W3C Recommendation

Constraints of the PROV Data Model

W3C Recommendation 30 April 2013

This version:
http://www.w3.org/TR/2013/REC-prov-constraints-20130430/
Latest published version:
http://www.w3.org/TR/prov-constraints/
Test suite:
http://dvcs.w3.org/hg/prov/raw-file/default/testcases/process.html
Implementation report:
http://www.w3.org/TR/2013/NOTE-prov-implementations-20130430/
Previous version:
http://www.w3.org/TR/2013/PR-prov-constraints-20130312/ (color-coded diff)
Editors:
James Cheney, University of Edinburgh
Paolo Missier, Newcastle University
Luc Moreau, University of Southampton
Author:
Tom De Nies, iMinds - Ghent University

Please refer to the **errata** for this document, which may include some normative corrections.

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness. PROV-DM is the conceptual data model that forms a basis for the W3C provenance (PROV) family of specifications.

This document defines a subset of PROV instances called *valid* PROV instances, by analogy with notions of validity for other Web standards. The intent of validation is to ensure that a PROV instance 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 instances satisfy certain definitions, inferences, and constraints. These definitions, inferences, and constraints provide a measure of consistency checking for provenance and reasoning over provenance. They can also be used to normalize PROV instances to forms that can easily be compared in order to determine whether two PROV instances are equivalent. Validity and equivalence are also defined for PROV bundles (that is, named instances) and documents (that is, a toplevel instance together with zero or more bundles).

The PROV Document Overview describes the overall state of PROV, and should be read before other PROV documents.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical

report can be found in the W3C technical reports index at http://www.w3.org/TR/.

PROV Family of Documents

This document is part of the PROV family of documents, a set of documents defining various aspects that are necessary to achieve the vision of inter-operable interchange of provenance information in heterogeneous environments such as the Web. These documents are listed below. Please consult the [PROV-OVERVIEW] for a guide to reading these documents.

- PROV-OVERVIEW (Note), an overview of the PROV family of documents [PROV-OVERVIEW];
- PROV-PRIMER (Note), a primer for the PROV data model [PROV-PRIMER];
- PROV-O (Recommendation), the PROV ontology, an OWL2 ontology allowing the mapping of the PROV data model to RDF [PROV-O];
- PROV-DM (Recommendation), the PROV data model for provenance [PROV-DM];
- PROV-N (Recommendation), a notation for provenance aimed at human consumption [PROV-N];
- PROV-CONSTRAINTS (Recommendation), a set of constraints applying to the PROV data model (this document);
- PROV-XML (Note), an XML schema for the PROV data model [PROV-XML];
- PROV-AQ (Note), mechanisms for accessing and querying provenance [PROV-AQ];
- PROV-DICTIONARY (Note) introduces a specific type of collection, consisting of key-entity pairs [PROV-DICTIONARY];
- PROV-DC (Note) provides a mapping between PROV-O and Dublin Core Terms [PROV-DC];
- PROV-SEM (Note), a declarative specification in terms of first-order logic of the PROV data model [PROV-SEM];
- PROV-LINKS (Note) introduces a mechanism to link across bundles [PROV-LINKS].

Endorsed By W3C

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

Please Send Comments

This document was published by the Provenance Working Group as a Recommendation. If you wish to make comments regarding this document, please send them to public-prov-comments@w3.org (subscribe, archives). All comments are welcome.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

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. This document complements the PROV-DM specification [PROV-DM] that defines a data model for provenance on the Web. This document defines a form of validation for provenance.

1.1 Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

In this document, logical formulas contain variables written as lower-case identifiers. Some of these variables are written beginning with the underscore character _, by convention, to indicate that they appear only once in the formula. Such variables are provided merely as an aid to the reader.

1.2 Purpose of this document

The PROV Data Model, PROV-DM, is a conceptual data model for provenance, which is realizable using different representations such as PROV-N and PROV-O. A ***PROV instance*** is a set of PROV statements. A ***PROV document*** consists of an anonymous instance, called the ***toplevel instance***, together with zero or more named instances, called ***bundles***. For example, a PROV document could be a .provn document, the result of a query, a triple store containing PROV statements in RDF, etc. The PROV-DM specification [PROV-DM] imposes minimal requirements upon PROV instances. A valid PROV instance corresponds to a consistent history of objects and interactions to which logical reasoning can be safely applied. PROV instances need not be valid. The term valid is chosen by analogy with notions of validity in other W3C specifications. This terminology differs from the usual meaning of "validity" in logic; our notion of validity of a PROV instance/document is closer to logical "consistency".

This document specifies *definitions* of some provenance statements in terms of others, *inferences* over PROV instances that applications MAY employ, and also defines a class of valid PROV instances by specifying *constraints* that valid PROV instances must satisfy. There are four kinds of constraints: *uniqueness constraints*, *event ordering constraints*, *impossibility constraints*, and *type constraints*. Further discussion of the semantics of PROV statements, which justifies the definitions, inferences and

constraints, and relates the procedural specification approach taken here to a declarative specification, can be found in the formal semantics [PROV-SEM].

We define validity and equivalence in terms of a concept called normalization. Definitions, inferences, and uniqueness constraints can be applied to normalize PROV instances, and event ordering, typing, and impossibility constraints can be checked on the normal form to determine validity. Equivalence of two PROV instances can be determined by comparing their normal forms. For PROV documents, validity and equivalence amount to checking the validity or pairwise equivalence of their respective instances.

This specification defines validity and equivalence procedurally, via an algorithm based on normalization. Applications MAY implement validity and equivalence checking using normalization, as outlined here. Applications MAY also implement validation and equivalence checking in any other way as long as the same instances or documents are considered valid or equivalent, respectively.

Checking validity or equivalence are RECOMMENDED, but not required, for applications compliant with PROV. Applications producing provenance SHOULD ensure that it is valid, and similarly applications consuming provenance MAY reject provenance that is not valid. Applications that are determining whether PROV instances or documents convey the same information SHOULD check equivalence as specified here. As a guideline, applications should treat equivalent instances or documents in the same way. This is a guideline only, because meaning of "in the same way" is application-specific. For example, applications that manipulate the syntax of PROV instances in particular representations, such as pretty-printing or digital signing, have good reasons to treat syntactically different, but equivalent, documents differently.

1.3 Structure of this document

Section 2 gives a brief rationale for the definitions, inferences and constraints.

Section 3 summarizes the requirements for compliance with this document, which are specified in detail in the rest of the document.

Section 4 defines basic concepts used in the rest of the specification.

Section 5 presents definitions and inferences. Definitions allow replacing shorthand notation in [PROV-N] with more explicit and complete statements; inferences allow adding new facts representing implicit knowledge about the structure of provenance.

Section 6 presents four kinds of constraints, *uniqueness* constraints that prescribe that certain statements must be unique within PROV instances, *event ordering* constraints that require that the records in a PROV instance are consistent with a sensible ordering of events relating the activities, entities and agents involved, *impossibility* constraints that forbid certain patterns of statements in valid PROV instances, and *type* constraints that classify the types of identifiers in valid PROV instances.

Section 7 defines the notions of validity, equivalence and normalization.

1.4 Audience

The audience for this document is the same as for [PROV-DM]: developers and users who wish to create, process, share or integrate provenance records on the (Semantic) Web. Not all PROV-compliant applications need to perform inferences or check validity when processing provenance. However, applications that create or transform provenance SHOULD attempt to produce valid provenance, to make it more useful to other applications by ruling out nonsensical or inconsistent information.

This document assumes familiarity with [PROV-DM] and employs the [PROV-N] notation.

2. Rationale (Informative)

This section is non-normative.

This section gives a high-level rationale that provides some further background for the constraints, but does not affect the technical content of the rest of the specification.

2.1 Entities, Activities and Agents

This section is non-normative.

One of the central challenges in representing provenance information is how to deal with change. Real-world objects, information objects and Web resources change over time, and the characteristics that make them identifiable in a given situation are sometimes subject to change as well. PROV allows for things to be described in different ways, with different descriptions of their state.

An entity is a thing one wants to provide provenance for and whose situation in the world is described by some fixed attributes. An entity has a ***lifetime***, defined as the period between its generation event and its invalidation event. An entity's attributes are established when the entity is created and (partially) describe the entity's situation and state during the entirety of the entity's lifetime.

A different entity (perhaps representing a different user or system perspective) may fix other aspects of the same thing, and its provenance may be different. Different entities that fix aspects of the same thing are called *alternates*, and the PROV relations of `specializationOf` and `alternateOf` can be used to link such entities.

Besides entities, a variety of other PROV objects and relationships carry attributes, including activity, generation, usage, invalidation, start, end, communication, attribution, association, delegation, and derivation. Each object has an associated duration interval (which may be a single time point), and attribute-value pairs for a given object are expected to be descriptions that hold for the object's duration.

However, the attributes of entities have special meaning because they are considered to be fixed aspects of underlying, changing things. This motivates constraints on `alternateOf` and `specializationOf` relating the attribute values of different entities.

In order to describe the provenance of something during an interval over which relevant attributes of the thing are not fixed, a PROV instance would describe multiple entities, each with its own identifier, lifetime, and fixed attributes, and express dependencies between the various entities using *events*. For example, in order to describe the provenance of several versions of a document, involving attributes such as authorship that change over time, one can use different entities for the versions linked by appropriate generation, usage, revision, and invalidation events.

There is no assumption that the set of attributes listed in an `entity` statement is complete, nor that the attributes are independent or orthogonal of each other. Similarly, there is no assumption that the attributes of an entity uniquely identify it. Two different entities that present the same aspects of possibly different things can have the same attributes; this leads to potential ambiguity, which is mitigated through the use of identifiers.

An activity's lifetime is delimited by its start and its end events. It occurs over an interval delimited by two instantaneous events. However, an activity statement need not mention start or end time information, because they may not be known. An activity's attribute-value pairs are expected to describe the activity's situation during its lifetime.

An activity is not an entity. Indeed, an entity exists in full at any point in its lifetime, persists during this interval, and preserves the characteristics provided. In contrast, an activity is something that occurs, happens, unfolds, or develops through time. This distinction is similar to the distinction between 'continuant' and 'occurrent' in logic [Logic].

2.2 Events

This section is non-normative.

Although time is important for provenance, provenance can be used in many different contexts within individual systems and across the Web. Different systems may use different clocks which may not be precisely synchronized, so when provenance statements are combined by different systems, an application may not be able to align the times involved to a single global timeline. Hence, PROV is designed to minimize assumptions about time. Instead, PROV talks about (identified) events.

The PROV data model is implicitly based on a notion of ***instantaneous event***s (or just events), that mark transitions in the world. Events include generation, usage, or invalidation of entities, as well as start or end of activities. This notion of event is not first-class in the data model, but it is useful for explaining its other concepts and its semantics [PROV-SEM]. Thus, events help justify *inferences* on provenance as well as *validity* constraints indicating when provenance is self-consistent.

Five kinds of instantaneous events are used in PROV. The **activity start** and **activity end** events delimit the beginning and the end of activities, respectively. The **entity generation**, **entity usage**, and **entity invalidation** events apply to entities, and the generation and invalidation events delimit the lifetime of an entity. More precisely:

An ***activity start event*** is the instantaneous event that marks the instant an activity starts.

An ***activity end event*** is the instantaneous event that marks the instant an activity ends.

An ***entity generation event*** is the instantaneous event that marks the final instant of an entity's creation timespan, after which it is available for use. The entity did not exist before this event.

An ***entity usage event*** is the instantaneous event that marks the first instant of an entity's consumption timespan by an activity. The described usage had not started before this instant, although the activity could potentially have used the same entity at a different time.

An ***entity invalidation event*** is the instantaneous event that marks the initial instant of the destruction, invalidation, or cessation of an entity, after which the entity is no longer available for use. The entity no longer exists after this event.

2.3 Types

This section is non-normative.

As set out in other specifications, the identifiers used in PROV documents have associated type information. An identifier can have more than one type, reflecting subtyping or allowed overlap between types, and so we define a set of types of each identifier, `typeOf(id)`. Some types are, however, required not to overlap (for example, no identifier can describe both an entity and an activity). In addition, an identifier cannot be used to identify both an object (that is, an entity, activity or agent) and a property (that is, a named event such as usage, generation, or a relationship such as attribution.) This specification includes disjointness and typing constraints that check these requirements. Here, we summarize the type constraints in Table 1.

Table 1: Summary of Typing Constraints

In relation...	identifier	has type(s)...
`entity(e,attrs)`	`e`	`'entity'`
`activity(a,t1,t2,attrs)`	`a`	`'activity'`
`agent(ag,attrs)`	`ag`	`'agent'`
`used(id; a,e,t,attrs)`	`e`	`'entity'`
	`a`	`'activity'`
`wasGeneratedBy(id; e,a,t,attrs)`	`e`	`'entity'`
	`a`	`'activity'`
`wasInformedBy(id; a2,a1,attrs)`	`a2`	`'activity'`
	`a1`	`'activity'`
`wasStartedBy(id; a2,e,a1,t,attrs)`	`a2`	`'activity'`
	`e`	`'entity'`
	`a1`	`'activity'`
`wasEndedBy(id; a2,e,a1,t,attrs)`	`a2`	`'activity'`
	`e`	`'entity'`
	`a1`	`'activity'`
`wasInvalidatedBy(id; e,a,t,attrs)`	`e`	`'entity'`
	`a`	`'activity'`
`wasDerivedFrom(id; e2,e1,a,g,u,attrs)`	`e2`	`'entity'`
	`e1`	`'entity'`
	`a`	`'activity'`
`wasAttributedTo(id; e,ag,attr)`	`e`	`'entity'`
	`ag`	`'agent'`
`wasAssociatedWith(id; a,ag,pl,attrs)`	`a`	`'activity'`
	`ag`	`'agent'`

	pl	'entity'
actedOnBehalfOf(id; ag2,ag1,a,attrs)	ag2	'agent'
	ag1	'agent'
	a	'activity'
alternateOf(e1,e2)	e1	'entity'
	e2	'entity'
specializationOf(e1,e2)	e1	'entity'
	e2	'entity'
hadMember(c,e)	c	'entity' 'prov:Collection'
	e	'entity'
entity(c,[prov:type='prov:EmptyCollection,...])	c	'entity' 'prov:Collection' 'prov:EmptyCollection'

2.4 Validation Process Overview

This section is non-normative.

This section collects common concepts and operations that are used throughout the specification, and relates them to background terminology and ideas from logic [Logic], constraint programming [CHR], and database constraints [DBCONSTRAINTS]. This section does not attempt to provide a complete introduction to these topics, but it is provided in order to aid readers familiar with one or more of these topics in understanding the specification, and to clarify some of the motivations for choices in the specification to all readers.

As discussed below, the definitions, inferences and constraints can be viewed as pure logical assertions that could be checked in a variety of ways. The rest of this document specifies validity and equivalence procedurally, that is, in terms of a reference implementation based on normalization. Although both declarative and procedural specification techniques have advantages, a purely declarative specification offers much less guidance for implementers, while the procedural approach adopted here immediately demonstrates implementability and provides an adequate (polynomial-time) default implementation. In this section we relate the declarative meaning of formulas to their procedural meaning. [PROV-SEM] provides an alternative, declarative characterization of validity which could be used as a starting point for other implementation strategies.

Constants, Variables and Placeholders

PROV statements involve identifiers, literals, placeholders, and attribute lists. Identifiers are, according to PROV-N, expressed as qualified names which can be mapped to URIs [RFC3987]. However, in order to specify constraints over PROV instances, we also need *variables* that represent unknown identifiers, literals, or placeholders. These variables are similar to those in first-order logic [Logic]. A variable is a symbol that can be replaced by other symbols, including either other variables or constant identifiers, literals, or placeholders. In a few special cases, we also use variables for unknown attribute lists. To help distinguish identifiers and variables, we also term the former 'constant identifiers' to highlight their non-variable nature.

Several definitions and inferences conclude by saying that some objects exist such that some other formulas hold. Such an inference introduces fresh existential variables into the instance. An existential variable denotes a fixed object that exists, but its exact identity is unknown. Existential variables can stand for unknown identifiers or literal values only; we do not allow existential variables that stand for unknown attribute lists.

In particular, many occurrences of the placeholder symbol - stand for unknown objects; these are handled by expanding them to existential variables. Some placeholders, however, indicate the absence of an object, rather than an unknown object. In other words, the placeholder is overloaded, with different meanings in different places.

An expression is called a *term* if it is either a constant identifier, literal, placeholder, or variable. We write to denote an arbitrary term.

Substitution

A *substitution* is a function that maps variables to terms. Concretely, since we only need to consider substitutions of finite sets of variables, we can write substitutions as . A substitution can be *applied* to a term by replacing occurrences of with .

In addition, a substitution can be applied to an atomic formula (PROV statement) by applying it to each term, that is, . Likewise, a substitution can be applied to an instance by applying it to each atomic formula (PROV statement) in , that is, ∈.

Formulas

For the purpose of constraint checking, we view PROV statements (possibly involving existential variables) as ***formulas***. An instance is analogous to a "theory" in logic, that is, a set of formulas all thought to describe the same situation. The set can also be thought of a single, large formula: the conjunction of all of the atomic formulas.

The atomic constraints considered in this specification can be viewed as atomic formulas:

- Uniqueness constraints employ atomic equational formulas .
- Ordering constraints employ atomic precedence relations that can be thought of as binary formulas or
- Typing constraints `'type' ∈ typeOf(id)` can be represented as a atomic formulas `typeOf(id,'type')`.
- Impossibility constraints employ the conclusion `INVALID`, which is equivalent to the logical constant .

Similarly, the definitions, inferences, and constraint rules in this specification can also be viewed as logical formulas, built up out of atomic formulas, logical connectives "and" (∧), "implies" (⇒), and quantifiers "for all" (∀) and "there exists" (∃). For more background on logical formulas, see a logic textbook such as [Logic].

- A definition of the form "A IF AND ONLY IF there exists $y_1 ... y_m$ such that B_1 and ... and B_k" can be thought of as a formula ∀⇔∃∧∧, where ... are the free variables of the definition.
- An inference of the form "IF A_1 and ... and A_p THEN there exists $y_1 ... y_m$ such that B_1 and ... and B_k" can be thought of as a formula ∀∧∧⇒∃∧∧, where ... are the free variables of the inference.
- A uniqueness, ordering, or typing constraint of the form "IF $A_1 \wedge \ldots \wedge A_p$ THEN C" can be viewed as a formula ∀∧∧⇒.
- A constraint of the form "IF $A_1 \wedge \ldots \wedge A_p$ THEN INVALID" can be viewed as a formula ∀∧∧⇒.

Satisfying definitions, inferences, and constraints

In logic, a formula's meaning is defined by saying when it is *satisfied*. We can view definitions, inferences, and constraints as being satisfied or not satisfied in a PROV instance, augmented with information about the constraints.

1. A logical equivalence as used in a definition is satisfied when the formula ∀⇔∃∧∧ holds, that is, for any substitution of the variables , formula and formula ∃∧∧ are either both true or both false.
2. A logical implication as used in an inference is satisfied when the formula ∀∧∧⇒∃∧∧ holds, that is, for any substitution of the variables , if ∧∧ is true, then for some further substitution of terms for variables , formula ∧∧ is also true.
3. A uniqueness, ordering, or typing constraint is satisfied when its associated formula ∀∧∧⇒ holds, that is, for any substitution of the variables , if ∧∧ is true, then is also true.
4. An impossibility constraint is satisfied when the formula ∀∧∧⇒ holds. This is logically equivalent to ∄∧∧, that is, there exists no substitution for making ∧∧ true.

Unification and Merging

Unification is an operation that takes two terms and compares them to determine whether they can be

made equal by substituting an existential variable with another term. If so, the result is such a substitution; otherwise, the result is failure. Unification is an essential concept in logic programming and automated reasoning, where terms can involve variables, constants and function symbols. In PROV, by comparison, unification only needs to deal with variables, constants and literals.

Unifying two terms results in either substitution such that , or failure indicating that there is no substitution that can be applied to both and to make them equal. Unification is also used to define an operation on PROV statements called *merging*. Merging takes two statements that have equal identifiers, unifies their corresponding term arguments, and combines their attribute lists.

Applying definitions, inferences, and constraints

Formulas can also be interpreted as having computational content. That is, if an instance does not satisfy a formula, we can often *apply* the formula to the instance to produce another instance that does satisfy the formula. Definitions, inferences, and uniqueness constraints can be applied to instances:

- A definition of the form ∀⇔∃∧∧ can be applied by searching for any occurrences of in the instance and adding , generating fresh existential variables , and conversely, whenever there is an occurrence of , adding . In our setting, the defined formulas are never used in other formulas, so it is sufficient to replace all occurrences of with their definitions. The formula is then redundant, and can be removed from the instance.
- An inference of the form ∀∧∧⇒∃∧∧ can be applied by searching for any occurrences of ∧∧ in the instance and, for each such match for which the entire conclusion does not already hold (for some), adding ∧∧ to the instance, generating fresh existential variables .
- A uniqueness constraint of the form ∀∧∧⇒ can be applied by searching for an occurrence ∧∧ in the instance, and if one is found, unifying the terms and . If successful, the resulting substitution is applied to the instance; otherwise, the application of the uniqueness constraint fails.
- A key constraint can similarly be applied by searching for different occurrences of a statement with the same identifier, unifying the corresponding parameters of the statements, and concatenating their attribute lists, to form a single statement. The substitutions obtained by unification are applied to the merged statement and the rest of the instance.

As noted above, uniqueness or key constraint application can *fail*, if a required unification or merging step fails. Failure of constraint application means that there is no way to add information to the instance to satisfy the constraint, which in turn implies that the instance is *invalid*.

The process of applying definitions, inferences, and constraints to a PROV instance until all of them are satisfied is similar to what is sometimes called *chasing* [DBCONSTRAINTS] or *saturation* [CHR]. We call this process *normalization*.

Although this specification outlines one particular way of performing inferences and checking constraints, based on normalization, implementations can use any other equivalent algorithm. The logical formulas corresponding to the definitions, inferences, and constraints outlined above (and further elaborated in [PROV-SEM]) provides an equivalent specification, and any implementation that correctly checks validity and equivalence (whether it performs normalization or not) complies with this specification.

Termination

In general, applying sets of logical formulas of the above definition, inference, and constraint forms is not guaranteed to terminate. A simple example is the inference ⇒∃∧, which can be applied to to generate an infinite sequence of larger and larger instances. To ensure that normalization, validity, and equivalence are decidable, we require that normalization terminates. There is a great deal of work on termination of the chase in databases, or of sets of constraint handling rules. The termination of the notion of normalization defined in this specification is guaranteed because the definitions, inferences and uniqueness/key constraints correspond to a *weakly acyclic* set of tuple-generating and equality-generating dependencies, in the terminology of [DBCONSTRAINTS]. The termination of the remaining ordering, typing, and impossibility constraints is easy to show. Appendix A gives a proof that the definitions, inferences, and uniqueness and key constraints are weakly acyclic and therefore terminating.

There is an important subtlety that is essential to guarantee termination. This specification draws a distinction between knowing that an identifier has type `'entity'`, `'activity'`, or `'agent'`, and having an

explicit `entity(id)`, `activity(id)`, or `agent(id)` statement in the instance. For example, focusing on entity statements, we can infer `'entity' ∈ typeOf(id)` if `entity(id)` holds in the instance. In contrast, if we only know that `'entity' ∈ typeOf(id)`, this does not imply that `entity(id)` holds.

This distinction (for both entities and activities) is essential to ensure termination of the inferences, because we allow inferring that a declared `entity(id,attrs)` has a generation and invalidation event, using Inference 7 (entity-generation-invalidation-inference). Likewise, for activities, we allow inferring that a declared `activity(id,t1,t2,attrs)` has a generation and invalidation event, using Inference 8 (activity-start-end-inference). These inferences do not apply to identifiers whose types are known, but for which there is not an explicit entity or activity statement. If we strengthened the type inference constraints to add new entity or activity statements for the entities and activities involved in generating or starting other declared entities or activities, then we could keep generating new entities and activities in an unbounded chain into the past (as in the "chicken and egg" paradox). The design adopted here requires that instances explicitly declare the entities and activities that are relevant for validity checking, and only these can be inferred to have invalidation/generation and start/end events. This inference is not supported for identifiers that are indirectly referenced in other relations and therefore have type `'entity'` or `'activity'`.

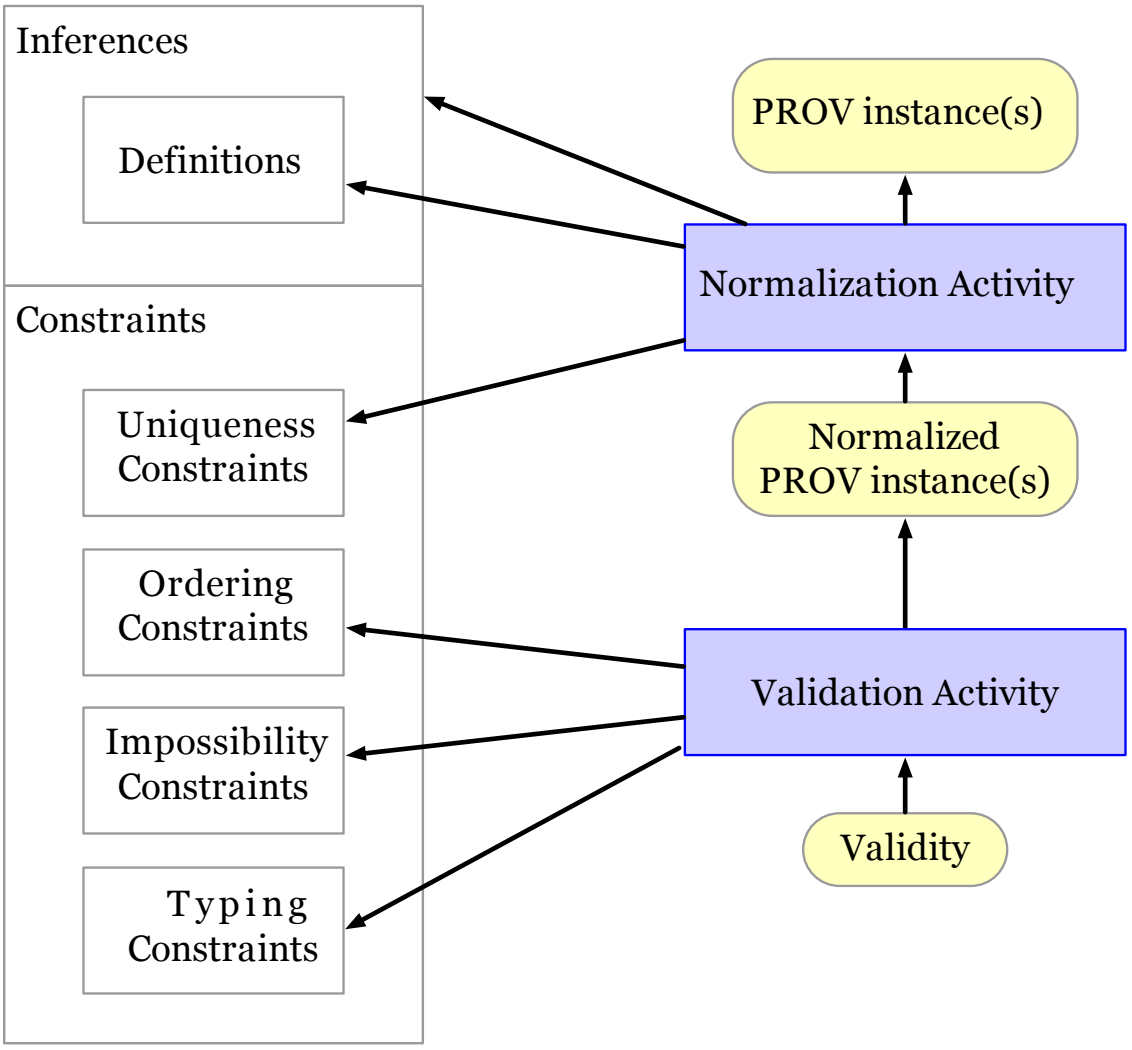


Figure 1 ◊: Overview of the Validation Process

Checking ordering, typing, and impossibility constraints

The ordering, typing, and impossibility constraints are checked rather than applied. This means that they do not generate new formulas expressible in PROV, but they do generate basic constraints that might or might not be consistent with each other. Checking such constraints follows a saturation strategy similar to that for normalization:

1. For ordering constraints, we check by generating all of the precedes and strictly-precedes

relationships specified by the rules. These can be thought of as a directed graph whose nodes are terms, and whose edges are precedes or strictly-precedes relationships. An ordering constraint of the form ∀∧∧⇒ can be applied by searching for occurrences of ∧∧ and for each such match adding the atomic formula to the instance, and similarly for strictly-precedes constraints. After all such constraints have been checked, and the resulting edges added to the graph, the ordering constraints are violated if there is a cycle in the graph that includes a strictly-precedes edge, and satisfied otherwise.

2. For typing constraints, we check by constructing a function mapping identifiers to sets of possible types. We start with a function mapping each identifier to the empty set, reflecting no constraints on the identifiers' types. A typing constraint of the form ∀∧∧⇒∈is checked by adjusting the function by adding `'type'` to for each conclusion `'type' ∈ typeOf(id)` of the rule. Typing constraints with multiple conclusions are handled analogously. Once all constraints have been checked in all possible ways, we check that the disjointness constraints hold of the resulting function. (These are essentially impossibility constraints).

3. For impossibility constraints, we check by searching for the forbidden pattern that the impossibility constraint describes. Any match of this pattern leads to failure of the constraint checking process. An impossibility constraint of the form ∀∧∧⇒ can be applied by searching for occurrences of ∧∧ in the instance, and if any such occurrence is found, signaling failure.

A normalized instance that passes all of the ordering, typing, and impossibility constraint checks is called valid. Validity can be, but is not required to be, checked by normalizing and then checking constraints. Any other algorithm that provides equivalent behavior (that is, accepts the same valid instances and rejects the same invalid instances) is allowed. In particular, the checked constraints and the applied definitions, inferences and uniqueness constraints do not interfere with one another, so it is also possible to mix checking and application. This may be desirable in order to detect invalidity more quickly.

Equivalence and Isomorphism

Given two normal forms, a natural question is whether they contain the same information, that is, whether they are equivalent (if so, then the original instances are also equivalent.) By analogy with logic, if we consider normalized PROV instances with existential variables to represent sets of possible situations, then two normal forms may describe the same situation but differ in inessential details such as the order of statements or of elements of attribute-value lists. To remedy this, we can easily consider instances to be equivalent up to reordering of attributes. However, instances can also be equivalent if they differ only in choice of names of existential variables. Because of this, the appropriate notion of equivalence of normal forms is *isomorphism*. Two instances and are *isomorphic* if there is an invertible substitution mapping existential variables to existential variables such that .

Equivalence can be checked by normalizing instances, checking that both instances are valid, then testing whether the two normal forms are isomorphic. (It is technically possible for two invalid normal forms to be isomorphic, but to be considered equivalent, the two instances must also be valid.) As with validity, the algorithm suggested by this specification is just one of many possible ways to implement equivalence checking; it is not required that implementations compute normal forms explicitly, only that their determinations of equivalence match those obtained by the algorithm in this specification.

Equivalence is only explicitly specified for valid instances (whose normal forms exist and are unique up to isomorphism). Implementations may test equivalences involving valid and invalid documents. This specification does not constrain the behavior of equivalence checking involving invalid instances, provided that:

- instance equivalence is reflexive, symmetric and transitive on all instances
- no valid instance is equivalent to an invalid instance.

Because of the second constraint, equivalence is essentially the union of two equivalence relations on the disjoint sets of valid and invalid instances. There are two simple implementations of equivalence for invalid documents that are correct:

1. each invalid instance is equivalent only to itself
2. every pair of invalid instances are equivalent

From Instances to Bundles and Documents

PROV documents can contain multiple instances: a toplevel instance, and zero or more additional, named instances called bundles. For the purpose of inference and constraint checking, these instances are treated independently. That is, a PROV document is valid provided that each instance in it is valid and the names of its bundles are distinct. In other words, there are no validity constraints that need to be checked across the different instances in a PROV document; the contents of one instance in a multi-instance PROV document cannot affect the validity of another instance. Similarly, a PROV document is equivalent to another if their toplevel instances are equivalent, they have the same number of bundles with the same names, and the instances of their corresponding bundles are equivalent. The scope of an existential variable in PROV is delimited at the instance level. This means that occurrences of existential variables with the same name appearing in different statements within the same instance stand for a common, unknown term. However, existential variables with the same name occurring in different instances do not necessarily denote the same term. This is a consequence of the fact that the instances of two equivalent documents only need to be pairwise isomorphic; this is a weaker property than requiring that there be a single isomorphism that works for all of the corresponding instances.

2.5 Summary of inferences and constraints

This section is non-normative.

Table 2 summarizes the inferences, and constraints specified in this document, broken down by component and type or relation involved.

Table 2: Summary of inferences and constraints for PROV Types and Relations

Type or Relation Name	Inferences and Constraints	Component
Entity	Inference 7 (entity-generation-invalidation-inference) Inference 21 (specialization-attributes-inference) Constraint 22 (key-object) Constraint 54 (impossible-object-property-overlap) Constraint 55 (entity-activity-disjoint)	
Activity	Inference 8 (activity-start-end-inference) Constraint 22 (key-object) Constraint 28 (unique-startTime) Constraint 29 (unique-endTime) Constraint 54 (impossible-object-property-overlap) Constraint 55 (entity-activity-disjoint)	
Generation	Inference 6 (generation-use-communication-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 24 (unique-generation) Constraint 34 (generation-within-activity) Constraint 36 (generation-precedes-invalidation) Constraint 37 (generation-precedes-usage) Constraint 39 (generation-generation-ordering) Constraint 41 (derivation-usage-generation-ordering) Constraint 42 (derivation-generation-generation-ordering) Constraint 43 (wasStartedBy-ordering) Constraint 44 (wasEndedBy-ordering) Constraint 45 (specialization-generation-ordering) Constraint 47 (wasAssociatedWith-ordering) Constraint 48 (wasAttributedTo-ordering) Constraint 49 (actedOnBehalfOf-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
Usage	Inference 6 (generation-use-communication-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 33 (usage-within-activity) Constraint 37 (generation-precedes-usage) Constraint 38 (usage-precedes-invalidation)	

	Constraint 41 (derivation-usage-generation-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	1
Communication	Inference 5 (communication-generation-use-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 35 (wasInformedBy-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
Start	Inference 9 (wasStartedBy-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 26 (unique-wasStartedBy) Constraint 28 (unique-startTime) Constraint 30 (start-precedes-end) Constraint 33 (usage-within-activity) Constraint 34 (generation-within-activity) Constraint 35 (wasInformedBy-ordering) Constraint 31 (start-start-ordering) Constraint 43 (wasStartedBy-ordering) Constraint 47 (wasAssociatedWith-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
End	Inference 10 (wasEndedBy-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 27 (unique-wasEndedBy) Constraint 29 (unique-endTime) Constraint 30 (start-precedes-end) Constraint 33 (usage-within-activity) Constraint 34 (generation-within-activity) Constraint 35 (wasInformedBy-ordering) Constraint 32 (end-end-ordering) Constraint 44 (wasEndedBy-ordering) Constraint 47 (wasAssociatedWith-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
Invalidation	Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 25 (unique-invalidation) Constraint 36 (generation-precedes-invalidation) Constraint 38 (usage-precedes-invalidation) Constraint 40 (invalidation-invalidation-ordering) Constraint 43 (wasStartedBy-ordering) Constraint 44 (wasEndedBy-ordering) Constraint 46 (specialization-invalidation-ordering) Constraint 47 (wasAssociatedWith-ordering) Constraint 48 (wasAttributedTo-ordering) Constraint 49 (actedOnBehalfOf-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
Derivation	Inference 11 (derivation-generation-use-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 41 (derivation-usage-generation-ordering) Constraint 42 (derivation-generation-generation-ordering) Constraint 50 (typing)	2
Revision	Inference 12 (revision-is-alternate-inference)	
Quotation	No specific constraints	
Primary Source	No specific constraints	

Influence	No specific constraints	
Agent	Constraint 22 (key-object) Constraint 54 (impossible-object-property-overlap)	3
Attribution	Inference 13 (attribution-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 48 (wasAttributedTo-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
Association	Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 47 (wasAssociatedWith-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
Delegation	Inference 14 (delegation-inference) Inference 15 (influence-inference) Constraint 23 (key-properties) Constraint 49 (actedOnBehalfOf-ordering) Constraint 53 (impossible-property-overlap) Constraint 50 (typing)	
Influence	Inference 15 (influence-inference) Constraint 23 (key-properties)	
Bundle constructor	No specific constraints; see section 7.2 Bundles and Documents	4
Bundle type	No specific constraints; see section 7.2 Bundles and Documents	
Alternate	Inference 16 (alternate-reflexive) Inference 17 (alternate-transitive) Inference 18 (alternate-symmetric) Constraint 50 (typing)	5
Specialization	Inference 19 (specialization-transitive) Inference 20 (specialization-alternate-inference) Inference 21 (specialization-attributes-inference) Constraint 45 (specialization-generation-ordering) Constraint 46 (specialization-invalidation-ordering) Constraint 52 (impossible-specialization-reflexive) Constraint 50 (typing)	
Collection	No specific constraints	6
Membership	Constraint 56 (membership-empty-collection) Constraint 50 (typing)	

3. Compliance with this document

For the purpose of compliance, the normative sections of this document are section 3. Compliance with this document, section 4. Basic concepts, section 5. Definitions and Inferences, section 6. Constraints, and section 7. Normalization, Validity, and Equivalence. To be compliant:

1. When processing provenance, an application MAY apply the inferences and definitions in section 5. Definitions and Inferences.
2. If determining whether a PROV instance or document is valid, an application MUST determine whether all of the constraints of section 6. Constraints are satisfied on the normal form of the instance or document.
3. If producing provenance meant for other applications to use, the application SHOULD produce valid provenance, as specified in section 7. Normalization, Validity, and Equivalence.
4. If determining whether two PROV instances or documents are equivalent, an application MUST determine whether their normal forms are equal, as specified in section 7. Normalization, Validity, and Equivalence.

This specification defines validity and equivalence procedurally via reduction to normal forms. If checking validity or equivalence, the results MUST be the same as would be obtained by computing normal forms as

defined in this specification. Applications that explicitly compute normal forms, following the implementation strategy suggested by this specification, are by definition compliant. However, applications can also comply by checking validity and equivalence in any other way that yields the same answers without explicitly applying definitions, inferences, and constraints or constructing normal forms.

All figures are for illustration purposes only. Information in tables is normative if it appears in a normative section; specifically, Table 3 is normative. Text in appendices and in boxes labeled "Remark" is informative. Where there is any apparent ambiguity between the descriptive text and the formal text in a "definition", "inference" or "constraint" box, the formal text takes priority.

4. Basic concepts

This section specifies the key concepts of terms, statements, instances, substitution, satisfaction, and unification, which have already been discussed in Section 2.

Many PROV relation statements have an identifier, identifying a link between two or more related objects. Identifiers can sometimes be omitted in [PROV-N] notation. For the purpose of inference and validity checking, we generate special identifiers called ***variables*** denoting the unknown values. Generally, identifiers occurring in constraints and inferences are variables. Variables that are generated during inferences and appear inside an instance are often called ***existential variables***, because they are implicitly existentially quantified.

A ***PROV term*** is a constant identifier , a placeholder –, a literal value, or an existential variable . An arbitrary PROV term is written .

A ***PROV statement*** is an expression of the form or where are PROV terms and is one of the basic PROV relations. An arbitrary PROV statement is written .

A ***PROV instance*** is a set of PROV statements. Two instances are considered to be the same if they contain the same statements, without regard to order or repetition. An arbitrary PROV instance is written .

A ***substitution*** is a mapping associating existential variables with terms. A substitution is *applied* to a term, statement or instance by replacing all occurrences of each of the variables with the corresponding . Specifically, if then the application of to a term, statement or instance, written , and respectively, is defined as follows:

- if is a constant identifier.
- if is one of the variables bound to a term in .
- if is a variable not bound in .
- .
- .
- ∈if is an instance.

Suppose is a statement and is an instance and a substitution. We say that is ***satisfied*** in by if ∈. Likewise, we say that a set of statements is satisfied in if each is satisfied in by . Finally, we say that a set of statements is ***satisfiable*** in if there is some substitution that satisfies the statements in .

Unification is an operation that can be applied to a pair of terms. The result of unification is either a ***unifier***, that is, a substitution such that , or failure, indicating that there is no unifier. Unification of pairs of terms is defined as follows.

- If and are constant identifiers or literal values (including the placeholder –), then there are two cases. If then their unifier is the empty substitution, otherwise unification fails.
- If is an existential variable and is any term (identifier, constant, placeholder –, or existential variable), then their unifier is . In the special case where , the unifier is the empty substitution.
- If is any term (identifier, constant, placeholder –, or existential variable) and is an existential variable, then their unifier is the same as the unifier of and .

Remark

Unification is analogous to unification in logic programming and theorem proving, restricted to flat terms with constants and variables but no function symbols. No "occurs check" is needed because

there are no function symbols.

Two PROV instances and are ***isomorphic*** if there exists an invertible substitution that maps each variable of to a distinct variable of and such that .

Remark

The scope of an existential variable is at the instance level. When we obtain information about an existential variable, for example through unification or merging during uniqueness constraint application, we substitute all other occurrences of that variable occurring in the same instance. Occurrences in other instances are not affected.

5. Definitions and Inferences

This section describes definitions and inferences that MAY be used on provenance data, and that preserve equivalence on valid PROV instances (as detailed in section 7. Normalization, Validity, and Equivalence). A ***definition*** is a rule that can be applied to PROV instances to replace defined statements with other statements. An ***inference*** is a rule that can be applied to PROV instances to add new PROV statements. A definition states that a provenance statement is equivalent to some other statements, whereas an inference only states one direction of an implication.

Definitions have the following general form:

Definition-example NNN (definition-example)

`defined_stmt` IF AND ONLY IF there exists a_1,..., a_m such that `defining_stmt`$_1$ and ... and `defining_stmt`$_n$.

A definition can be applied to a PROV instance, since its `defined_stmt` is defined in terms of other statements. Applying a definition to an instance means that if an occurrence of a defined provenance statement `defined_stmt` can be found in a PROV instance, then we can remove it and add all of the statements `defining_stmt`$_1$ ... `defining_stmt`$_n$ to the instance, possibly after generating fresh identifiers a_1,...,a_m for existential variables. In other words, it is safe to replace a defined statement with its definition.

Remark

We use definitions primarily to expand the compact, concrete PROV-N syntax, including short forms and optional parameters, to the abstract syntax implicitly used in PROV-DM.

Inferences have the following general form:

Inference-example NNN (inference-example)

IF `hyp`$_1$ and ... and `hyp`$_k$ THEN there exists a_1 and ... and a_m such that `concl`$_1$ and ... and `concl`$_n$.

Inferences can be applied to PROV instances. Applying an inference to an instance means that if all of the provenance statements matching `hyp`$_1$... `hyp`$_k$ can be found in the instance, then we check whether the conclusion `concl`$_1$ ... `concl`$_n$ is satisfied for some values of existential variables. If so, application of the inference has no effect on the instance. If not, then a copy the conclusion should be added to the instance, after generating fresh identifiers a_1,...,a_m for the existential variables. These fresh identifiers might later be found to be equal to known identifiers; they play a similar role in PROV constraints to existential variables in logic [Logic] or database theory [DBCONSTRAINTS]. In general, omitted optional parameters to [PROV-N] statements, or explicit - markers, are placeholders for existentially quantified variables; that is, they denote unknown values. There are a few exceptions to this general rule, which are specified in Definition 4 (optional-placeholders).

Definitions and inferences can be viewed as logical formulas; similar formalisms are often used in rule-based reasoning [CHR] and in databases [DBCONSTRAINTS]. In particular, the identifiers a_1 ... a_n should be viewed as existentially quantified variables, meaning that through subsequent reasoning steps they may turn out to be equal to other identifiers that are already known, or to other existentially quantified variables. In contrast, distinct URIs or literal values in PROV are assumed to be distinct for the purpose of checking validity or inferences. This issue is discussed in more detail under Uniqueness Constraints.

In a definition or inference, term symbols such as `id`, `start`, `end`, `e`, `a`, `attrs`, are assumed to be variables unless otherwise specified. These variables are scoped at the definition, inference, or constraint level, so the rule is equivalent to any one-for-one renaming of the variable names. When several rules are collected within a definition or inference as an ordered list, the scope of the variables in each rule is at the level of list elements, and so reuse of variable names in different rules does not affect the meaning.

5.1 Optional Identifiers and Attributes

Definition 1 (optional-identifiers), Definition 2 (optional-attributes), and Definition 3 (definition-short-forms), explain how to expand the compact forms of PROV-N notation into a normal form. Definition 4 (optional-placeholders) indicates when other optional parameters can be replaced by existential variables.

Definition 1 (optional-identifiers)

For each `r` in { `used`, `wasGeneratedBy`, `wasInvalidatedBy`, `wasInfluencedBy`, `wasStartedBy`, `wasEndedBy`, `wasInformedBy`, `wasDerivedFrom`, `wasAttributedTo`, `wasAssociatedWith`, `actedOnBehalfOf`}, the following definitional rules hold:

1. `r(`a_1`,...,`a_n`)` IF AND ONLY IF there exists `id` such that `r(id;` a_1`,...,`a_n`)`.
2. `r(-;` a_1`,...,`a_n`)` IF AND ONLY IF there exists `id` such that `r(id;` a_1`,...,`a_n`)`.

Likewise, many PROV-N statements allow for an optional attribute list. If it is omitted, this is the same as specifying an empty attribute list:

Definition 2 (optional-attributes)

1. The following definitional rules hold:
   - `entity(id)` IF AND ONLY IF `entity(id,[])`.
   - `activity(id)` IF AND ONLY IF `activity(id,[])`.
   - `activity(id,t1,t2)` IF AND ONLY IF `activity(id,t1,t2,[])`.
   - `agent(id)` IF AND ONLY IF `agent(id,[])`.
2. For each `r` in { `used`, `wasGeneratedBy`, `wasInvalidated`, `wasInfluencedBy`, `wasStartedBy`, `wasEndedBy`, `wasInformedBy`, `wasDerivedFrom`, `wasAttributedTo`, `wasAssociatedWith`, `actedOnBehalfOf`}, if a_n is not an attribute list parameter then the following definition holds:

   `r(id;` a_1`,...,`a_n`)` IF AND ONLY IF `r(id;` a_1`,...,`a_n`,[])`.

Remark

Definitions Definition 1 (optional-identifiers) and Definition 2 (optional-attributes) do not apply to `alternateOf` and `specializationOf`, which do not have identifiers and attributes.

Finally, many PROV statements have other optional arguments or short forms that can be used if none of the optional arguments is present. These are handled by specific rules listed below.

Definition 3 (definition-short-forms)

1. `activity(id,attrs)` IF AND ONLY IF `activity(id,-,-,attrs)`.
2. `wasGeneratedBy(id; e,attrs)` IF AND ONLY IF `wasGeneratedBy(id; e,-,-,attrs)`.
3. `used(id; a,attrs)` IF AND ONLY IF `used(id; a,-,-,attrs)`.
4. `wasStartedBy(id; a,attrs)` IF AND ONLY IF `wasStartedBy(id; a,-,-,-,attrs)`.
5. `wasEndedBy(id; a,attrs)` IF AND ONLY IF `wasEndedBy(id; a,-,-,-,attrs)`.
6. `wasInvalidatedBy(id; e,attrs)` IF AND ONLY IF `wasInvalidatedBy(id; e,-,-,attrs)`.
7. `wasDerivedFrom(id; e2,e1,attrs)` IF AND ONLY IF `wasDerivedFrom(id; e2,e1,-,-,-,attrs)`.
8. `wasAssociatedWith(id; e,attrs)` IF AND ONLY IF `wasAssociatedWith(id; e,-,-,attrs)`.
9. `actedOnBehalfOf(id; a2,a1,attrs)` IF AND ONLY IF `actedOnBehalfOf(id; a2,a1,-,attrs)`.

Remark

There are no expansion rules for entity, agent, communication, attribution, influence, alternate, or specialization relations, because these have no optional parameters aside from the identifier and attributes, which are expanded by the rules in Definition 1 (optional-identifiers) and Definition 2 (optional-attributes).

Finally, most optional parameters (written `-`) are, for the purpose of this document, considered to be distinct, fresh existential variables. Optional parameters are defined in [PROV-DM] and in [PROV-N] for each type of PROV statement. Thus, before proceeding to apply other definitions or inferences, most occurrences of `-` are to be replaced by fresh existential variables, distinct from any others occurring in the instance. The only exceptions to this general rule, where `-` are to be left in place, are the activity, generation, and usage parameters in `wasDerivedFrom` and the plan parameter in `wasAssociatedWith`. This is further explained in remarks below.

The treatment of optional parameters is specified formally using the auxiliary concept of ***expandable parameter***. An expandable parameter is one that can be omitted using the placeholder `-`, and if so, it is to be replaced by a fresh existential identifier. Table 3 defines the expandable parameters of the properties of PROV, needed in Definition 4 (optional-placeholders). For emphasis, the four optional parameters that are not expandable are also listed. Parameters that cannot have value `-`, and identifiers that are expanded by Definition 1 (optional-identifiers), are not listed.

Table 3: Expandable and Non-Expandable Parameters

Relation	Expandable	Non-expandable
`used(id; a,e,t,attrs)`	`e,t`	
`wasGeneratedBy(id; e,a,t,attrs)`	`a,t`	
`wasStartedBy(id; a2,e,a1,t,attrs)`	`e,a1,t`	
`wasEndedBy(id; a2,e,a1,t,attrs)`	`e,a1,t`	
`wasInvalidatedBy(id; e,a,t,attrs)`	`a,t`	
`wasDerivedFrom(id; e2,e1,-,g,u,attrs)`		`g,u`
`wasDerivedFrom(id; e2,e1,a,g,u,attrs)` (where a is not placeholder -)	`g,u`	`a`
`wasAssociatedWith(id; a,ag,pl,attrs)`	`ag`	`pl`
`actedOnBehalfOf(id; ag2,ag1,a,attrs)`	`a`	

Definition 4 (optional-placeholders) states how parameters are to be expanded, using the expandable parameters defined in Table 3. The last two parts, 4 and 5, indicate how to handle expansion of parameters for `wasDerivedFrom` expansion, which is only allowed for the generation and use parameters when the activity is specified. Essentially, the definitions state that parameters `g,u` are expandable only if the activity is specified, i.e., if parameter `a` is provided. The rationale for this is that when `a` is provided, then there have to be two events, namely `u` and `g`, which account for the usage of `e1` and the generation of `e2`, respectively, by `a`. Conversely, if `a` is not provided, then one cannot tell whether one or more activities are involved in the derivation, and the explicit introduction of such events, which correspond to a single activity, would therefore not be justified.

A later constraint, Constraint 51 (impossible-unspecified-derivation-generation-use), forbids specifying generation and use parameters when the activity is unspecified.

Definition 4 (optional-placeholders)

1. `activity(id,-,t2,attrs)` IF AND ONLY IF there exists `t1` such that `activity(id,t1,t2,attrs)`. Here, `t2` MAY be a placeholder.
2. `activity(id,t1,-,attrs)` IF AND ONLY IF there exists `t2` such that `activity(id,t1,t2,attrs)`. Here, `t1` MAY be a placeholder.
3. For each `r` in { `used`, `wasGeneratedBy`, `wasStartedBy`, `wasEndedBy`, `wasInvalidatedBy`, `wasAssociatedWith`, `actedOnBehalfOf` }, if the `i`th parameter of `r` is an expandable parameter of `r` as specified in Table 3 then the following definition holds:

   $r(a_0;...,a_{i-1}, -, a_{i+1}, ...,a_n)$ IF AND ONLY IF there exists `a'` such that $r(a_0;...,a_{i-1},a',a_{i+1},...,a_n)$.

4. If `a` is not the placeholder `-`, and `u` is any term, then the following definition holds:

   `wasDerivedFrom(id; e2,e1,a,-,u,attrs)` IF AND ONLY IF there exists `g` such that `wasDerivedFrom(id; e2,e1,a,g,u,attrs)`.

5. If `a` is not the placeholder `-`, and `g` is any term, then the following definition holds:

   `wasDerivedFrom(id; e2,e1,a,g,-,attrs)` IF AND ONLY IF there exists `u` such that `wasDerivedFrom(id; e2,e1,a,g,u,attrs)`.

Remark

In an association of the form `wasAssociatedWith(id; a,ag,-,attr)`, the absence of a plan means: either no plan exists, or a plan exists but it is not identified. Thus, it is not equivalent to `wasAssociatedWith(id; a,ag,p,attr)` where a plan `p` is given.

Remark

A derivation `wasDerivedFrom(id; e2,e1,a,gen,use,attrs)` that specifies an activity explicitly indicates that this activity achieved the derivation, with a usage `use` of entity `e1`, and a generation `gen` of entity `e2`. It differs from a derivation of the form `wasDerivedFrom(id; e2,e1,-,-,-,attrs)` with missing activity, generation, and usage. In the latter form, it is not specified if one or more activities are involved in the derivation.

Let us consider a system, in which a derivation is underpinned by multiple activities. Conceptually, one could also model such a system with a new activity that encompasses the two original activities and underpins the derivation. The inferences defined in this specification do not allow the latter modeling to be inferred from the former. Hence, the two modeling of the same system are regarded as different in the context of this specification.

5.2 Entities and Activities

Communication between activities implies the existence of an underlying entity generated by one activity and used by the other, and vice versa.

Inference 5 (communication-generation-use-inference)

IF `wasInformedBy(_id; a2,a1,_attrs)` THEN there exist `e`, `_gen`, `_t1`, `_use`, and `_t2`, such that `wasGeneratedBy(_gen; e,a1,_t1,[])` and `used(_use; a2,e,_t2,[])` hold.

Inference 6 (generation-use-communication-inference)

IF `wasGeneratedBy(_gen; e,a1,_t1,_attrs1)` and `used(_use; a2,e,_t2,_attrs2)` hold THEN there exists `_id` such that `wasInformedBy(_id; a2,a1,[])`

Remark

The relationship `wasInformedBy` is not transitive. Indeed, consider the following statements.

```
wasInformedBy(a2,a1)
wasInformedBy(a3,a2)
```

We cannot infer `wasInformedBy(a3,a1)` from these statements alone. Indeed, from `wasInformedBy(a2,a1)`, we know that there exists `e1` such that `e1` was generated by `a1` and used by `a2`. Likewise, from `wasInformedBy(a3,a2)`, we know that there exists `e2` such that `e2` was generated by `a2` and used by `a3`. The following illustration shows a counterexample to transitivity. The horizontal axis represents the event line. We see that `e1` was generated after `e2` was used. Furthermore, the illustration also shows that `a3` completes before `a1` started. So in this example (with no other information) it is impossible for `a3` to have used an entity generated by `a1`. This is illustrated in Figure 2.

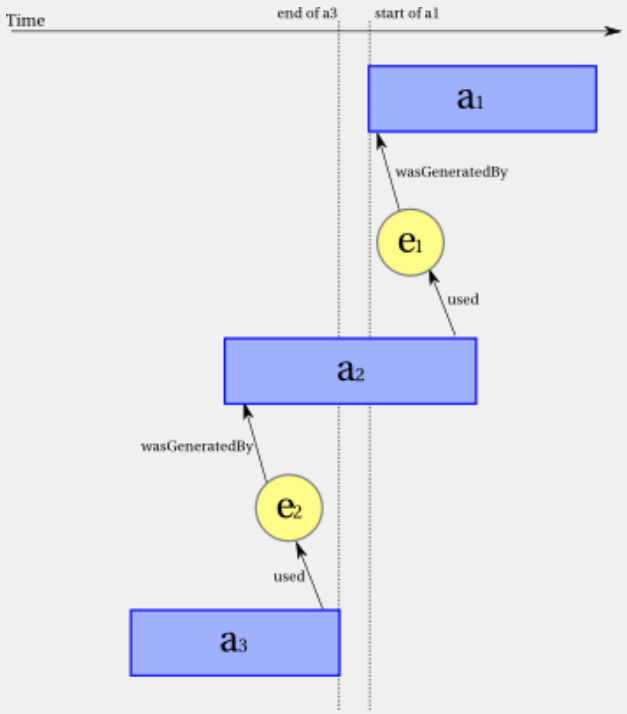


Figure 2 Counter-example for transitivity of wasInformedBy

From an entity statement, we can infer the existence of generation and invalidation events.

Inference 7 (entity-generation-invalidation-inference)

IF `entity(e,_attrs)` THEN there exist `_gen`, `_a1`, `_t1`, `_inv`, `_a2`, and `_t2` such that `wasGeneratedBy(_gen; e,_a1,_t1,[])` and `wasInvalidatedBy(_inv; e,_a2,_t2,[])`.

From an activity statement, we can infer start and end events whose times match the start and end times of the activity, respectively.

Inference 8 (activity-start-end-inference)

IF `activity(a,t1,t2,_attrs)` THEN there exist `_start`, `_e1`, `_a1`, `_end`, `_a2`, and `_e2` such that `wasStartedBy(_start; a,_e1,_a1,t1,[])` and `wasEndedBy(_end; a,_e2,_a2,t2,[])`.

The start of an activity `a` triggered by entity `e1` implies that `e1` was generated by the starting activity `a1`.

Inference 9 (wasStartedBy-inference)

IF `wasStartedBy(_id; _a,e1,a1,_t,_attrs)`, THEN there exist `_gen` and `_t1` such that `wasGeneratedBy(_gen; e1,a1,_t1,[])`.

Likewise, the ending of activity `a` by triggering entity `e1` implies that `e1` was generated by the ending activity `a1`.

Inference 10 (wasEndedBy-inference)

IF `wasEndedBy(_id; _a,e1,a1,_t,_attrs)`, THEN there exist `_gen` and `_t1` such that `wasGeneratedBy(_gen; e1,a1,_t1,[])`.

5.3 Derivations

Derivations with explicit activity, generation, and usage admit the following inference:

Inference 11 (derivation-generation-use-inference)

In this inference, none of `a`, `gen2` or `use1` can be placeholders `-`.

IF `wasDerivedFrom(_id; e2,e1,a,gen2,use1,_attrs)`, THEN there exists `_t1` and `_t2` such that `used(use1; a,e1,_t1,[])` and `wasGeneratedBy(gen2; e2,a,_t2,[])`.

A revision admits the following inference, stating that the two entities linked by a revision are also alternates.

Inference 12 (revision-is-alternate-inference)

In this inference, any of `_a`, `_g` or `_u` MAY be placeholders.

IF `wasDerivedFrom(_id; e2,e1,_a,_g,_u,[prov:type='prov:Revision'])`, THEN `alternateOf(e2,e1)`.

Remark

There is no inference stating that `wasDerivedFrom` is transitive.

5.4 Agents

Attribution is the ascribing of an entity to an agent. An entity can only be ascribed to an agent if the agent was associated with an activity that generated the entity. If the activity, generation and association events are not explicit in the instance, they can be inferred.

Inference 13 (attribution-inference)

IF `wasAttributedTo(_att; e,ag,_attrs)` THEN there exist `a`, `_t`, `_gen`, `_assoc`, `_pl`, such that

`wasGeneratedBy(_gen; e,a,_t,[])` and `wasAssociatedWith(_assoc; a,ag,_pl,[])`.

Remark

In the above inference, `_pl` is an existential variable, so it can be unified with a constant identifier, another existential variable, or a placeholder `-`, as explained in the definition of unification.

Delegation relates agents where one agent acts on behalf of another, in the context of some activity. The supervising agent delegates some responsibility for part of the activity to the subordinate agent, while retaining some responsibility for the overall activity. Both agents are associated with this activity.

Inference 14 (delegation-inference)

IF `actedOnBehalfOf(_id; ag1, ag2, a, _attrs)` THEN there exist `_id1`, `_pl1`, `_id2`, and `_pl2` such that `wasAssociatedWith(_id1; a, ag1, _pl1, [])` and `wasAssociatedWith(_id2; a, ag2, _pl2, []).`

Remark

The two associations between the agents and the activity may have different identifiers, different plans, and different attributes. In particular, the plans of the two agents need not be the same, and one, both, or neither can be the placeholder `-` indicating that there is no plan, because the existential variables `_pl1` and `_pl2` can be replaced with constant identifiers, existential variables, or placeholders `-` independently, as explained in the definition of unification.

The `wasInfluencedBy` relation is implied by other relations, including usage, start, end, generation, invalidation, communication, derivation, attribution, association, and delegation. To capture this explicitly, we allow the following inferences:

Inference 15 (influence-inference)

1. IF `wasGeneratedBy(id; e,a,_t,attrs)` THEN `wasInfluencedBy(id; e, a, attrs).`
2. IF `used(id; a,e,_t,attrs)` THEN `wasInfluencedBy(id; a, e, attrs).`
3. IF `wasInformedBy(id; a2,a1,attrs)` THEN `wasInfluencedBy(id; a2, a1, attrs).`
4. IF `wasStartedBy(id; a2,e,_a1,_t,attrs)` THEN `wasInfluencedBy(id; a2, e, attrs).`
5. IF `wasEndedBy(id; a2,e,_a1,_t,attrs)` THEN `wasInfluencedBy(id; a2, e, attrs).`
6. IF `wasInvalidatedBy(id; e,a,_t,attrs)` THEN `wasInfluencedBy(id; e, a, attrs).`
7. IF `wasDerivedFrom(id; e2, e1, _a, _g, _u, attrs)` THEN `wasInfluencedBy(id; e2, e1, attrs)`. Here, `_a`, `_g`, `_u` MAY be placeholders `-`.
8. IF `wasAttributedTo(id; e,ag,attrs)` THEN `wasInfluencedBy(id; e, ag, attrs).`
9. IF `wasAssociatedWith(id; a,ag,_pl,attrs)` THEN `wasInfluencedBy(id; a, ag, attrs).` Here, `_pl` MAY be a placeholder `-`.
10. IF `actedOnBehalfOf(id; ag2,ag1,_a,attrs)` THEN `wasInfluencedBy(id; ag2, ag1, attrs).`

Remark

The inferences above permit the use of same identifier for an influence relationship and a more specific relationship.

5.5 Alternate and Specialized Entities

The relation `alternateOf` is an equivalence relation on entities: that is, it is reflexive, transitive and symmetric. As a consequence, the following inferences can be applied:

Inference 16 (alternate-reflexive)

IF `entity(e)` THEN `alternateOf(e,e)`.

Inference 17 (alternate-transitive)

IF `alternateOf(e1,e2)` and `alternateOf(e2,e3)` THEN `alternateOf(e1,e3)`.

Inference 18 (alternate-symmetric)

IF `alternateOf(e1,e2)` THEN `alternateOf(e2,e1)`.

Similarly, specialization is a strict partial order: it is irreflexive and transitive. Irreflexivity is handled later as Constraint 52 (impossible-specialization-reflexive)

Inference 19 (specialization-transitive)

IF `specializationOf(e1,e2)` and `specializationOf(e2,e3)` THEN `specializationOf(e1,e3)`.

If one entity specializes another, then they are also alternates:

Inference 20 (specialization-alternate-inference)

IF `specializationOf(e1,e2)` THEN `alternateOf(e1,e2)`.

If one entity specializes another then all attributes of the more general entity are also attributes of the more specific one.

Inference 21 (specialization-attributes-inference)

IF `entity(e1, attrs)` and `specializationOf(e2,e1)`, THEN `entity(e2, attrs)`.

6. Constraints

This section defines a collection of constraints on PROV instances. There are three kinds of constraints:

- *uniqueness constraints* that say that a PROV instance can contain at most one statement of each kind with a given identifier. For example, if we describe the same generation event twice, then the two statements should have the same times;
- *event ordering constraints* that say that it should be possible to arrange the events (generation, usage, invalidation, start, end) described in a PROV instance into a preorder that corresponds to a sensible "history" (for example, an entity should not be generated after it is used); and
- *impossibility constraints*, which forbid certain patterns of statements in valid PROV instances.

As in a definition or inference, term symbols such as `id`, `start`, `end`, `e`, `a`, `attrs` in a constraint, are assumed to be variables unless otherwise specified. These variables are scoped at the constraint level, so

the rule is equivalent to any one-for-one renaming of the variable names. When several rules are collected within a constraint as an ordered list, the scope of the variables in each rule is at the level of list elements, and so reuse of variable names in different rules does not affect the meaning.

6.1 Uniqueness Constraints

In the absence of existential variables, uniqueness constraints could be checked directly by checking that no identifier appears more than once for a given statement. However, in the presence of existential variables, we need to be more careful to combine partial information that might be present in multiple compatible statements, due to inferences. Uniqueness constraints are enforced through merging pairs of statements subject to equalities. For example, suppose we have two activity statements `activity(a,2011-11-16T16:00:00,_t1,[a=1])` and `activity(a,_t2,2011-11-16T18:00:00,[b=2])`, with existential variables `_t1` and `_t2`. The ***merge*** of these two statements (describing the same activity `a`) is `activity(a,2011-11-16T16:00:00,2011-11-16T18:00:00,[a=1,b=2])`.

A typical uniqueness constraint is as follows:

Constraint-example NNN (uniqueness-example)

IF hyp_1 and ... and hyp_n THEN $t_1 = u_1$ and ... and $t_n = u_n$.

Such a constraint is enforced as follows:

1. Suppose PROV instance contains all of the hypotheses hyp_1 and ... and hyp_n.
2. Attempt to unify all of the equated terms in the conclusion $t_1 = u_1$ and ... and $t_n = u_n$.
3. If unification fails, then the constraint is unsatisfiable, so application of the constraint to fails. If this failure occurs during normalization prior to validation, then is invalid, as explained in Section 6.
4. If unification succeeds with a substitution , then is applied to the instance , yielding result .

Key constraints are uniqueness constraints that specify that a particular key field of a relation uniquely determines the other parameters. Key constraints are written as follows:

Constraint-example NNN (key-example)

The a_k field is a KEY for relation $r(a_0; a_1,...,a_n)$.

Because of the presence of attributes, key constraints do not reduce directly to uniqueness constraints. Instead, we enforce key constraints using the following ***merging*** process.

1. Suppose $r(a_0; a_1,...a_n,attrs1)$ and $r(b_0; b_1,...b_n,attrs2)$ hold in PROV instance , where the key fields $a_k = b_k$ are equal.
2. Attempt to unify all of the corresponding parameters $a_0 = b_0$ and ... and $a_n = b_n$.
3. If unification fails, then the constraint is unsatisfiable, so application of the key constraint to fails.
4. If unification succeeds with substitution , then we remove $r(a_0; a_1,...a_n,attrs1)$ and $r(b_0; b_1,...b_n,attrs2)$ from , obtaining instance , and return instance $\{r(S(a_0); S(a_1),...S(a_n),attrs1 \cup attrs2)\} \cup$.

Thus, if a PROV instance contains an apparent violation of a uniqueness constraint or key constraint, unification or merging can be used to determine whether the constraint can be satisfied by instantiating some existential variables with other terms. For key constraints, this is the same as merging pairs of statements whose keys are equal and whose corresponding arguments are compatible, because after unifying respective arguments and combining attribute lists, the two statements become equal and one can be omitted.

The various identified objects of PROV MUST have unique statements describing them within a valid PROV instance. This is enforced through the following key constraints:

Constraint 22 (key-object)

1. The identifier field `id` is a KEY for the `entity(id,attrs)` statement.
2. The identifier field `id` is a KEY for the `activity(id,t1,t2,attrs)` statement.
3. The identifier field `id` is a KEY for the `agent(id,attrs)` statement.

Likewise, the statements in a valid PROV instance must provide consistent information about each identified object or relationship. The following key constraints require that all of the information about each identified statement can be merged into a single, consistent statement:

Constraint 23 (key-properties)

1. The identifier field `id` is a KEY for the `wasGeneratedBy(id; e,a,t,attrs)` statement.
2. The identifier field `id` is a KEY for the `used(id; a,e,t,attrs)` statement.
3. The identifier field `id` is a KEY for the `wasInformedBy(id; a2,a1,attrs)` statement.
4. The identifier field `id` is a KEY for the `wasStartedBy(id; a2,e,a1,t,attrs)` statement.
5. The identifier field `id` is a KEY for the `wasEndedBy(id; a2,e,a1,t,attrs)` statement.
6. The identifier field `id` is a KEY for the `wasInvalidatedBy(id; e,a,t,attrs)` statement.
7. The identifier field `id` is a KEY for the `wasDerivedFrom(id; e2, e1, a, g2, u1, attrs)` statement.
8. The identifier field `id` is a KEY for the `wasAttributedTo(id; e,ag,attr)` statement.
9. The identifier field `id` is a KEY for the `wasAssociatedWith(id; a,ag,pl,attrs)` statement.
10. The identifier field `id` is a KEY for the `actedOnBehalfOf(id; ag2,ag1,a,attrs)` statement.
11. The identifier field `id` is a KEY for the `wasInfluencedBy(id; o2,o1,attrs)` statement.

Entities may have multiple generation or invalidation events (either or both may, however, be left implicit). An entity can be generated by more than one activity, with one generation event per each entity-activity pair. These events must be simultaneous, as required by Constraint 39 (generation-generation-ordering) and Constraint 40 (invalidation-invalidation-ordering).

Constraint 24 (unique-generation)

IF `wasGeneratedBy(gen1; e,a,_t1,_attrs1)` and `wasGeneratedBy(gen2; e,a,_t2,_attrs2)`, THEN `gen1` = `gen2`.

Constraint 25 (unique-invalidation)

IF `wasInvalidatedBy(inv1; e,a,_t1,_attrs1)` and `wasInvalidatedBy(inv2; e,a,_t2,_attrs2)`, THEN `inv1` = `inv2`.

Remark

It follows from the above uniqueness and key constraints that the generation and invalidation events linking an entity and activity are unique, if specified. However, because we apply the constraints by merging, it is possible for a valid PROV instance to contain multiple statements about the same generation or invalidation event, for example:

```
wasGeneratedBy(id1; e,a,-,[prov:location="Paris"])
wasGeneratedBy(-; e,a,-,[color="Red"])
```

When the uniqueness and key constraints are applied, the instance is normalized to the following form:

```
wasGeneratedBy(id1; e,a,_t,[prov:location="Paris",color="Red"])
```

where `_t` is a new existential variable.

An activity may have more than one start and end event, each having a different activity (either or both may, however, be left implicit). However, the triggering entity linking any two activities in a start or end event is unique. That is, an activity may be started by several other activities, with shared or separate triggering entities. If an activity is started or ended by multiple events, they must all be simultaneous, as specified in Constraint 31 (start-start-ordering) and Constraint 32 (end-end-ordering).

Constraint 26 (unique-wasStartedBy)

IF `wasStartedBy(start1; a,_e1,a0,_t1,_attrs1)` and `wasStartedBy(start2; a,_e2,a0,_t2,_attrs2)`, THEN `start1` = `start2`.

Constraint 27 (unique-wasEndedBy)

IF `wasEndedBy(end1; a,_e1,a0,_t1,_attrs1)` and `wasEndedBy(end2; a,_e2,a0,_t2,_attrs2)`, THEN `end1` = `end2`.

An activity start event is the instantaneous event that marks the instant an activity starts. It allows for an optional time attribute. Activities also allow for an optional start time attribute. If both are specified, they MUST be the same, as expressed by the following constraint.

Constraint 28 (unique-startTime)

IF `activity(a2,t1,_t2,_attrs)` and `wasStartedBy(_start; a2,_e,_a1,t,_attrs)`, THEN `t1`=`t`.

An activity end event is the instantaneous event that marks the instant an activity ends. It allows for an optional time attribute. Activities also allow for an optional end time attribute. If both are specified, they MUST be the same, as expressed by the following constraint.

Constraint 29 (unique-endTime)

IF `activity(a2,_t1,t2,_attrs)` and `wasEndedBy(_end; a2,_e,_a1,t,_attrs1)`, THEN `t2` = `t`.

6.2 Event Ordering Constraints

Given that provenance consists of a description of past entities and activities, valid provenance instances MUST satisfy *ordering constraints* between instantaneous events, which are introduced in this section. For instance, an entity can only be used after it was generated; in other words, an entity's generation event precedes any of this entity's usage events. Should this ordering constraint be violated, the associated generation and usage would not be credible. The rest of this section defines the ***temporal interpretation*** of provenance instances as a set of instantaneous event ordering constraints.

To allow for minimalistic clock assumptions, like Lamport [CLOCK], PROV relies on a notion of relative ordering of instantaneous events, without using physical clocks. This specification assumes that a preorder exists between instantaneous events.

Specifically, ***precedes*** is a preorder between instantaneous events. A constraint of the form `e1` precedes `e2` means that `e1` happened at the same time as or before `e2`. For symmetry, ***follows*** is defined as the inverse of precedes; that is, a constraint of the form `e1` follows `e2` means that `e1` happened at the same time

as or after e2. Both relations are preorders, meaning that they are reflexive and transitive. Moreover, we sometimes consider *strict* forms of these orders: we say e1 ***strictly precedes*** e2 to indicate that e1 happened before e2, but not at the same time. This is a transitive, irreflexive relation.

PROV also allows for time observations to be inserted in specific provenance statements, for each of the five kinds of instantaneous events introduced in this specification. Times in provenance records arising from different sources might be with respect to different timelines (e.g. different time zones) leading to apparent inconsistencies. For the purpose of checking ordering constraints, the times associated with events are irrelevant; thus, there is no inference that time ordering implies event ordering, or vice versa. However, an application MAY flag time values that appear inconsistent with the event ordering as possible inconsistencies. When generating provenance, an application SHOULD use a consistent timeline for related PROV statements within an instance.

A typical ordering constraint is as follows.

Constraint-example NNN (ordering-example)

IF hyp_1 and ... and hyp_n THEN evt1 precedes/strictly precedes evt2.

The conclusion of an ordering constraint is either precedes or strictly precedes. One way to check ordering constraints is to generate all precedes and strictly precedes relationships arising from the ordering constraints to form a directed graph, with edges marked precedes or strictly precedes, and check that there is no cycle containing a strictly precedes edge.

6.2.1 Activity constraints

This section specifies ordering constraints from the perspective of the lifetime of an activity. An activity starts, then during its lifetime can use, generate or invalidate entities, communicate with, start, or end other activities, or be associated with agents, and finally it ends. The following constraints amount to checking that all of the events associated with an activity take place within the activity's lifetime, and the start and end events mark the start and endpoints of its lifetime.

Figure 3 summarizes the ordering constraints on activities in a graphical manner. For this and subsequent figures, an event time line points to the right. Activities are represented by rectangles, whereas entities are represented by circles. Usage, generation and invalidation are represented by the corresponding edges between entities and activities. The five kinds of instantaneous events are represented by vertical dotted lines (adjacent to the vertical sides of an activity's rectangle, or intersecting usage and generation edges). The ordering constraints are represented by triangles: an occurrence of a triangle between two instantaneous event vertical dotted lines represents that the event denoted by the left line precedes the event denoted by the right line.

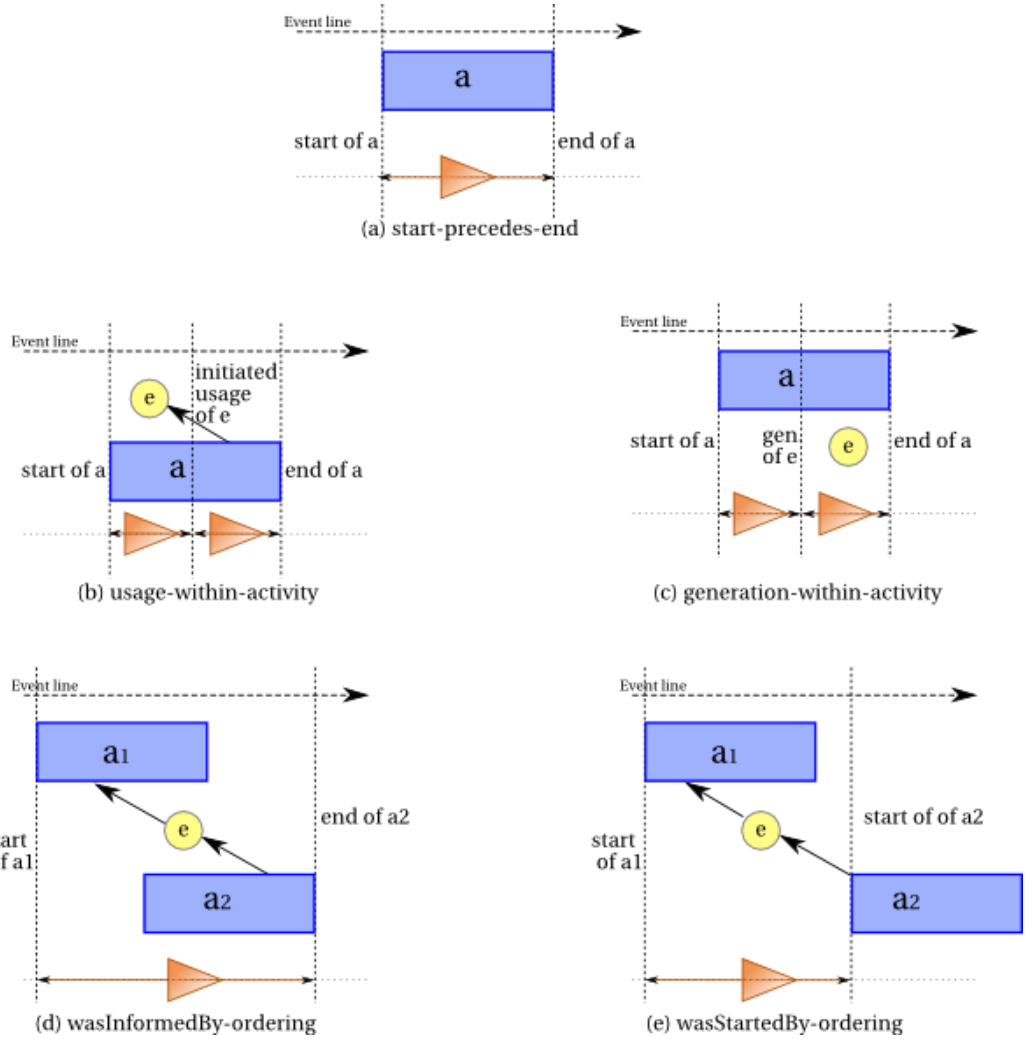


Figure 3 ◊: Summary of instantaneous event ordering constraints for activities

---

The existence of an activity implies that the activity start event always precedes the corresponding activity end event. This is illustrated by Figure 3 (a) and expressed by Constraint 30 (start-precedes-end).

Constraint 30 (start-precedes-end)

IF `wasStartedBy(start; a,_e1,_a1,_t1,_attrs1)` and `wasEndedBy(end; a,_e2,_a2,_t2,_attrs2)` THEN `start` precedes `end`.

If an activity is started by more than one activity, the events must all be simultaneous. The following constraint requires that if there are two start events that start the same activity, then one precedes the other. Using this constraint in both directions means that each event precedes the other.

Constraint 31 (start-start-ordering)

IF `wasStartedBy(start1; a,_e1,_a1,_t1,_attrs1)` and `wasStartedBy(start2; a,_e2,_a2,_t2,_attrs2)` THEN `start1` precedes `start2`.

If an activity is ended by more than one activity, the events must all be simultaneous. The following constraint requires that if there are two end events that end the same activity, then one precedes the other. Using this constraint in both directions means that each event precedes the other, that is, they are simultaneous.

Constraint 32 (end-end-ordering)

IF `wasEndedBy(end1; a,_e1,_a1,_t1,_attrs1)` and `wasEndedBy(end2; a,_e2,_a2,_t2,_attrs2)` THEN `end1` precedes `end2`.

A usage implies ordering of events, since the usage event had to occur during the associated activity. This is illustrated by Figure 3 (b) and expressed by Constraint 33 (usage-within-activity).

Constraint 33 (usage-within-activity)

1. IF `wasStartedBy(start; a,_e1,_a1,_t1,_attrs1)` and `used(use; a,_e2,_t2,_attrs2)` THEN `start` precedes `use`.
2. IF `used(use; a,_e1,_t1,_attrs1)` and `wasEndedBy(end; a,_e2,_a2,_t2,_attrs2)` THEN `use` precedes `end`.

A generation implies ordering of events, since the generation event had to occur during the associated activity. This is illustrated by Figure 3 (c) and expressed by Constraint 34 (generation-within-activity).

Constraint 34 (generation-within-activity)

1. IF `wasStartedBy(start; a,_e1,_a1,_t1,_attrs1)` and `wasGeneratedBy(gen; _e2,a,_t2,_attrs2)` THEN `start` precedes `gen`.
2. IF `wasGeneratedBy(gen; _e,a,_t,_attrs)` and `wasEndedBy(end; a,_e1,_a1,_t1,_attrs1)` THEN `gen` precedes `end`.

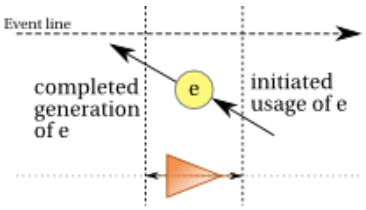
Communication between two activities `a1` and `a2` also implies ordering of events, since some entity must have been generated by the former and used by the latter, which implies that the start event of `a1` cannot follow the end event of `a2`. This is illustrated by Figure 3 (d) and expressed by Constraint 35 (wasInformedBy-ordering).

Constraint 35 (wasInformedBy-ordering)

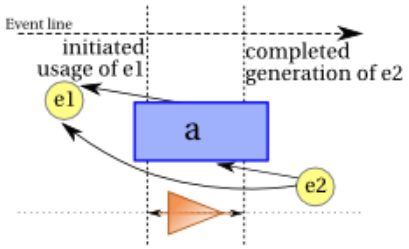
IF `wasInformedBy(_id; a2,a1,_attrs)` and `wasStartedBy(start; a1,_e1,_a1',_t1,_attrs1)` and `wasEndedBy(end; a2,_e2,_a2',_t2,_attrs2)` THEN `start` precedes `end`.

6.2.2 Entity constraints

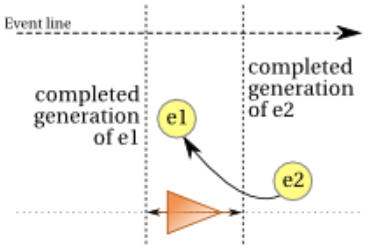
As with activities, entities have lifetimes: they are generated, then can be used, other entities can be derived from them, and finally they can be invalidated. The constraints on these events are illustrated graphically in Figure 4 and Figure 5.



(a) generation-precedes-usage



(b) derivation-usage-generation-ordering



(c) derivation-generation-generation-ordering

Figure 4 ◊: Summary of instantaneous event ordering constraints for entities

---

Generation of an entity precedes its invalidation. (This follows from other constraints if the entity is used, but it is stated explicitly here to cover the case of an entity that is generated and invalidated without being used.)

Constraint 36 (generation-precedes-invalidation)

IF `wasGeneratedBy(gen; e,_a1,_t1,_attrs1)` and `wasInvalidatedBy(inv; e,_a2,_t2,_attrs2)` THEN `gen` precedes `inv`.

A usage and a generation for a given entity implies ordering of events, since the generation event had to precede the usage event. This is illustrated by Figure 4(a) and expressed by Constraint 37 (generation-precedes-usage).

Constraint 37 (generation-precedes-usage)

IF `wasGeneratedBy(gen; e,_a1,_t1,_attrs1)` and `used(use; _a2,e,_t2,_attrs2)` THEN `gen` precedes `use`.

All usages of an entity precede its invalidation, which is captured by Constraint 38 (usage-precedes-invalidation) (without any explicit graphical representation).

Constraint 38 (usage-precedes-invalidation)

IF `used(use; _a1,e,_t1,_attrs1)` and `wasInvalidatedBy(inv; e,_a2,_t2,_attrs2)` THEN `use` precedes `inv`.

If an entity is generated by more than one activity, the events must all be simultaneous. The following constraint requires that if there are two generation events that generate the same entity, then one precedes the other. Using this constraint in both directions means that each event precedes the other.

Constraint 39 (generation-generation-ordering)

IF `wasGeneratedBy(gen1; e,_a1,_t1,_attrs1)` and `wasGeneratedBy(gen2; e,_a2,_t2,_attrs2)`

THEN `gen1` precedes `gen2`.

If an entity is invalidated by more than one activity, the events must all be simultaneous. The following constraint requires that if there are two invalidation events that invalidate the same entity, then one precedes the other. Using this constraint in both directions means that each event precedes the other, that is, they are simultaneous.

Constraint 40 (invalidation-invalidation-ordering)

IF `wasInvalidatedBy(inv1; e,_a1,_t1,_attrs1)` and `wasInvalidatedBy(inv2; e,_a2,_t2,_attrs2)` THEN `inv1` precedes `inv2`.

If there is a derivation relationship linking `e2` and `e1`, then this means that the entity `e1` had some influence on the entity `e2`; for this to be possible, some event ordering must be satisfied. First, we consider derivations, where the activity and usage are known. In that case, the usage of `e1` has to precede the generation of `e2`. This is illustrated by Figure 4 (b) and expressed by Constraint 41 (derivation-usage-generation-ordering).

Constraint 41 (derivation-usage-generation-ordering)

In this constraint, `_a`, `gen2`, `use1` MUST NOT be placeholders.

IF `wasDerivedFrom(_d; _e2,_e1,_a,gen2,use1,_attrs)` THEN `use1` precedes `gen2`.

When the activity, generation or usage is unknown, a similar constraint exists, except that the constraint refers to its generation event, as illustrated by Figure 4 (c) and expressed by Constraint 42 (derivation-generation-generation-ordering).

Constraint 42 (derivation-generation-generation-ordering)

In this constraint, any of `_a`, `_g`, `_u` MAY be placeholders.

IF `wasDerivedFrom(_d; e2,e1,_a,_g,_u,attrs)` and `wasGeneratedBy(gen1; e1,_a1,_t1,_attrs1)` and `wasGeneratedBy(gen2; e2,_a2,_t2,_attrs2)` THEN `gen1` strictly precedes `gen2`.

Remark

This constraint requires the derived entity to be generated strictly following the generation of the original entity. This follows from the [PROV-DM] definition of derivation: *A derivation is a transformation of an entity into another, an update of an entity resulting in a newone, or the construction of a newentity based on a pre-existing entity*, thus the derived entity must be newer than the original entity.

The event ordering is between generations of `e1` and `e2`, as opposed to derivation where usage is known, which implies ordering between the usage of `e1` and generation of `e2`.

The entity that triggered the start of an activity must exist before the activity starts. This is illustrated by Figure 5(a) and expressed by Constraint 43 (wasStartedBy-ordering).

Constraint 43 (wasStartedBy-ordering)

1. IF `wasGeneratedBy(gen; e,_a1,_t1,_attrs1)` and `wasStartedBy(start; _a,e,_a2,_t2,_attrs2)` THEN `gen` precedes `start`.
2. IF `wasStartedBy(start; _a,e,_a1,_t1,_attrs1)` and `wasInvalidatedBy(inv; e,_a2,_t2,_attrs2)` THEN `start` precedes `inv`.

Similarly, the entity that triggered the end of an activity must exist before the activity ends, as illustrated by Figure 5(b).

Constraint 44 (wasEndedBy-ordering)

1. IF `wasGeneratedBy(gen; e,_a1,_t1,_attrs1)` and `wasEndedBy(end; _a,e,_a2,_t2,_attrs2)` THEN `gen` precedes `end`.
2. IF `wasEndedBy(end; _a,e,_a1,_t1,_attrs1)` and `wasInvalidatedBy(inv; e,_a2,_t2,_attrs2)` THEN `end` precedes `inv`.

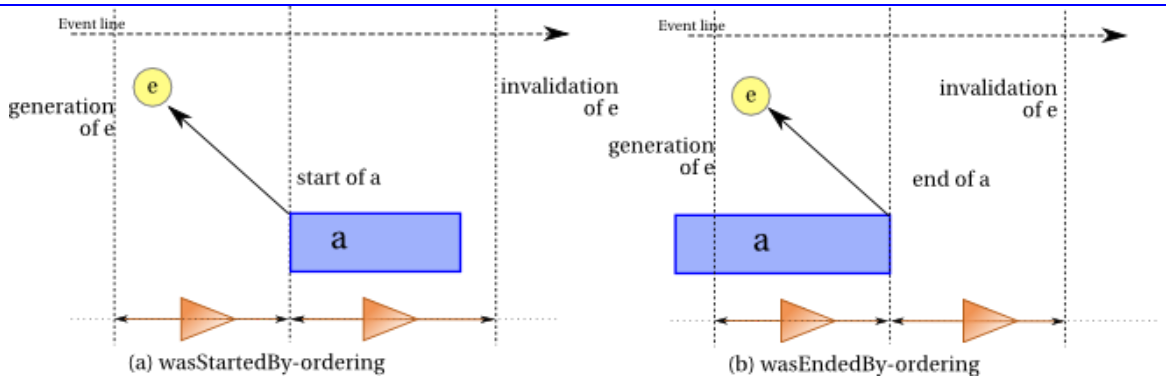


Figure 5 ◊: Summary of instantaneous event ordering constraints for trigger entities

If an entity is a specialization of another, then the more specific entity must have been generated after the less specific entity was generated.

Constraint 45 (specialization-generation-ordering)

IF `specializationOf(e2,e1)` and `wasGeneratedBy(gen1; e1,_a1,_t1,_attrs1)` and `wasGeneratedBy(gen2; e2,_a2,_t2,_attrs2)` THEN `gen1` precedes `gen2`.

Similarly, if an entity is a specialization of another entity, and then the invalidation event of the more specific entity precedes that of the less specific entity.

Constraint 46 (specialization-invalidation-ordering)

IF `specializationOf(e1,e2)` and `wasInvalidatedBy(inv1; e1,_a1,_t1,_attrs1)` and `wasInvalidatedBy(inv2; e2,_a2,_t2,_attrs2)` THEN `inv1` precedes `inv2`.

6.2.3 Agent constraints

Like entities and activities, agents have lifetimes that follow a familiar pattern. An agent that is also an entity can be generated and invalidated; an agent that is also an activity can be started or ended. During its lifetime, an agent can participate in interactions such as starting or ending other activities, association with an activity, attribution, or delegation.

Further constraints associated with agents appear in Figure 6 and are discussed below.

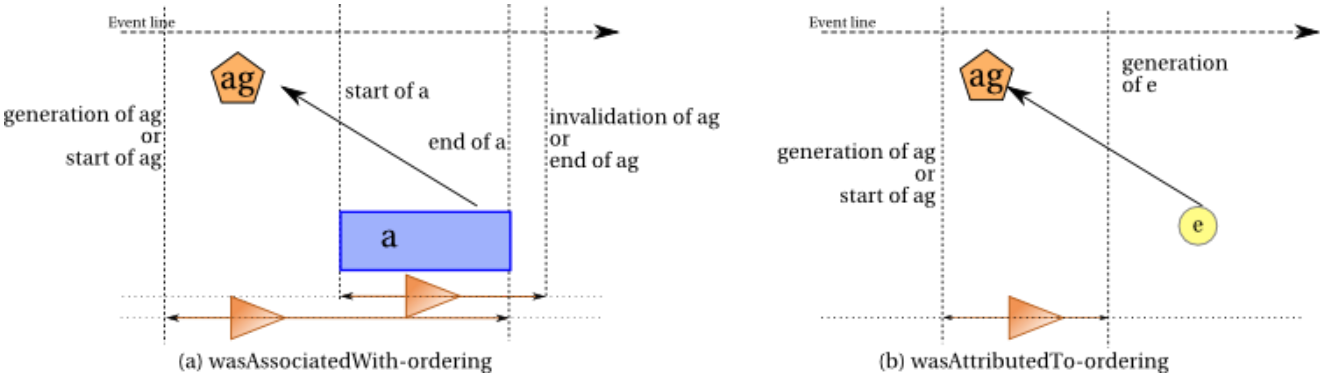


Figure 6 ◊: Summary of instantaneous event ordering constraints for agents

An activity that was associated with an agent must have some overlap with the agent. The agent MUST have been generated (or started), or MUST have become associated with the activity, after the activity start: so, the agent MUST exist before the activity end. Likewise, the agent may be destructed (or ended), or may terminate its association with the activity, before the activity end: hence, the agent invalidation (or end) is required to happen after the activity start. This is illustrated by Figure 6 (a) and expressed by Constraint 47 (wasAssociatedWith-ordering).

Constraint 47 (wasAssociatedWith-ordering)

In the following inferences, `_pl` MAY be a placeholder `-`.

1. IF `wasAssociatedWith(_assoc; a,ag,_pl,_attrs)` and `wasStartedBy(start1; a,_e1,_a1,_t1,_attrs1)` and `wasInvalidatedBy(inv2; ag,_a2,_t2,_attrs2)` THEN `start1` precedes `inv2`.
2. IF `wasAssociatedWith(_assoc; a,ag,_pl,_attrs)` and `wasGeneratedBy(gen1; ag,_a1,_t1,_attrs1)` and `wasEndedBy(end2; a,_e2,_a2,_t2,_attrs2)` THEN `gen1` precedes `end2`.
3. IF `wasAssociatedWith(_assoc; a,ag,_pl,_attrs)` and `wasStartedBy(start1; a,_e1,_a1,_t1,_attrs1)` and `wasEndedBy(end2; ag,_e2,_a2,_t2,_attrs2)` THEN `start1` precedes `end2`.
4. IF `wasAssociatedWith(_assoc; a,ag,_pl,_attrs)` and `wasStartedBy(start1; ag,_e1,_a1,_t1,_attrs1)` and `wasEndedBy(end2; a,_e2,_a2,_t2,_attrs2)` THEN `start1` precedes `end2`.

Remark

Case 3 of the above constraint says that the agent `ag` must have ended after the start of the activity `a`, ensuring some overlap between the two. Since `ag` is the subject of a `wasEndedBy` statement, it is an activity according to the typing constraints. Case 4 handles the symmetric case, ensuring that the start of an agent-activity precedes the start of an associated activity.

An agent to which an entity was attributed, MUST exist before this entity was generated. This is illustrated by Figure 6 (b) and expressed by Constraint 48 (wasAttributedTo-ordering).

Constraint 48 (wasAttributedTo-ordering)

1. IF `wasAttributedTo(_at; e,ag,_attrs)` and `wasGeneratedBy(gen1; ag,_a1,_t1,_attrs1)` and `wasGeneratedBy(gen2; e,_a2,_t2,_attrs2)` THEN `gen1` precedes `gen2`.
2. IF `wasAttributedTo(_at; e,ag,_attrs)` and `wasStartedBy(start1; ag,_e1,_a1,_t1,_attrs1)` and `wasGeneratedBy(gen2; e,_a2,_t2,_attrs2)` THEN `start1` precedes `gen2`.

For delegation, the responsible agent has to precede or have some overlap with the subordinate agent.

Constraint 49 (actedOnBehalfOf-ordering)

1. IF `actedOnBehalfOf(_del; ag2,ag1,_a,_attrs)` and `wasGeneratedBy(gen1; ag1,_a1,_t1,_attrs1)` and `wasInvalidatedBy(inv2; ag2,_a2,_t2,_attrs2)` THEN `gen1` precedes `inv2`.
2. IF `actedOnBehalfOf(_del; ag2,ag1,_a,_attrs)` and `wasStartedBy(start1; ag1,_e1,_a1,_t1,_attrs1)` and `wasEndedBy(end2; ag2,_e2,_a2,_t2,_attrs2)` THEN `start1` precedes `end2`.

6.3 Type Constraints

The following rules assign types to identifiers based on their use within statements. The function `typeOf` gives the set of types denoted by an identifier. That is, `typeOf(e)` returns the set of types associated with identifier `e`. The function `typeOf` is not a PROV statement, but a construct used only during validation, similar to precedes.

For any identifier `id`, `typeOf(id)` is a subset of {`'entity'`, `'activity'`, `'agent'`, `'prov:Collection'`, `'prov:EmptyCollection'`}. For identifiers that do not have a type, `typeOf` gives the empty set. Identifiers can have more than one type, because of subtyping (e.g. `'prov:EmptyCollection'` is a subtype of `'prov:Collection'`) or because certain types are not disjoint (such as `'agent'` and `'entity'`). The set of types does not reflect all of the distinctions among objects, only those relevant for checking validity. In particular, a subtype such as `'plan'` is omitted, and statements such as `wasAssociatedWith` that have plan parameters only check that these parameters are entities.

To check if a PROV instance satisfies type constraints, one obtains the types of identifiers by application of Constraint 50 (typing) and check that none of the impossibility constraints Constraint 55 (entity-activity-disjoint) and Constraint 56 (membership-empty-collection) are violated as a result.

Constraint 50 (typing)

1. IF `entity(e,attrs)` THEN `'entity' ∈ typeOf(e)`.
2. IF `agent(ag,attrs)` THEN `'agent' ∈ typeOf(ag)`.
3. IF `activity(a,t1,t2,attrs)` THEN `'activity' ∈ typeOf(a)`.
4. IF `used(u; a,e,t,attrs)` THEN `'activity' ∈ typeOf(a)` AND `'entity' ∈ typeOf(e)`.
5. IF `wasGeneratedBy(gen; e,a,t,attrs)` THEN `'entity' ∈ typeOf(e)` AND `'activity' ∈ typeOf(a)`.
6. IF `wasInformedBy(id; a2,a1,attrs)` THEN `'activity' ∈ typeOf(a2)` AND `'activity' ∈ typeOf(a1)`.
7. IF `wasStartedBy(id; a2,e,a1,t,attrs)` THEN `'activity' ∈ typeOf(a2)` AND `'entity' ∈ typeOf(e)` AND `'activity' ∈ typeOf(a1)`.
8. IF `wasEndedBy(id; a2,e,a1,t,attrs)` THEN `'activity' ∈ typeOf(a2)` AND `'entity' ∈ typeOf(e)` AND `'activity' ∈ typeOf(a1)`.
9. IF `wasInvalidatedBy(id; e,a,t,attrs)` THEN `'entity' ∈ typeOf(e)` AND `'activity' ∈ typeOf(a)`.
10. IF `wasDerivedFrom(id; e2, e1, a, g2, u1, attrs)` THEN `'entity' ∈ typeOf(e2)` AND `'entity' ∈ typeOf(e1)` AND `'activity' ∈ typeOf(a)`. In this constraint, `a`, `g2`, and `u1` MUST NOT be placeholders.
11. IF `wasDerivedFrom(id; e2, e1, -, -, -, attrs)` THEN `'entity' ∈ typeOf(e2)` AND `'entity' ∈ typeOf(e1)`.
12. IF `wasAttributedTo(id; e,ag,attr)` THEN `'entity' ∈ typeOf(e)` AND `'agent' ∈ typeOf(ag)`.
13. IF `wasAssociatedWith(id; a,ag,pl,attrs)` THEN `'activity' ∈ typeOf(a)` AND `'agent' ∈ typeOf(ag)` AND `'entity' ∈ typeOf(pl)`. In this constraint, `pl` MUST NOT be a placeholder.
14. IF `wasAssociatedWith(id; a,ag,-,attrs)` THEN `'activity' ∈ typeOf(a)` AND `'agent' ∈ typeOf(ag)`.
15. IF `actedOnBehalfOf(id; ag2,ag1,a,attrs)` THEN `'agent' ∈ typeOf(ag2)` AND `'agent' ∈ typeOf(ag1)` AND `'activity' ∈ typeOf(a)`.
16. IF `alternateOf(e2, e1)` THEN `'entity' ∈ typeOf(e2)` AND `'entity' ∈ typeOf(e1)`.

17. IF `specializationOf(e2, e1)` THEN `'entity' ∈ typeOf(e2)` AND `'entity' ∈ typeOf(e1)`.
18. IF `hadMember(c,e)` THEN `'prov:Collection' ∈ typeOf(c)` AND `'entity' ∈ typeOf(c)` AND `'entity' ∈ typeOf(e)`.
19. IF `entity(c,[prov:type='prov:EmptyCollection'])` THEN `'entity' ∈ typeOf(c)` AND `'prov:Collection' ∈ typeOf(c)` AND `'prov:EmptyCollection' ∈ typeOf(c)`.

6.4 Impossibility constraints

Impossibility constraints require that certain patterns of statements never appear in valid PROV instances. Impossibility constraints have the following general form:

Constraint-example NNN (impossible-example)

IF hyp_1 and ... and hyp_n THEN INVALID.

Checking an impossibility constraint on instance means checking whether there is any way of matching the pattern hyp_1, ..., hyp_n. If there is, then checking the constraint on fails (which implies that is invalid).

A derivation with unspecified activity `wasDerivedFrom(id;e1,e2,-,g,u,attrs)` represents a derivation that takes one or more steps, whose activity, generation and use events are unspecified. It is forbidden to specify a generation or use event without specifying the activity.

Constraint 51 (impossible-unspecified-derivation-generation-use)

In the following rules, `g` and `u` MUST NOT be `-`.

1. IF `wasDerivedFrom(_id;_e2,_e1,-,g,-,attrs)` THEN INVALID.
2. IF `wasDerivedFrom(_id;_e2,_e1,-,-,u,attrs)` THEN INVALID.
3. IF `wasDerivedFrom(_id;_e2,_e1,-,g,u,attrs)` THEN INVALID.

As noted previously, specialization is a strict partial order: it is irreflexive and transitive.

Constraint 52 (impossible-specialization-reflexive)

IF `specializationOf(e,e)` THEN INVALID.

Furthermore, identifiers of basic relationships are disjoint.

Constraint 53 (impossible-property-overlap)

For each `r` and `s` in { `used`, `wasGeneratedBy`, `wasInvalidatedBy`, `wasStartedBy`, `wasEndedBy`, `wasInformedBy`, `wasAttributedTo`, `wasAssociatedWith`, `actedOnBehalfOf`} such that `r` and `s` are different relation names, the following constraint holds:

IF $r(id; a_1,...,a_m)$ and $s(id; b_1,...,b_n)$ THEN INVALID.

Remark

Since `wasInfluencedBy` is a superproperty of many other properties, it is excluded from the set of properties whose identifiers are required to be pairwise disjoint. The following example illustrates this observation:

```
wasInfluencedBy(id;e2,e1)
wasDerivedFrom(id;e2,e1)
```

This satisfies the disjointness constraint.

There is, however, no constraint requiring that every influence relationship is accompanied by a more specific relationship having the same identifier. The following valid example illustrates this observation:

```
wasInfluencedBy(id; e2,e1)
```

This is valid; there is no inferrable information about what kind of influence relates `e2` and `e1`, other than its identity.

Identifiers of entities, agents and activities cannot also be identifiers of properties.

Constraint 54 (impossible-object-property-overlap)

For each `p` in {`entity`, `activity` or `agent`} and for each `r` in { `used`, `wasGeneratedBy`, `wasInvalidatedBy`, `wasInfluencedBy`, `wasStartedBy`, `wasEndedBy`, `wasInformedBy`, `wasDerivedFrom`, `wasAttributedTo`, `wasAssociatedWith`, `actedOnBehalfOf`}, the following impossibility constraint holds:

IF $p(id,a_1,...,a_m)$ and $r(id; b_1,...,b_n)$ THEN INVALID.

The set of entities and activities are disjoint, expressed by the following constraint:

Constraint 55 (entity-activity-disjoint)

IF `'entity' ∈ typeOf(id)` AND `'activity' ∈ typeOf(id)` THEN INVALID.

Remark

There is no disjointness between entities and agents. This is because one might want to make statements about the provenance of an agent, by making it an entity. For example, one can assert both `entity(a1)` and `agent(a1)` in a valid PROV instance. Similarly, there is no disjointness between activities and agents, and one can assert both `activity(a1)` and `agent(a1)` in a valid PROV instance. However, one should keep in mind that some specific types of agents may not be suitable as activities. For example, asserting statements such as `agent(Bob, [type=prov:Person])` and `activity(Bob)` is discouraged. In these cases, disjointness can be ensured by explicitly asserting the agent as both agent and entity, and applying Constraint 55 (entity-activity-disjoint).

An empty collection cannot contain any member, expressed by the following constraint:

Constraint 56 (membership-empty-collection)

IF `hadMember(c,e)` and `'prov:EmptyCollection' ∈ typeOf(c)` THEN INVALID.

7. Normalization, Validity, and Equivalence

We define the notions of normalization, validity and equivalence of PROV documents and instances. We first define these concepts for PROV instances and then extend them to PROV documents.

Remark

Implementations should expand namespace prefixes and perform any appropriate reasoning about

co-reference of identifiers, and rewrite the instance (by replacing co-referent identifiers with a single common identifier) to make this explicit, before doing validation, equivalence checking, or normalization. All of the following definitions assume that the application has already determined which URIs in the PROV instance are co-referent (e.g. `owl:sameAs` as a result of OWL reasoning).

7.1 Instances

We define the ***normal form*** of a PROV instance as the set of provenance statements resulting from applying all definitions, inferences, and uniqueness constraints, obtained as follows:

1. Apply all definitions to by replacing each defined statement by its definition (possibly introducing fresh existential variables in the process), yielding an instance .
2. Apply all inferences to by adding the conclusion of each inference whose hypotheses are satisfied and whose entire conclusion does not already hold (again, possibly introducing fresh existential variables), yielding an instance .
3. Apply all uniqueness constraints to by unifying terms or merging statements and applying the resulting substitution to the instance, yielding an instance . If some uniqueness constraint cannot be applied, then normalization fails.
4. If no definitions, inferences, or uniqueness constraints can be applied to instance , then is the normal form of .
5. Otherwise, the normal form of is the same as the normal form of (that is, proceed by normalizing at step 1).

Because of the potential interaction among definitions, inferences, and constraints, the above algorithm is iterative. Nevertheless, all of our constraints fall into a class of *tuple-generating dependencies* and *equality-generating dependencies* that satisfy a termination condition called *weak acyclicity* that has been studied in the context of relational databases [DBCONSTRAINTS]. Therefore, the above algorithm terminates, independently of the order in which inferences and constraints are applied. Appendix A gives a proof that normalization terminates and produces a unique (up to isomorphism) normal form.

A PROV instance is ***valid*** if its normal form exists and all of the validity constraints succeed on the normal form. The following algorithm can be used to test validity:

1. Normalize the instance , obtaining normal form . If normalization fails, then is not valid.
2. Apply all event ordering constraints to to build a graph whose nodes are event identifiers and edges are labeled by "precedes" and "strictly precedes" relationships among events induced by the constraints.
3. Determine whether there is a cycle in that contains a "strictly precedes" edge. If so, then is not valid.
4. Apply the type constraints (section 5.3) to determine whether there are any violations of disjointness. If so, then is not valid.
5. Check that none of the impossibility constraints (section 5.4) are violated. If any are violated, then is not valid. Otherwise, is valid.

A normal form of a PROV instance does not exist when a uniqueness constraint fails due to unification or merging failure.

Two valid PROV instances are ***equivalent*** if they have isomorphic normal forms. That is, after applying all possible inference rules, the two instances produce the same set of PROV statements, up to reordering of statements and attributes within attribute lists, and renaming of existential variables.

Equivalence can also be checked over pairs of PROV instances that are not necessarily valid, subject to the following rules:

- If both are valid, then equivalence is defined above.
- If both are invalid, then equivalence can be implemented in any way provided it is reflexive, symmetric, and transitive.
- If one instance is valid and the other is invalid, then the two instances are not equivalent.

Equivalence has the following characteristics over valid instances:

- The order of provenance statements is irrelevant to the meaning of a PROV instance. That is, a

PROV instance is equivalent to any other instance obtained by reordering its statements.
- The order of attribute-value pairs in attribute lists is irrelevant to the meaning of a PROV statement. That is, a PROV statement carrying attributes is equivalent to any other statement obtained by reordering attribute-value pairs and eliminating duplicate pairs.
- The particular choices of names of existential variables are irrelevant to the meaning of an instance; that is, the names can be renamed without changing the meaning, as long as different names are always replaced with different names. (Replacing two different names with equal names, however, can change the meaning, so does not preserve equivalence.)
- Applying inference rules, definitions, and uniqueness constraints preserves equivalence. That is, a PROV instance is equivalent to the instance obtained by applying any inference rule or definition, or by unifying two terms or merging two statements to enforce a uniqueness constraint.
- Equivalence is reflexive, symmetric, and transitive. (This is because a valid instance has a unique normal form up to isomorphism [DBCONSTRAINTS]).

An application that processes PROV data SHOULD handle equivalent instances in the same way. This guideline is necessarily imprecise because "in the same way" is application-specific. Common exceptions to this guideline include, for example, applications that pretty-print or digitally sign provenance, where the order and syntactic form of statements matters.

7.2 Bundles and Documents

The definitions, inferences, and constraints, and the resulting notions of normalization, validity and equivalence, work on a single PROV instance. In this section, we describe how to deal with general PROV documents, possibly including multiple named bundles as well as a toplevel instance. Briefly, each bundle is handled independently; there is no interaction between bundles from the perspective of applying definitions, inferences, or constraints, computing normal forms, or checking validity or equivalence.

We model a general PROV document, containing n named bundles $b_1 \ldots b_n$, as a tuple $(I_0, [b_1=I_1, \ldots, b_n=I_n])$ where I_0 is the toplevel instance, and for each i, I_i is the instance associated with bundle b_i. This notation is shorthand for the following PROV-N syntax:

```
document
   I0
   bundle b1
      I1
   endBundle
   ...
   bundle bn
      In
   endBundle
endDocument
```

The normal form of a PROV document $(I_0, [b_1=I_1, \ldots, [b_n=I_n])$ is $(I'_0, [b_1=I'_1, \ldots, b_n=I'_n])$ where I'_i is the normal form of I_i for each i between 0 and n.

A PROV document is valid if each of the bundles I_0, ..., I_n are valid and none of the bundle identifiers b_i are repeated.

Two (valid) PROV documents $(I_0, [b_1=I_1, \ldots, b_n=I_n])$ and $(I'_0, [b_1'=I'_1, \ldots, b'_m=I'_m])$ are equivalent if I_0 is equivalent to I'_0 and $n = m$ and there exists a permutation $P : \{1..n\} \to \{1..n\}$ such that for each i, $b_i = b'_{P(i)}$ and I_i is equivalent to $I'_{P(i)}$.

8. Glossary

- ***antisymmetric***: A relation over is antisymmetric if for any elements , of , if and then .
- ***asymmetric***: A relation over is asymmetric if and do not hold for any elements , of .
- ***equivalence relation***: An equivalence relation is a relation that is reflexive, symmetric, and transitive.
- ***irreflexive***: A relation over is irreflexive if for does not hold for any element of .
- ***partial order***: A partial order is a relation that is reflexive, antisymmetric, and transitive.
- ***preorder***: A preorder is a relation that is reflexive and transitive. (It is not necessarily antisymmetric, meaning there can be cycles of distinct elements
- ***reflexive***: A relation over is reflexive if for any element of , we have .

- ***strict partial order***: A strict partial order is a relation that is irreflexive, asymmetric and transitive.
- ***strict preorder***: A strict preorder is a relation that is irreflexive and transitive.
- ***symmetric***: A relation over is symmetric if for any elements , of , if then .
- ***transitive***: A relation over is transitive if for any elements , , of , if and then .

A. Termination of normalization

This section is non-normative.

We will show that normalization terminates, that is, that applying definitions, inferences and uniqueness/key constraints eventually either fails (due to constraint violation) or terminates with a normal form.

First, since the inferences and constraints never introduce new defined statements, for the purpose of termination we always expand the definitions first and then consider only normalization of instances in which there are no remaining defined statements.

We will prove termination for the simple case where there are no attributes. For the general case, we will show that any nontermination arising from an instance that does involve attributes would also arise from one with no attributes.

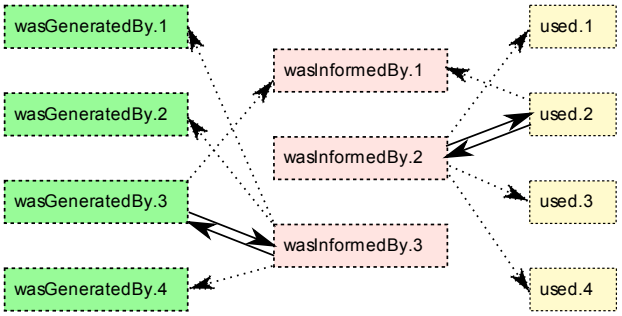
Termination for instances without attributes. As shown in [DBCONSTRAINTS], termination of normalization can be shown by checking that the inference rules are *weakly acyclic*. In addition, weak acyclicity can be checked in a modular fashion for our system, because there are only a few possible cycles among statements. The following table summarizes seven *stages* of the inference rules; because there are no cycles among stages, it is sufficient to check weak acyclicity of each stage independently.

Stage #	Inference	Hypotheses	Conclusions
1	19, 20, 21	`specializationOf`	`specializationOf, entity`
2	7, 8, 13, 14	`entity, activity, wasAttributedTo, actedOnBehalfOf`	`wasInvalidatedBy, wasStartedBy, wasEndedBy, wasAssociatedWith`
3	9, 10	`wasStartedBy, wasEndedBy`	`wasGeneratedBy`
4	11, 12	`wasDerivedFrom`	`wasGeneratedBy, used, alternateOf`
5	16, 17, 18	`alternateOf, entity`	`alternateOf`
6	5, 6	`wasInformedBy, generated, used`	`wasInformedBy, generated, used`
7	15	many	`wasInfluencedBy`

For each stage, we show that the stage is weakly acyclic.

- Stages 1 and 5 have no rules with existential quantifiers, so they are weakly acyclic.
- Stages 2, 3, 4, and 7 have no cycles among the formulas involved, so they are weakly acyclic.
- For stage 6, we check weak acyclicity using the algorithm in [DBCONSTRAINTS], namely:
  - Given a formula $\varphi \Rightarrow \exists \psi$
  - For every that occurs in ψ, and for every occurrence of in φ in position :
    1. For every occurrence of in position , add an edge from to (if it does not already exist).
    2. In addition, for every existentially quantified variable and for every occurrence of in ψ in position , add a special edge from to (if it does not already exist).

  Weak acyclicity means that there is no cycle involving a special edge in the resulting graph. For the two inferences in stage 6, the following dependency graph witnesses weak acyclicity. The nodes `wasGeneratedBy.i`, `wasInformedBy.i`, and `used.i` denote the th arguments of the corresponding predicates. The solid edges are ordinary edges, and the dashed edges are *special* edges.



Termination for instances with attributes. We can translate an instance with attributes to an alternative, purely relational language by introducing a relation `attribute(id,a,v)` and replacing every statement of the form $r(id;a1,...,a_n,[(k_1,v_1),...,(k_m,v_m)])$ with $r(id;a1,...,a_n),attribute(id,k_1,v_1),...,attribute(id,k_m,v_m)$, and similarly for `entity`, `activity` and `agent` attributes. The inference rules can also be translated so as to work on these instances, and a similar argument to the above shows that inference is terminating on instances with explicit attributes. Any infinite sequence of normalization steps on the original instance would lead to an infinite sequence of translated normalization steps on instances with explicit attributes.

B. Change Log

B.1 Changes since Proposed Recommendation

- Changed the status of this document section.
- Changed all URLs to PROV documents.
- Fixed some minor typos in section 2.4.
- Fixed typo in remark after definition 2.
- Fixed typo in paragraph after table 2.
- Fixed typo in sec. 6.3.
- Fixed wording in remark in sec. 7.1, and moved it before section heading.
- Fixed typo in appendix A.

B.2 Changes from Candidate Recommendation to Proposed Recommendation

Please see the Responses to Public Comments on the Candidate Recommendation for more details about the justification of these changes.

- Reworded description of toplevel instance and bundles to provide a clearer restatement in one place (Sec. 1.2)
- Fixed a number of broken links from definition references to definitions
- Updated the Status of This Document to describe proposed recommendation status and updated references to other PR-stage PROV documents
- Added clarification concerning constraint wasAssociatedWith-ordering (issue-615).
- Added underscores to some variables in inferences 9, 10, 15 (issue-611).
- Corrected a typo in the name `hadMember` in constraint 56 (issue-611).
- Clarified that existential variables are scoped at the instance level, not statement level, thus it is correct to apply uniqueness constraints by substituting variables through an instance (issue-611). (Remark at the end of section 4. Basic concepts, and clarified discussion in Validation Process Overview)
- Gave equivalent form of Definition 2 (issue-611).

B.3 Changes from Last Call Working Draft to Candidate Recommendation

Please see the Responses to Public Comments on the Last Call Working Draft for more details about the

justification of these changes.

- Abstract: clarified that term "validity" is analogous to other notions of validity in Web standards.
- Added bullet point linking to PROV-SEM under "How to read the PROV family of specifications"
- Revised sec. 1.2 to clarify terminology (validity), emphasize that any implementation equivalent to the procedural specification is compliant, and clarify that treating "equivalent instances in the same way" is a guideline.
- Added paragraph to sec 2.4 to clarify the purpose of the section.
- Sec 2.4 Unification and Merging: changed "merging" to "unification" for terms
- Sec. 2.4 "Applying definitions, inferences and constraints": Updated merging to unification and added paragraph reinforcing that compliance is algorithm independent
- Sec. 2.4 "Checking ordering, typing and impossibility constraints": Avoided use of the term "satisfies".
- Sec. 2.4 "Equivalence and Isomorphism": Extended equivalence to be defined on all instances, valid or not. Removed analogy to RDF.
- Sec. 2.4 "From Instances to Bundles and Documents": Revised to avoid giving the impression that toplevel instances must be disjoint from bundles; removed reference to RDF.
- Sec. 3. Clarified and reinforced algorithm independence.
- Sec. 4. Added clarifying remark about role of definitions.
- Sec. 4. Avoided reference to RDF, minor clarifications to discussion of existential variables.
- Sec. 4.1. Spelling correction.
- Sec. 4.4. Merging -> unification
- Sec. 5. Merging -> unification terminology change. Added declarative definition of unification. Clarified procedural definition. Removed definition of merging of attribute lists. Updated descriptions of uniqueness and key constraint application.
- Constraint 23. Renamed e, a, ag to id.
- Sec. 5.2. Explicitly stated that strictly-precedes is irreflexive.
- Sec. 5.2. Spelling
- Sec. 5.2, just before constraint 51: updated text to accurately describe constraint.
- Sec. 6. Merging -> unification. Updated definition of validity to avoid referring to "satisfies". Explicitly defined isomorphism of instances. Broadened the definition of equivalence so that it is allowed to test arbitrary instances for equivalence. Reinforce the intention of the guideline that applications treat equivalent instances "in the same way".
- Dropped RDF as a normative reference.
- Made PROV-DM and PROV-N into normative references.
- Added "document" and "endDocument" to sec. 6.2.
- Added sentence of explanation of purpose to beginning.
- Moved "mention" to a separate note.
- Added Section 4: Basic Concepts.
- Miscellaneous final cleanup prior to CR staging.
- Added html link to provenance.

C. Acknowledgements

This document has been produced by the Provenance Working Group, and its contents reflect extensive discussion within the Working Group as a whole. The editors extend special thanks to Ivan Herman (W3C/ERCIM), Paul Groth, Tim Lebo, Simon Miles, Stian Soiland-Reyes, for their thorough reviews.

The editors acknowledge valuable contributions from the following: Tom Baker, David Booth, Robert Freimuth, Satrajit Ghosh, Ralph Hodgson, Renato Iannella, Jacek Kopecky, James Leigh, Jacco van Ossenbruggen, Héctor Pérez-Urbina, Alan Ruttenberg, Reza Samavi, Evren Sirin, Antoine Zimmermann.

Members of the Provenance Working Group at the time of publication of this document were: Ilkay Altintas (Invited expert), Reza B'Far (Oracle Corporation), Khalid Belhajjame (University of Manchester), James Cheney (University of Edinburgh, School of Informatics), Sam Coppens (iMinds - Ghent University), David Corsar (University of Aberdeen, Computing Science), Stephen Cresswell (The National Archives), Tom De Nies (iMinds - Ghent University), Helena Deus (DERI Galway at the National University of Ireland, Galway, Ireland), Simon Dobson (Invited expert), Martin Doerr (Foundation for Research and Technology - Hellas(FORTH)), Kai Eckert (Invited expert), Jean-Pierre EVAIN (European Broadcasting Union, EBU-UER), James Frew (Invited expert), Irini Fundulaki (Foundation for Research and Technology - Hellas(FORTH)), Daniel Garijo (Universidad Politécnica de Madrid), Yolanda Gil (Invited expert), Ryan Golden (Oracle Corporation), Paul Groth (Vrije Universiteit), Olaf Hartig (Invited expert), David Hau (National Cancer Institute, NCI), Sandro Hawke (W3C/MIT), Jörn Hees (German Research Center for

Artificial Intelligence (DFKI) Gmbh), Ivan Herman, (W3C/ERCIM), Ralph Hodgson (TopQuadrant), Hook Hua (Invited expert), Trung Dong Huynh (University of Southampton), Graham Klyne (University of Oxford), Michael Lang (Revelytix, Inc.), Timothy Lebo (Rensselaer Polytechnic Institute), James McCusker (Rensselaer Polytechnic Institute), Deborah McGuinness (Rensselaer Polytechnic Institute), Simon Miles (Invited expert), Paolo Missier (School of Computing Science, Newcastle university), Luc Moreau (University of Southampton), James Myers (Rensselaer Polytechnic Institute), Vinh Nguyen (Wright State University), Edoardo Pignotti (University of Aberdeen, Computing Science), Paulo da Silva Pinheiro (Rensselaer Polytechnic Institute), Carl Reed (Open Geospatial Consortium), Adam Retter (Invited Expert), Christine Runnegar (Invited expert), Satya Sahoo (Invited expert), David Schaengold (Revelytix, Inc.), Daniel Schutzer (FSTC, Financial Services Technology Consortium), Yogesh Simmhan (Invited expert), Stian Soiland-Reyes (University of Manchester), Eric Stephan (Pacific Northwest National Laboratory), Linda Stewart (The National Archives), Ed Summers (Library of Congress), Maria Theodoridou (Foundation for Research and Technology - Hellas(FORTH)), Ted Thibodeau (OpenLink Software Inc.), Curt Tilmes (National Aeronautics and Space Administration), Craig Trim (IBM Corporation), Stephan Zednik (Rensselaer Polytechnic Institute), Jun Zhao (University of Oxford), Yuting Zhao (University of Aberdeen, Computing Science).

D. References

D.1 Normative references

[PROV-DM]
Luc Moreau; Paolo Missier; eds. *PROV-DM: The PROV Data Model*. 30 April 2013, W3C Recommendation. URL: http://www.w3.org/TR/2013/REC-prov-dm-20130430/

[PROV-N]
Luc Moreau; Paolo Missier; eds. *PROV-N: The Provenance Notation*. 30 April 2013, W3C Recommendation. URL: http://www.w3.org/TR/2013/REC-prov-n-20130430/

[PROV-O]
Timothy Lebo; Satya Sahoo; Deborah McGuinness; eds. *PROV-O: The PROV Ontology*. 30 April 2013, W3C Recommendation. URL: http://www.w3.org/TR/2013/REC-prov-o-20130430/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels.* March 1997. Internet RFC 2119. URL: http://www.ietf.org/rfc/rfc2119.txt

[RFC3987]
M. Dürst; M. Suignard. *Internationalized Resource Identifiers (IRIs) (RFC 3987)*. January 2005. RFC. URL: http://www.ietf.org/rfc/rfc3987.txt

D.2 Informative references

[CHR]
Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press URL: http://constraint-handling-rules.org/

[CLOCK]
L. Lamport. *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM 21 (7): 558–565. 1978. URL: http://research.microsoft.com/users/lamport/pubs/time-clocks.pdf DOI: doi:10.1145/359545.359563.

[DBCONSTRAINTS]
Ronald Fagin; Phokion G. Kolaitis; Renée J. Miller; Lucian Popa. *Data exchange: Semantics and query answering*. Theoretical computer science 336(1):89-124 Elsevier URL: http://dx.doi.org/10.1016/j.tcs.2004.10.033

[Logic]
W. E. Johnson. *Logic: Part III*.1924. URL: http://www.ditext.com/johnson/intro-3.html

[PROV-AQ]
Graham Klyne; Paul Groth; eds. *Provenance Access and Query*. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-aq-20130430/

[PROV-DC]
Daniel Garijo; Kai Eckert; eds. *Dublin Core to PROV Mapping*. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-dc-20130430/

[PROV-DICTIONARY]
Tom De Nies; Sam Coppens; eds. *PROV Dictionary: Modeling Provenance for Dictionary Data*

Structures. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-dictionary-20130430/

[PROV-LINKS]

Luc Moreau; Timothy Lebo; eds. *Linking Across Provenance Bundles*. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-links-20130430/

[PROV-OVERVIEW]

Paul Groth; Luc Moreau; eds. *PROV-OVERVIEW: An Overview of the PROV Family of Documents*. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/

[PROV-PRIMER]

Yolanda Gil; Simon Miles; eds. *PROV Model Primer*. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-primer-20130430/

[PROV-SEM]

James Cheney; ed. *Semantics of the PROV Data Model*. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-sem-20130430.

[PROV-XML]

Hook Hua; Curt Tilmes; Stephan Zednik; eds. *PROV-XML: The PROV XML Schema*. 30 April 2013, W3C Note. URL: http://www.w3.org/TR/2013/NOTE-prov-xml-20130430/