An Online Validator for Provenance: Algorithmic Design, Testing, and API

Luc Moreau, Trung Dong Huynh, Danius Michaelides

Electronics and Computer Science, University of Southampton

Abstract. Provenance is a record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing. The W3C Provenance Working group has just published the PROV family of specifications, which include a data model for provenance on the Web. The working group introduces a notion of valid PROV document whose intent is to ensure that a PROV document represents a consistent history of objects and their interactions that is safe to use for the purpose of reasoning and other kinds of analysis. Valid PROV documents satisfy certain definitions, inferences, and constraints, specified in PROV-CONSTRAINTS. This paper discusses the design of ProvValidator, an online service for validating provenance documents according to PROV-CONSTRAINTS. It discusses the algorithmic design of the validator, the complexity of the algorithm, how we demonstrated compliance with the standard, and its REST API.

Keywords: provenance, prov, validation

1 Introduction

Provenance is a record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing [1]. (Such a record is encoded in a PROV *document* [2].) The W3C Provenance Working group has just published the PROV family of specifications [3], which include a data model for provenance on the Web (PROV-DM [1]).

PROV comprises a notion of *valid* document [2]. A valid PROV document is one that represents a consistent history of objects and their interactions that is safe to use for the purpose of logical reasoning and other kinds of analysis. Valid PROV documents satisfy certain definitions, inferences, and constraints, specified in PROV-CONSTRAINTS [2]. There are several issues related to PROV-CONSTRAINTS that motivate this work: we discuss them now.

By design, PROV-CONSTRAINTS provides a logic specification of what valid provenance is. This gives implementors the opportunity to design their own implementation, allowing them to meet the requirements set by their applications. To be compliant with PROV-CONSTRAINTS, implementations are expected to produce the same results. In essence, compliance with PROV-CONSTRAINTS is established by observational equivalence with the specification.

PROV-CONSTRAINTS relies on inference rules that lend themselves to implementation by rule-based languages. However, such a paradigm is not an option for some implementors (for instance, having to work with an imperative language or having to control memory management). Furthermore, rule-based specifications do not make explicit the execution order and the type of data structures that are required. Thus, an open research question is the formulation of an algorithm for PROV validation that could be readily adopted by implementors.

In PROV-CONSTRAINTS, not all inferences are necessary for validating documents. Instead, some simply exist because they are considered useful. While this goal helps understand what is meant by provenance, it does not help implementors determine what is essential to implement in a validator.

PROV-CONSTRAINTS does not analyse the complexity of the problem of validity of provenance documents. Understanding this complexity would be useful since provenance documents can become very big, especially those generated by distributed applications with many nodes that run for a very long time.

PROV-CONSTRAINTS is concerned with specifying whether a provenance document is valid. Hence, from this perspective, the outcome of validity checking is a simple yes/no answer. We argue that the validation procedure can also output useful information, which can be exploited by other provenance-processing tools. For instance, the order of events underpinning a provenance document may be useful for Gantt chart plotting applications.

Finally, a question relevant to practitioners is how such validation-checking facility can be accessed. In the context of the Web, exposing such a functionality as a REST service, which can be exploited by browser-based user interfaces or specific applications, would be desirable.

This paper provides answers to these questions, as summarized by its contributions: *(i)* An algorithm to validate provenance; *(ii)* An analysis of its complexity; *(iii)* A REST API for validating provenance graphs, but also accessing validation by-products. Doing so, the paper identifies those essential inferences to save the effort of future validator implementors. Finally, we discuss ProvValidator, an implementation of this algorithm, its exposition as a REST service, and its testing.

Notation Convention We refrain from copying the text of definitions, inference rules, and constraints of PROV-CONSTRAINTS; instead, we refer to them using the following notations DEF 1, INF 5, CON 50, for definitions, inferences, and constraints respectively. In the electronic version of this paper, they directly link to the corresponding entries in the PROV-CONSTRAINTS specification.

2 A Brief Introduction to PROV

PROV is a family of specifications [3] for representing provenance on the Web. It includes a conceptual data model, PROV-DM [1], which can be mapped and serialized to different technologies. There is an OWL2 ontology for PROV, allowing mapping of PROV to RDF, an XML schema for provenance, and a textual representation for PROV.

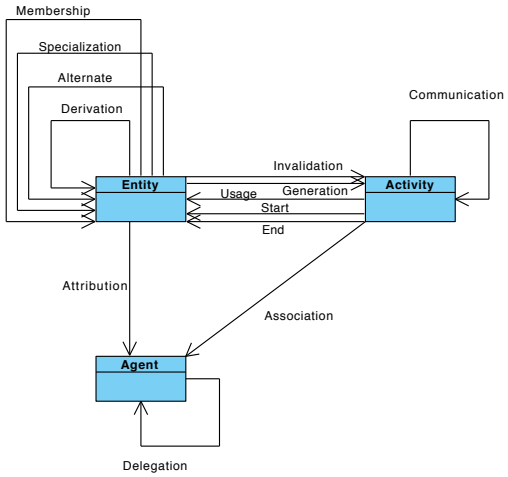

Fig. 1. PROV-DM UML Classes and Associations (simplified view)

Figure 1 summarises PROV-DM [1]. There are three classes: entities (the data or things for which we want to express provenance), activities (representing what happens in systems), and agents (bearing responsibility for things and activities). These three classes can be related with some relations.

1. Derivation view: entities may be derived from others (Derivation).
2. Responsibility view: agents may be responsible for other entities (Attribution), for activities (Association), or for other agents (Delegation).
3. Process view: activities may have used entities (Usage), and vice-versa entities may have been generated by activities (Generation). Furthermore, activities can be informed by other activities (Communication). Activities can be started and ended by entity triggers (Start and End).
4. Alternate and Membership views: entities may have alternates and specializations; entities may be collections with members.

In reality, relations are not necessarily binary, but may involve more instances and may also contain *attributes* such as time information. Table 1 (in Section 3.1) summarizes a textual notation for the model.

3 Validation Algorithm

The overall validation procedure is described in Algorithm 1. It consists of three steps: *(i)* perform the inferences that are relevant to validation; *(ii)* merge terms; *(iii)* finally, if successful, check constraints. We discuss these steps in turn.

Our approach relies on a type system and well-formed terms to deal with illegal situations (many of the so-called impossibility rules in PROV-CONSTRAINTS). First, we present the terms that are accepted by PROV-DM.

Algorithm 1 Validation Procedure

```
1: function VALIDATE(D : Document, T : TypeMap) ⇒ true|fail
2:     D_1, T_1 ← PERFORMRELEVANTINFERENCES(D, T)
3:     res ← MERGETERMS(D_1, T_1)                    ▷ merge can succeed or fail
4:     if res = D_2, U_2 then
5:         return CHECKCONSTRAINTS(D_2, U_2)
6:     else
7:         return false
8:     end if
9: end function
```

3.1 Terms

A document is a set of terms, whose definitions are summarized in Table 1. We assume here that, prior to validation, each term has been expanded[1] (DEF 3) and has been put in a completed form, by introducing existential identifiers, where appropriate, for optional term identifiers (DEF 1) and for optional placeholders (DEF 4). For derivation and association, we consider two variants of these terms, when placeholders are unknown; with these terms, CON 51 is enforced.

There are a few further points worth noting. First, identifiers occur in the first position of terms. Second, entity, activity, and agent statements include a ground identifier specified by the provenance asserter. For relations, their identifiers may be grounded or existential variables (noted with the symbol ν). Finally, time is not a PROV term, but occurs in several of them; hence, its listing in Table 1.

PROV allows for optional extra attributes to be added to terms (see DEF 2). For the purpose of validation, they can simply be ignored, except for `prov:type`, which affects type checking. So, in the interest of space, we have also dropped them from Table 1. We assume a map of types T populated as follows: $v \in T[\alpha]$ whenever the term with identifier α contains an attribute-value pair `prov:type=`v (this caters for EmptyCollection in CON 50). Furthermore, we determine the type of identifiers for entities, activities, and agents, as follows. For every occurrence of variable α^e in a term, $ent \in T[\alpha^e]$; for every α^a occurring in a term, $act \in T[\alpha^a]$; for every α^{ag} occurring in a term, $ag \in T[\alpha^a]$ (cf. CON 50).

We also support bundles [1], which are named sets of terms occurring at the top-level of documents. Due to space limitation, we do not discuss them. Bundles are treated by PROV-CONSTRAINTS as mini-documents that can be validated independently.

3.2 Relevant Inferences

PROV-CONSTRAINTS specifies inferences that potentially affect the outcome of the merging (Section 3.3) and constraint checking procedures (Section 3.4), but also inferences that have no impact on the outcome of the validation procedure. Algorithm 2 specifies the former, whereas Section 3.5 discusses the latter.

[1] Expansion makes explicit optional arguments ommitted in PROV concise notation.

Algorithm 2 Inference Procedure

function PERFORMRELEVANTINFERENCES($D : Document, T : Type$)
$\Rightarrow Document \times Type$
  for *any* α *such that* $relation(\alpha, \ldots) \in D$ **do** ▷ CON 50
    $T \leftarrow T[\alpha \rightarrow \{typeof(relation)\} \cup T[\alpha]]$
  end for
  for *any* α^e *such that* $ent(\alpha^e) \in D$ **do** ▷ INF 7
    if $\nexists\ \alpha^g, \alpha^a, \alpha^t\ gen(\alpha^g, \alpha^e, \alpha^a, \alpha^t) \in D$ **then**
      $D \leftarrow D \cup \{gen(\nu^g, \alpha^e, \nu^a, \nu^t)\}$ with fresh ν^g, ν^a, ν^t
    end if
    if $\nexists\ \alpha^i, \alpha^a, \alpha^t\ inv(\alpha^i, \alpha^e, \alpha^a, \alpha^t) \in D$ **then**
      $D \leftarrow D \cup \{inv(\nu^i, \alpha^e, \nu^a, \nu^t)\}$ with fresh ν^i, ν^a, ν^t
    end if
  end for
  for *any* $\alpha^a, \alpha_1^t, \alpha_2^t$ *such that* $act(\alpha^a, \alpha_1^t, \alpha_2^t) \in D$ **do** ▷ INF 8
    if $\nexists\ \alpha^s, \alpha^e, \alpha_1^a, \alpha^t\ start(\alpha^s, \alpha^a, \alpha^e, \alpha_1^a, \alpha^t) \in D$ **then**
      $D \leftarrow D \cup \{start(\nu^s, \alpha^a, \nu^e, \nu_1^a, \nu^t)\}$ with fresh $\nu^s, \nu^e, \nu_1^a, \nu^t$
    end if
    if $\nexists\ \alpha^n, \alpha^e, \alpha_1^a, \alpha^t\ end(\alpha^n, \alpha^a, \alpha^e, \alpha_1^a, \alpha^t) \in D$ **then**
      $D \leftarrow D \cup \{end(\nu^n, \alpha^a, \nu^e, \nu_1^a, \nu^t)\}$ with fresh $\nu^n, \nu^e, \nu_1^a, \nu^t$
    end if
  end for
  for any $\alpha_1^s, \alpha_1^{en}$ such that **do** ▷ INF 9,INF 10
      $(start(\ \alpha_1^s, \alpha_1^a, \alpha_1^e, \alpha_2^a, \alpha_1^t) \in D$
       or $end(\alpha_1^{en}, \alpha_1^a, \alpha_1^e, \alpha_2^a, \alpha_1^t), \in D)$
      and $\nexists \alpha^g, \alpha^t, gen(\alpha^g, \alpha_1^e, \alpha_2^a, \alpha^t) \in D$
    $T \leftarrow T[\nu^g \rightarrow \{gen\}]$; $D \leftarrow D \cup \{gen(\nu^g, \alpha_1^e, \alpha_2^a, \nu^t)\}$ with fresh ν^g, ν^t
  end for
  for any α^d such that $der(\alpha^d, \alpha_1^e, \alpha_2^e, \alpha^a, \alpha^g, \alpha^u) \in D$ **do** ▷ INF 11
    $D \leftarrow D \cup \{gen(\alpha^g, \alpha_1^e, \alpha^a, \nu_1^t), use(\alpha^u, \alpha^a, \alpha_2^e, \nu_2^t)\}$ with fresh ν_1^t, ν_2^t
  end for
  for any α^{del} such that $del(\alpha^{del}, \alpha_1^{ag}, \alpha_2^{ag}, \alpha^a) \in D$ **do** ▷ INF 14
    if $assoc(\alpha_1^{as}, \alpha^a, \alpha_1^{ag}, \alpha_1^e), assoc(\alpha_2^{as}, \alpha^a, \alpha_2^{ag}, \alpha_2^e) \notin D$ **then**
      for some $\alpha_{1,2}^{as}, \alpha_{1,2}^e$
      $D \leftarrow D \cup \{assoc(\nu_1^{as}, \alpha^a, \alpha_1^{ag}, \nu_1^e), assoc(\nu_2^{as}, \alpha^a, \alpha_2^{ag}, \nu_2^e)\}$
        with fresh $\nu_1^{as}, \nu_1^e, \nu_2^{as}, \nu_2^e$
      $T \leftarrow T[\nu_1^e \rightarrow \{ent\}][\nu_2^e \rightarrow \{ent\}][\nu_1^{as} \rightarrow \{assoc\}][\nu_2^{as} \rightarrow \{assoc\}]$
    end if
  end for
  for any $\alpha_1^e, \alpha_2^e, \alpha_3^e$ such that $spec(\alpha_1^e, \alpha_2^e), spec(\alpha_2^e, \alpha_3^e) \in D$ **do** ▷ INF 19
    $D \leftarrow D \cup \{spec(\alpha_1^e, \alpha_3^e)\}$
  end for
  for any α_1^e, α_2^e such that $spec(\alpha_1^e, \alpha_2^e), ent(\alpha_2^e) \in D$ **do** ▷ INF 21
    $D \leftarrow D \cup \{ent(\alpha_1^e)\}$; $T \leftarrow T[\alpha_1^e \rightarrow T[\alpha_1^e] \cup T[\alpha_2^e]]$
  end for
  for any α_1^e, α_2^e such that $mem(\alpha_1^e, \alpha_2^e) \in D$ **do**
    $T \leftarrow T[\alpha_1^e \rightarrow T[\alpha_1^e] \cup \{nonEmptyCollection\}]$
  end for
  return D, T
end function

Table 1. Terms, Term Types, and Variable Types

identifier type	ground identifier	existential variable	pattern variable	term	type
Entity	id^e	ν^e	α^e	$ent(id^e)$	ent
Activity	id^a	ν^a	α^a	$act(id^a)$	act
Agent	id^{ag}	ν^{ag}	α^{ag}	$ag(id^{ag})$	ag
Generation	id^g	ν^g	α^g	$gen(\alpha^g, id^e, \alpha^a, \alpha^t)$	gen
Usage	id^u	ν^u	α^u	$use(\alpha^u, id^e, \alpha^a, \alpha^t)$	use
Invalidation	id^i	ν^i	α^i	$inv(\alpha^g, id^e, \alpha^a, \alpha^t)$	inv
Start	id^s	ν^s	α^s	$start(\alpha^s, id^a, \alpha^e, \alpha^a, \alpha^t)$	start
End	id^n	ν^n	α^n	$end(\alpha^e, id^a, \alpha^e, \alpha^a, \alpha^t)$	end
Derivation	id^d	ν^d	α^d	$der(\alpha^d, id^e, id^e, \alpha^a, \alpha^g, \alpha^u)$	der
				$der_\perp(\alpha^d, id^e, id^e)$	der $_\perp$
Association	id^{as}	ν^{as}	α^{as}	$assoc_\perp(\alpha^{as}, id^a, \alpha^{ag})$	assoc $_\perp$
				$assoc(\alpha^{as}, id^a, \alpha^{ag}, \alpha^e)$	assoc
Delegation	id^d	ν^d	α^d	$del(\alpha^d, id^{ag}, id^{ag}, \alpha^a)$	del
Attribution	id^{at}	ν^{at}	α^{at}	$attr(\alpha^{at}, id^e, id^{ag})$	attr
Communication	id^c	ν^c	α^c	$comm(\alpha^c, id^a, id^e)$	comm
Influence	id^{inf}	ν^{inf}	α^{inf}	$infl(\alpha^{inf}, id, id)$	infl
Specialization				$spec(id^e, id^e)$	spec
Alternate				$alt(id^e, id^e)$	alt
Membership				$mem(id^e, id^e)$	mem
time	t	ν^t	α^t		

In Algorithm 2, lines 3–5, 23, 32, 39, and 42 update type information. Lines 6–21 ensure that all events relevant to the graph are made explicit: each entity is accompanied by generation and invalidation events, and each activity accompanied by start and end events. INF 9 and INF 10 (lines 22–24) ensure the presence of a generation event *gen* for every trigger α_1^e in *start* and *end* events. INF 11 (lines 25–27) links α^g, α^u in a derivation event *der* to corresponding generation and usage events, *gen*, *use*. INF 14 ensures that a delegation's activity is associated with both its agents (lines 28–34). INF 19 (lines 35–37) computes the transitive closure of specialization *spec*. INF 21 (lines 38–40) propagates types through specializations. Lines 41–43 infer the type *nonEmptyCollection* for any collection that has members. This type is introduced by this algorithm to enforce CON 56 by means of the type system (see Section 3.4).

These inferences are applied till saturation. The algorithm's termination can be explained as follows.

- Lines 6–21 process a finite set of $ent(\alpha^e), act(\alpha^a)$ in a finite document D.
- Lines 22–24 process a finite set of start/end events.
- Lines 25–27 process a finite set of der events.
- Lines 28–34 process a finite set of del events.
- Lines 35– 37 compute a transitive closure over a finite set of spec relations.
- Lines 38–40 process a finite set of spec tuples.
- Lines 41–43 process a finite set of mem relations.

So, the total number of iterations is bounded. We also note that at no point in these inferences, we infer terms from which previous inferences could have derived further terms.

3.3 Term Merging

MergeTerms (see Algorithm 3) ensures that events that must satisfy a uniqueness constraint are merged (lines 4–28); to this end, merging requires unification [4]. If successful, the resulting document is in a "quasi-normal form". Such a quasi-normal form is essentially equivalent to PROV-CONSTRAINTS normal form, except for some inferences that have not been carried out (see Section 3.5).

Algorithm 3 Term Merging Procedure

1: **function** MergeTerms($D : Document, T : TypeMap$)
2: $\Rightarrow Document \times UObject$ | fail
3: $U \leftarrow \langle \emptyset, T \rangle$
4: **repeat**
5: $U_p \leftarrow U$
6: **if** $relation(\alpha, \alpha_{1,1}, \alpha_{1,2}, \ldots), relation(\alpha, \alpha_{2,1}, \alpha_{2,2}, \ldots) \in D$ **then**
7: $U \leftarrow unify^*(\{\alpha_{1,1} = \alpha_{2,1}, \alpha_{1,2} = \alpha_{2,2}, \ldots\}, U)$ ▷ CON 22,CON 23
8: **end if**
9: **if** $gen(\alpha_1^g, \alpha^e, \alpha_1^a, \alpha_1^t), gen(\alpha_2^g, \alpha^e, \alpha_2^a, \alpha_2^t) \in D$ **then** ▷ CON 24
10: $U \leftarrow unify^*(\{\alpha_1^g = \alpha_2^g, \alpha_1^a = \alpha_2^a, \alpha_1^t = \alpha_2^t\}, U)$
11: **end if**
12: **if** $inv(\alpha_1^i, \alpha^e, \alpha_1^a, \alpha_1^t), inv(\alpha_2^i, \alpha^e, \alpha_2^a, \alpha_2^t) \in D$ **then** ▷ CON 25
13: $U \leftarrow unify^*(\{\alpha_1^i = \alpha_2^i, \alpha_1^a = \alpha_2^a, \alpha_1^t = \alpha_2^t\}, U)$
14: **end if**
15: **if** $start(\alpha_1^s, \alpha_1^a, \alpha_1^e, \alpha_2^a, \alpha_1^t), start(\alpha_2^s, \alpha_1^a, \alpha_2^e, \alpha_2^a, \alpha_2^t) \in D$ **then** ▷ CON 26
16: $U \leftarrow unify^*(\{\alpha_1^s = \alpha_2^s, \alpha_1^e = \alpha_2^e, \alpha_1^t = \alpha_2^t\}, U)$
17: **end if**
18: **if** $end(\alpha_1^n, \alpha_1^a, \alpha_1^e, \alpha_2^a, \alpha_1^t), end(\alpha_2^n, \alpha_1^a, \alpha_2^e, \alpha_2^a, \alpha_2^t) \in D$ **then** ▷ CON 27
19: $U \leftarrow unify^*(\{\alpha_1^n = \alpha_2^n, \alpha_1^e = \alpha_2^e, \alpha_1^t = \alpha_2^t\}, U)$
20: **end if**
21: **if** $start(\alpha_1^s, id_1^a, \alpha_1^e, \alpha_2^a, \alpha_1^t), act(id_1^a, \alpha_2^t, \alpha_3^t) \in D$ **then** ▷ CON 28
22: $U \leftarrow unify^*(\{\alpha_1^t = \alpha_2^t\}, U)$
23: **end if**
24: **if** $end(\alpha_1^s, id_1^a, \alpha_1^e, \alpha_2^a, \alpha_1^t), act(id_1^a, \alpha_2^t, \alpha_3^t) \in D$ **then** ▷ CON 29
25: $U \leftarrow unify^*(\{\alpha_1^t = \alpha_3^t\}, U)$
26: **end if**
27: $D \leftarrow applySubstitution(U, D)$
28: **until** $U = U_p$ or $U =$ fail
29: **if** $U =$ fail **then**
30: **return** fail
31: **else**
32: **return** D, U
33: **end if**
34: **end function**

The algorithm's termination can be explained as follows. Lines 4–28 can only generate a finite number of different bindings $\alpha_1 = \alpha_2$, since α_1, α_2 have to occur in a finite document D, and no new variable is generated by this algorithm. So, the number of iterations is bounded.

Term merging relies on unification, where the existential variables are considered as logical variables; for the purpose of validation of provenance terms, we require full unification [4], except for the fact that variables only occur at the top-level of PROV terms and cannot be nested in expressions. In Algorithm 4, the meaning of $U \in UObject$ is now explicit: it pairs up bindings B and a type map T.

3.4 Constraint Checking

Algorithm 5 is concerned with checking the applicable constraints. First, in lines 4–5, reflexive cases of specialization are rejected. Second, leveraging all the type inferences performed in previous steps, lines 6–10 detect type impossibility cases. They are all encoded in Table 2, where the presence of a cross in cell $conflict(\tau_1, \tau_2)$ indicates that τ_1 and τ_2 are conflicting types to which no variable is allowed to be simultaneously assigned. Finally, lines 11–14 detect violations of ordering constraints.

PROV-CONSTRAINTS defines an order between events, as opposed to an order between time instants. Thus, Ordering constraints checking relies on a two-dimensional matrix *order* indicating whether two events, identified by α_1 and α_2, are ordered by a "strictly precede" ($order[\alpha_1, \alpha_2] = 2$) or by a "precede" ($order[\alpha_1, \alpha_2] = 1$) relation, or unordered ($order[\alpha_1, \alpha_2] = 0$). The table *order* is initialized with value 0. The following indicates how the *order* table is assigned values, according to PROV-CONSTRAINTS.

Constraint	Ordering Relation
CON 30, CON 31, CON 32, CON 33, CON 34, CON 35, CON 36, CON 37, CON 38, CON 39, CON 40, CON 41, CON 43, CON 44, CON 45, CON 46, CON 47, CON 48, CON 49	$order[\alpha_1, \alpha_2] = 1$
CON 42	$order[\alpha_1, \alpha_2] = 2$

Next, the transitive closure for the ordering relations is computed by a variant of Floyd-Warshall algorithm [5], using the rule below.

$$\begin{aligned} &\textit{if } order[\alpha_1, \alpha_2] = x, \text{ for some } x > 0 \\ &\textit{and } order[\alpha_2, \alpha_3] = y, \text{ for some } y > 0 \\ &\textit{then } order[\alpha_1, \alpha_3] \leftarrow \max(order[\alpha_1, \alpha_3], x, y) \end{aligned}$$

This rule ensures that a strict precedence between two events is also recorded between sequence of events involving these two. The algorithm is further adapted to work on a sparse matrix representation suitable for provenance graphs.

3.5 Validation-Neutral Inferences

It is safe to ignore some inference rules, referred to as validation-neutral (VN) inferences. VN inferences are such that, for any document D, MERGETERMS

Algorithm 4 Unification Procedure

```
1: function UNIFY*({α^x = α^y} ∪ A, U)
2:     return UNIFY*(A, UNIFY(α^x, α^y, U))
3: end function
4: function UNIFY*(∅, U)
5:     return U
6: end function
7: function UNIFY(α^x, α^y, U)
8:     if U = fail then return fail
9:     end if
10:    if α^x = α^y then return U
11:    end if
12:    if α^x is an existential variable ν^x then
13:        return unifyVar(ν^x, α^y, U)
14:    end if
15:    if α^y is an existential variable ν^y then
16:        return unifyVar(ν^y, α^x, U)
17:    else
18:        return fail                                  ▷ Two distinct ground value
19:    end if
20: end function
21: function UNIFYVAR(ν, α^y, U))
22:    if ν = α^y then return U
23:    end if
24:    if bound(ν, U) then
25:        return UNIFYVAR(lookup(ν, U), α^y, U)
26:    end if
27:    if α^y is a variable ν^y and bound(ν^y, U) then
28:        return UNIFYVAR(ν, lookup(ν^y, U), U)
29:    else
30:        return EXTEND(ν, α^y, U)    ▷ α^y is an unbound variable or a ground value
31:    end if
32: end function
33: function EXTEND(ν, α^y, U)
34:    ⟨B, T⟩ ← U
35:    B' ← B[ν → α^y]
36:    if α^y is a variable ν^y then
37:        T' ← T[ν → T(ν) ∪ T(ν^y)][ν^y → T(ν) ∪ T(ν^y)]
38:    else
39:        T' ← T
40:    end if
41:    return ⟨B', T'⟩
42: end function
```

Algorithm 5 Checking Constraints Procedure

```
1: function CheckConstraints(D : Document, U : UObject)
2:                     ⇒ true|fail
3:     B, T ← U
4:     if spec(α^e, α^e) ∈ D then return fail                    ▷ CON 52
5:     end if
6:     if τ1, τ2 ∈ T[α] for some α in D then    ▷ CON 53,CON 54,CON 55, CON 56
7:         if conflict(τ1, τ2) then
8:             return fail
9:         end if
10:    end if
11:    order ← inferOrderingRelation(D)
12:    order ← transitiveClosure(order)
13:    if order[α, α] = 2 for some α then return fail
14:    end if
15:    return true
16: end function
```

succeeds for D if and only if MergeTerms succeeds for the document obtained by application of VN-inferences to D. Furthermore, application of VN-inferences do not entail ordering constraints that cannot be found otherwise. Below, we list the VN-inferences, and why they can be ignored.

INF 5: the new Generation and Usage events for a new entity always satisfy all ordering constraints.

INF 6: ordering constraints CON 35 can be inferred from CON 33, CON 34, and by transitivity of the ordering relation.

INF 13: the ordering constraints related to Attribution (CON 48) imply the ordering constraints related to Association (CON 47).

INF 15: can be ignored since there is no ordering constraint on Influence.

Likewise, INF 12, INF 16, INF 17, INF 18, INF 20 can be ignored since there is no ordering constraints on Alternate.

4 Complexity Analysis

In this section, we establish that the validation process is polynomial. Specifically, validate is $O(N^3)$, where N is the size of document D. To establish this result, we analyze the complexity of the various steps of the algorithm. We use the superscripts of Figure 1 to denote the number of terms of that type. For instance, we write $f = O(e)$ to say that f grows asymptotically no faster than the number of entities e (itself bounded by N).

PerformRelevantInferences is $O(N^3)$ (see Algorithm 2).

Lines 3–5 $O(N)$ by iterating over all elements and relations;
Lines 6–21 $O(e) + O(a) = O(N)$ by iterating over entities and activities;
Lines 22–24 $O(s) + O(en) = O(N)$ by iterating over all starts and ends;
Lines 25–27 $O(d) = O(N)$ by iterating over all derivations;

	entity	*activity*	*agent*	*generation*	*usage*	*communication*	*start*	*end*	*invalidation*	*derivation*	*derivation*$_\perp$	*revision*	*quotation*	*primarySource*	*attribution*	*association*	*association*$_\perp$	*delegation*	*influence*	*bundle*	*collection*	*emptyCollection*	*person*	*organization*	*softwareAgent*	*nonEmptyCollection*
entity		×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							
activity	×			×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×				×
agent				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							
generation	×	×	×		×	×	×	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×
usage	×	×	×	×		×	×	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×
communication	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×
start	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×
end	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×
invalidation	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×
derivation	×	×	×	×	×	×	×	×	×		×				×	×	×	×		×	×	×	×	×	×	×
derivation$_\perp$	×	×	×	×	×	×	×	×	×	×					×	×	×	×		×	×	×	×	×	×	×
revision	×	×	×	×	×	×	×	×	×						×	×	×	×		×	×	×	×	×	×	×
quotation	×	×	×	×	×	×	×	×	×						×	×	×	×		×	×	×	×	×	×	×
primarySource	×	×	×	×	×	×	×	×	×						×	×	×	×		×	×	×	×	×	×	×
attribution	×	×	×	×	×	×	×	×	×	×	×	×	×	×		×	×	×		×	×	×	×	×	×	×
association	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×		×	×		×	×	×	×	×	×	×
association$_\perp$	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×		×		×	×	×	×	×	×	×
delegation	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			×	×	×	×	×	×	×
influence	×	×	×																	×	×	×	×	×	×	×
bundle		×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							
collection		×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							
emptyCollection		×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							×
person				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							
organization				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							
softwareAgent				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×							
nonEmptyCollection		×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			×				

Table 2. Conflicting Types $conflict(\tau_1, \tau_2)$

Lines 28–34 $O(del) = O(N)$ by iterating over all delegations;
Lines 35–37 $O(spec^3) = O(N^3)$ by computing a transitive closure over the specialization edges;
Lines 38–40 $O(spec^2) = O(N^2)$ by iterating over the transitive closure of specialization edges;
Lines 41–43 $O(mem) = O(N)$ by iterating over membership edges.

Specialization-related inferences aside, each inference adds 2 terms at most to the document; with $O(N)$ inferences, the resulting document remains $O(N)$. In the worst case, a transitive closure over specialization can result in a quadratic number of terms. In practice, we observe that specialization is relatively infrequently used, and that specializations do not form long chains[2]. So, assuming[3] that $spec \ll N$, it is reasonable to conclude that the *average* inferred document is $O(N)$.

[2] It is in fact revisions of entities that potentially create long derivation chains, each entity in the chain being a specialization of one general entity.
[3] This does not hold for the corner case consisting of a document of N specializations.

The complexity of UNIFY is bounded by the number of bindings (see Algorithm 4).

Lines 21–32 $O(|U|)$: worst case is proportional to the number of bindings;
Lines 33–42 $O(1)$: constant time operation.

MERGETERMS is $O(N^2)$ (see Algorithm 3). Worst case binding size is when all variables are to be unified; binding size is proportional to document size.

Lines 6–8 $O(edges \times N)$: worst case scenario, all edges have the same identifier and need to be merged;
Lines 9–11 $O(g \times N)$: worst case scenario, all generation edges have the same entity identifier and need to be merged;
Lines 12–14 $O(i \times N)$: similar worst case scenario for invalidations;
Lines 15–17 $O(s \times N)$: similar worst case scenario for starts;
Lines 18–20 $O(en \times N)$: similar worst case scenario for ends;
Lines 21–23 $O(max(s, a) \times N)$: similar worst case scenario for starts or activities;
Lines 24–26 $O(max(en, a) \times N)$: similar worst case scenario for ends or activities;
Line 27 $O(N^2)$ since *applySubstitution* applies $O(N)$ substitutions, on average, each costing $O(N)$, on average.

The cost of checking constraints is $O(N^3)$ (see Algorithm 5). Let γ be the number of different types.

Lines 4–5 checking this impossibility constraint is $O(spec)$;
Lines 6–10 identifying conflicting pairs of types for each statement is $O(\gamma \times \gamma \times N)$=$O(N)$;
Lines 11–14 The size of *order* is $O(N^2)$, the number of ordering constraints directly inferred is $O(N)$, and the transitive closure computation is $O(N^3)$.

5 Testing and Establishing Compliance with PR

ProvValidator is a Java-based implementation of the algorithm presented in this paper. In order to make sure ProvValidator covers all the specified constraints, we collated a test suite containing 168 unit test cases for specific constraints.[4]

A test case here is a provenance document that is expected to pass or fail a validity check. Hence, the result from validating a test case can be either PASS for a valid provenance document or FAIL for an invalid one. Out of 168 test cases, there are 101 PASS cases and 67 FAIL cases, covering all constraints in the PROV-CONSTRAINTS specification (excluding the inferences). The test suite was reviewed by the W3C Provenance working group and was adopted by the group as the way to establish a validator implementation's compliance with PROV-CONSTRAINTS.

As shown in the PROV implementation report [6], ProvValidator fully covers the PROV-CONSTRAINTS specification by passing all the specified test cases.

[4] The full test suite is available at https://dvcs.w3.org/hg/prov/raw-file/default/testcases/process.html, which also summarizes the coverage of various constraints by its test cases.

6 Validator API

ProvValidator is deployed as a Web service accessible from http://provenance.ecs.soton.ac.uk/. This section discusses how the validation algorithm was exposed by means of a REST API.

In designing an API to expose the validator functionality, we wanted to tackle a number of requirements. First, the API should be easy to use and accessible on the Web. Second, we would be providing a web-based front-end but also expect other tools to interact with the facility. Third, the validation-checking process generates a number of by-products (e.g., ordering matrix, quasi-normal form) that may be of use to other tools, and therefore need to be exposed. Thus, we chose to expose the API as a RESTful web service. In such services, the API's focus is on exposing information as resources, how the information is represented, and the use of the verbs of the HTTP protocol to interact with the service [7].

The primary input to the validation process is a document containing PROV statements. The validator supports a variety of representations of PROV: PROV-N, PROV-XML, various formats of RDF, PROV-JSON. Documents are submitted to the service via the POST HTTP verb to the URL: `/documents/` . The body of the POST request is the PROV document and the `Content-type` HTTP header is used to indicate which representation is being used. In addition, to facilitate easy integration with web-pages, posting of standard HTML form data is also supported; here provenance documents can be submitted inline, by a URL or using the HTML form file upload mechanism. If the document is syntactically correct, a new resource for the document is created, with a URL following the schema `/documents/{id}` where `{id}` is an identifier. This resource represents the provenance document loaded by the service. In a hierarchical fashion, we further expose a number of other resources that are generated by the validation process (see Table 3).

Our API makes use of content negotiation in situations where there are multiple representations of an information resource [8]; then, we issue HTTP `303 See Other` responses to redirect the client to the correct URL for the representation they requested. For example, a client's request for `/documents/{id}` with an `Accept: text/provenance-notation` header is redirected to `/documents/{id}.provn`.

∗ `/documents/`	all the provenance documents
∗ `/documents/{id}`	a provenance document
∗ `/documents/{id}/validation/report`	a report generated by VALIDATE
`/documents/{id}/validation/report/{part}`	a section of the validation report
∗ `/documents/{id}/validation/matrix`	the *order* matrix
∗ `/documents/{id}/validation/normalForm`	the quasi-normal form

Table 3. Resources in the REST API. We use ∗ to indicate a resource that supports content negotiation.

The validation report is a document in XML format that indicates whether a PROV document validated; if not, it also lists problematic statements to help users identify and fix issues. The quasi-normal form and the *order* matrix are the two by-products of the validation process that are made available.

7 Related Work

Two other validators for PROV have been publicly reported in [6]. Paul Groth's prov-check[5], and James Cheney and Stephen Cresswell's checker.pl[6]. The first is based on SPARQL queries, whereas the second is Prolog based. SPARQL queries lend themselves to the implementation of rules, by means of insert statement, however, it is challenging to implement merging of terms with SPARQL only. On the other hand, Prolog comes with rules and unification and therefore handles easily term merging. While their source code is publicly available, it is not directly integrated in a software release that is readily installable. There is also a commercial implementation which reportedly[7] implements aspects of provenance validation using some extensions to OWL-based reasoning.

The PROV-CONSTRAINTS specification was designed with a view to deploying services on the Web supporting this PROV document validation. Several validators exist for other Web technologies. The W3C validator[8] checks the markup validity of Web documents in HTML, XHTML, SMIL, MathML. W3C Jigsaw[9] is a CSS validation service. The Manchester Validator [10] validates OWL ontologies. Finally, W3C also hosts an RDFa validator[11].

The PROV-CONSTRAINTS specification builds upon [9] providing a semantics for OPM [10], a precursor to and subset of PROV.

8 Conclusion

In this paper, we have presented an algorithm for provenance validation. It relies on a minimum set of inferences that have to be performed prior to validation, and on type checking to detect most impossible situations. We expose the algorithm functionality, and validation by-products such as the ordering matrix and quasi-normal form of a document through a REST API.

In this paper, we have also investigated the complexity of the validation process. Inferences are established to be linear in the size of the document to validate. Merging terms is quadratic in its size. This is really a worst case situation: it is indeed possible to generate provenance documents that do not require

[5] prov-check: https://github.com/pgroth/prov-check
[6] checker: https://github.com/jamescheney/prov-constraints
[7] http://semtechbizsf2013.semanticweb.com/sessionPop.cfm?confid=70&proposalid=5118
[8] http://validator.w3.org/
[9] http://jigsaw.w3.org/css-validator/
[10] http://owl.cs.manchester.ac.uk/validator/
[11] http://www.w3.org/2012/pyRdfa/Validator.html

any merging of terms. Finally, checking ordering constraints is cubic in the document size, due to the computing of a transitive closure of some precedence relation; however, it has been shown that it can be implemented efficiently.

Future work will investigate functionality that leverages the validation by-products, including editors of valid provenance and visualization of (in)valid provenance; the presented framework could also be extended with domain specific constraints capable of checking provenance even further.

Acknowledgements

Thanks to the Provenance Working Group members; the co-authors of PROV-CONSTRAINTS, James Cheney, Paolo Missier, Tom De Nies; other implementors of PROV-CONSTRAINTS Paul Groth, James Cheney, and Stephen Cresswell. This work is funded in part by the EPSRC SOCIAM (EP/J017728/1) and ORCHID Projects (EP/I011587/1), the FP7 SmartSociety Project (600854), and the ESRC estat2 (ES/K007246/1).

References

1. Moreau, L., Missier (eds.), P., Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., Miles, S., Myers, J., Sahoo, S., Tilmes, C.: PROV-DM: The PROV Data Model. W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium (October 2013)
2. Cheney, J., Missier, P., Moreau (eds.), L., Nies, T.D.: Constraints of the PROV Data Model. W3C Recommendation REC-prov-constraints-20130430, World Wide Web Consortium (October 2013)
3. Groth, P., Moreau (eds.), L.: PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note NOTE-prov-overview-20130430, World Wide Web Consortium (April 2013)
4. Norvig, P.: Correcting a widespread error in unification algorithms. Softw. Pract. Exper. **21**(2) (February 1991) 231–233
5. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. 2nd edn. McGraw-Hill Higher Education (2001)
6. Huynh, T.D., Groth, P., Zednik (eds.), S.: PROV Implementation Report. W3C Working Group Note NOTE-prov-implementations-20130430, World Wide Web Consortium (April 2013)
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T.: Hypertext transfer protocol – http/1.1. Rfc2068, World Wide Web Consortium (January 1997) Available from `http://www.w3.org/Protocols/Specs.html`.
8. Jacobs, I., Walsh, N.: Architecture of the world wide web, volume one. Technical report, World Wide Web Consortium (2004)
9. Kwasnikowska, N., Moreau, L., Van den Bussche, J.: A formal account of the open provenance model. (December 2010) Under Review.
10. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., Van den Bussche, J.: The open provenance model core specification (v1.1). Future Generation Computer Systems **27**(6) (June 2011) 743–756