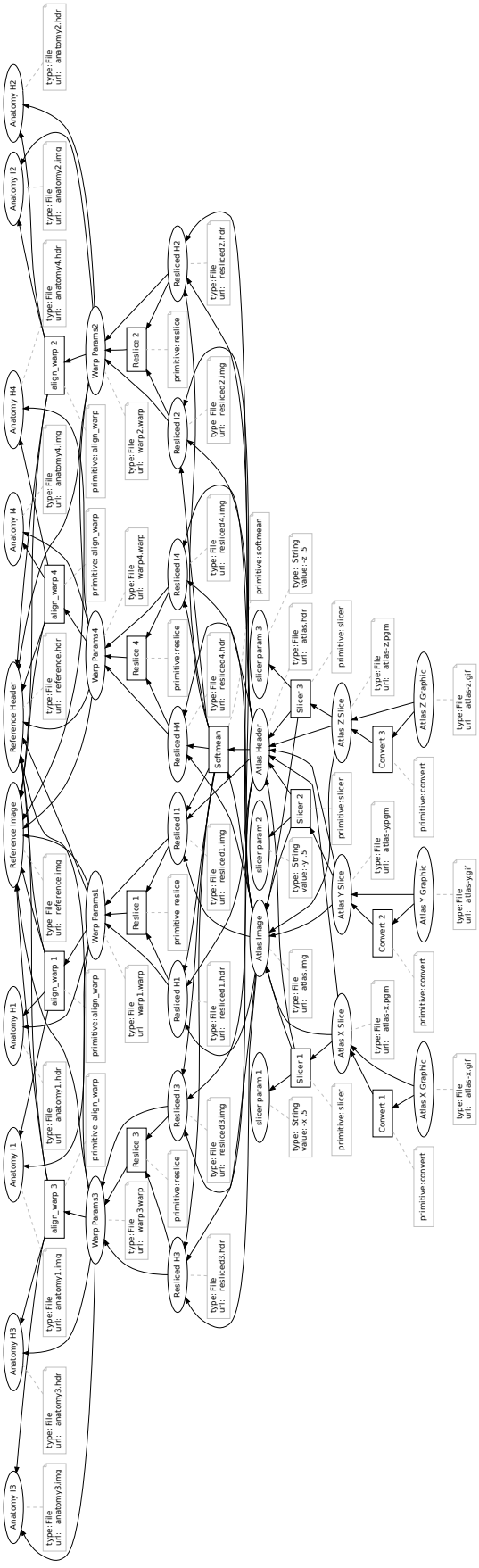


Figure A.18: PC1 OPM graph (for illustrative purpose only)