QuickRDA

Domain Model Interchange

Erik Eidt, Mathias Salle

Version 2.5

December, 2010

***Copyright (c) 2011, 2012 Hewlett-Packard Development Company, L.P.***

[Role-based Domain Architecture (RDA) 5](#_Toc336067036)

[Domain Model Interchange (DMI) 5](#_Toc336067037)

[DMI: Syntax using JSON 6](#_Toc336067038)

[Syntax for the Unit of Interchange 6](#_Toc336067039)

[Syntax for Quads 8](#_Toc336067040)

[The Fields of Quads as Tokens 10](#_Toc336067041)

[DMI Metamodel 11](#_Toc336067042)

[Logical Consequence of Applying a That operator to Triples 12](#_Toc336067043)

[Formal Description of Metamodel 13](#_Toc336067044)

[Background 13](#_Toc336067045)

[DMI Metamodel Diagram 14](#_Toc336067046)

[Diagram Discussion 15](#_Toc336067047)

[Values & Serialization 17](#_Toc336067048)

[Capabilities of Concepts 18](#_Toc336067049)

[Basic Entities 19](#_Toc336067050)

[Basic Properties 20](#_Toc336067051)

[On Names vs. Concepts 21](#_Toc336067052)

[On Internal Names vs. External Names 21](#_Toc336067053)

[Graph Properties 22](#_Toc336067054)

[DMI: The Unit of Interchange as a Container 23](#_Toc336067055)

[Container Overview 23](#_Toc336067056)

[Non-asserted section 23](#_Toc336067057)

[Asserted section 23](#_Toc336067058)

[Matter of Fact Assertions 23](#_Toc336067059)

[Strong Assertions 23](#_Toc336067060)

[Wrapping Not Necessarily Trusted Imports with Provenance 23](#_Toc336067061)

[Assertion 24](#_Toc336067062)

[Versions 24](#_Toc336067063)

[Collaboration using notion of Proposals 25](#_Toc336067064)

[Provenance 25](#_Toc336067065)

[DMI Container Boot Strapping 26](#_Toc336067066)

[Appendix A: Differences with DMI For Those Familiar with RDF 27](#_Toc336067067)

[Quads vs. Triples 27](#_Toc336067068)

[Consequences of Making Relations First-Class 28](#_Toc336067069)

[Assertion 29](#_Toc336067070)

[Use of Blank Nodes vs. IRI/URI 30](#_Toc336067071)

[Separation of Names vs. Concepts 31](#_Toc336067072)

[Explicitly Typed Literals 32](#_Toc336067073)

[Summary 33](#_Toc336067074)

[References & Other Readings 34](#_Toc336067075)

*Table of Figures*

[Figure 1. The DMI Metamodel 14](#_Toc336067076)

*Table of Tables*

[Table 1. Basic Entities of the Underlying Metamodel 19](#_Toc336067077)

[Table 2. Basic Properties of the Underlying Metamodel 20](#_Toc336067078)

[Table 3. Summary of Properties Relevant to the Container 23](#_Toc336067079)

[Table 4. Special Tokens 26](#_Toc336067080)

# Role-based Domain Architecture (RDA)

RDA, Role-based Domain Architecture provides a metamodel and methodology for the elicitation and capture of domain models at 4 levels, contextual, conceptual, logical and physical, with a notion of refinement that ties domain elements modeled at each layer to elements at the next.

RDA domain models are shared between various toolsets using the format describe here and known as RDA Domain Model Interchange.

## Domain Model Interchange (DMI)

RDA Domain Model Interchange supports the sharing of domain models using a notion of a unit of sharing, which contains a set of modeled elements and relations. Such a unit of interchange can be thought of as a single database. Interchange is designed to allow tools to easily combine sharing units, obviating the need to share multiple units for one interchange.

The underlying metamodel is rich, and provides for adornment. Specific adornments are grouped to provide additional capabilities; some of these specifically designed to support bi-directional or round trip sharing between tooling.

One group of adornments provides provenance for understanding the origination of the modeled elements (and relations). This provenance can be used to identify authors and source files, and, the modeled elements can described as such within a unit of sharing. Further, many domain models can be combined into a single database — a single unit of sharing — without losing origination information. This means that models can be stored as source files in one toolset, exported to another toolset, modified there, and later imported by the original toolset with the original source file configuration.

Another group of adornments provides for versioning and collaboration; these support expression of domain model changes and history by a mechanism of proposals. Sharing of models within a unit of interchange can thus include previous versions, a current version, along with proposed modifications.

The RDA Domain Model Interchange describes a unit of interchange whose format is based on:

* an underlying metamodel based on logic
* its container mechanism, and,
* a serialization using the syntax of JSON

# DMI: Syntax using JSON

This section describes the syntax for Domain Model Interchange, which is based on an underlying object model that is serialized to JSON.

The basic format of a unit of interchange is a single JSON array object that contains any number of JSON objects, where each entry in the array represents an element in the domain model, a quad. Like RDF’s triples, these quad objects are flat (they do not nest) yet are specifically designed to support composition, which, is done, as in RDF, by using multiple entries.

Note that all string values are expressed using the standard character encoding and escape character sequencing for JSON, meaning that Unicode characters are supported, the \ character is needed to escape embedded quote characters and to construct special characters such as line breaks and characters by hex code.

See [www.json.org](http://www.json.org) for more complete information about the encoding of characters and strings.

JSON/Javascript/Ecmascript comment syntax is allowed.

Beyond the literal interpretation of quads, one must understand the container mechanism and the underlying metamodel of the quads.

## Syntax for the Unit of Interchange

In DMI, a unit of interchange, unit-of-interchange, is expressed using JSON notation.

A unit of interchange consists of two sections, one for each of:

* the section of non-asserted quad
* the section of asserted quads

The unit-of-interchange is shared as a JSON object with two fields as follows:

{ “non-asserted” : quad-list, “asserted” : quad-list }

Where each quad-list looks like a JSON array (of quads where quads are described later) as follows:

**[** *quad*, *quad*, *quad*, *quad* **]**

Or, as more normally would be seen with optional line breaks:

[

*quad*,

*quad*,

*quad*,

*quad*

]

*The last quad before the closing bracket may be the JSON null object, and if present is ignored — this serves only to allow each actual quad to be terminated by a comma.*

**[** *quad*, *quad*, *quad*, *quad*, **null** **]**

## Syntax for Quads

Quads are elements of an array in a unit of interchange, and have:

* an *id*,
* a *subject*,
* a *verb*, and,
* one of either
  + an *object*, or,
  + a typed literal, consisting (minimally) of *type* and value

Quads can be represented in JSON using the object or array syntax shown below.

The id is optional and when not present a unique value is assumed.

The following list defines the possible forms for the quad. Items 1-4 present DMI JSON object syntax, while 5-8 present DMI JSON array syntax. Even numbers show optional id absent, while odd numbers show the optional id as present. 1-2 and 5-6 show last element in object form, while 3-4, and 7-8 show last element as typed literal. *(Numbering is for documentation only; spacing not normative, and is provided only for visual effect.)*

1. { “id”:”*id*”, “subject”: ”*subject*”, “verb”: ”*verb*”, “object”: ”*object*” }
2. { “subject”: ”*subject*”, “verb”: ”*verb*”, “object”: ”*object*” }
3. { “id”:”*id*”, “subject”: ”*subject*”, “verb”: ”*verb*”, “literal”: { “type”: ”*type*”, “value”: ”*value*” }
4. { “subject”: ”*subject*”, “verb”: ”*verb*”, “literal”: { “type”: ”*type*”, “value”: ”*value*” }
5. [ ”*id*”, ”*subject*”, ”*verb*”, ”*object*” ]
6. [ ”*subject*”, ”*verb*”, ”*object*” ]
7. [ ”*id*”, ”*subject*”, ”*verb*”, [ ”*type*”, ”*value*” ] ]
8. [ ”*subject*”, ”*verb*”, [ ”*type*”, ”*value*” ] ]

*TBD: In addition, a special syntax for collections is supported, but identical to the collection form when individual collection elements are enumerated in their own quad.*

Only the subset of the JSON syntax described here is considered in constitution of a legal unit of interchange in Domain Model Interchange.

## The Fields of Quads as Tokens

The values represented by id, subject, verb, and, object are tokens whose value has meaning as an identifier within the unit of interchange; these tokens can be numbers or descriptive names. A token is used as an identifier for and reference to a concept within the unit of interchange — the same character sequence refers to the same concept. Tokens have no meaning outside the unit of interchange. Tokens serve the same function as RDF blank nodes, and, even were the token to have the form of an IRI/URI, the scope of meaning is only the unit of interchange.

Identifiers used for reference are internal to the unit of sharing (like RDF blank nodes), and have no meaning outside that unit of sharing. Unlike RDF, even if identifiers take the form of IRI or URIs, this carries no meaning outside of the unit of sharing. However, a property can be used to associate an IRI or URI with an internal node, as well as a textual name, a qualifier, or an internationalize-able label and description.

The type field refers to a token that is a class, usually one of the built-in classes, such as String.

# DMI Metamodel

The underlying metamodel is based on logic similar to OWL with the addition of the “THAT” operator from IKL, and a formalization of the relationship between relations (statements as quads), properties, and classes. Concepts, given token as identifiers (internal to the unit of interchange), are described as having properties through the use of quads; one or more quads may be used to describe a concept.

Each concept in DMI can have qualities of classes, of instances, of relations and of properties. As classes we expect concepts to have super and sub classes, and instances. As instances we expect concepts to have names and types (classes). As relations we expect concepts to relate subjects via verbs to objects. As properties we expect concepts to have domains and ranges (linking properties to classes), and super and sub properties, and relations that use the property.

## Logical Consequence of Applying a That operator to Triples

Which mostly is that:

1. Triples become quads by adding a referencing capability (label), and, that
2. The quads (statements) as made now have an official type, which is Relation, and that
3. User-defined properties become classes — whose instances are Relations (statements, specifically the ones that use the property as a verb to connect subject and object), and that as a result,
4. We don’t need the notion of sub-properties separately from subclass since properties are classes, and
5. We have a (or in some sense don’t need a separate) mechanism to declare the “class for the statements using given property as their verb” (or predicate in RDF-speak); this is missing in RDF/OWL.

## Formal Description of Metamodel

Though the metamodel is relatively small, it can be very confusing. Because of this, we’re going to use two parallel approaches in describing metamodel. The first approach is to start with a background on the notion of class and instance and then, using UML, provide a formal diagram of the metamodel along with a discussion of the metamodel using UML terminology. The second way is to define the metamodel using its own native format, which results in a foundational set of classes & properties, similar to OWL, and axiomatic statements (quads similar to RDF triples).

### Background

In order to discuss the metamodel diagram, first let’s start with some terminology regarding the notion of class and instance and what we mean by using those terms.

Instances can represent individuals, anything really. Classes are used to identify a particular set of members, formally known as called instance members, or just instances, having commonality. This commonality among the instances can range from mere membership in the same class, to qualities that we expect all members to exhibit. Classes are used to describe this commonality once, instead of repeating it for every instance. We expect each instance of a class to exhibit its personalization of commonality of qualities that involve its class.

In UML, commonality of qualities is expressed with an Association between one class and (the same or) another class. An Association between two classes directly relates to a Link between two instances of those classes. End-point cardinality (and other restrictions) may be part of the Association: these apply to the expected and possible number of Links related to the Association.

In RDF, such commonality of qualities is expressed using Properties, whose Domain is one class and Range is (the same or) another class. When an instance is a member of a class then the properties of the class manifest themselves as relations between instances, which are statements or Triples of the form Subject-Predicate-Object. For a given triple statement, we expect the Subject to be an instance of the Domain of the Property and the Object to be an instance of the Range of the Property.

RDF does not directly give statements a type (there is the type rdf:Statement, but that is a parallel construct to the triples themselves); DMI, however, formalizes the notion of a relation, integrating it into the metamodel, and giving it its own class and instances.

Classes can have super and sub relationships with other classes. Instances of a sub class are also instance members of the super class. Thus, a subclass satisfies or matches a domain or range specification of its parent. Conversely, subclasses of instances of some class, X, are also considered instance members of X.

### DMI Metamodel Diagram

The following diagram captures the DMI metamodel using UML notation. It makes heavy use of the notion of class and instance, as you’ll see. One significant contribution of DMI over RDF is the integration of the statement itself directly into the metamodel as a first-class construct.



Figure 1. The DMI Metamodel

### Diagram Discussion

In words, what this is shown in Figure 1 follows. This discussion is entirely in terms of UML. (A description of the DMI metamodel defined it is own terms can be found in subsequent sections.) Words or phrases in bold refer directly to elements of the diagram.

1. There is a fundamental notion of Class and instances — classes are instances: a class has a type, which is **Class**, and, instances can also be classes. The foundation class for all instances is **Concept**; even Class is a subclass of Concept. Concept, which is a class, has as its type Class. Subclasses of Concept are also instances of Class. This relationship between the fundamental class, Concept, and the Class itself is the standard one found in object-oriented programming languages, for example , java.lang.Object and java.lang.Class. *OWL, even OWL 2, shies away from defining the fundamental class and instance this way, I believe due to the historical pressure to support known computable and efficient subsets (which manifest in the separate the levels of OWL: Lite, DL, full).*
2. Classes have a notion of sub class, and inversely — though not diagrammed — super class, expressed as the **Subclass Of** association. In this diagram, UML generalization notion is used to express Links of the Subclass Of association.
3. Concepts have zero or more names. A **Name** is first-class, so they are also Concepts. Many naming schemes can be created and used in parallel, including URIs and IRIs, namespace qualified names, product taxonomies, etc… The diagram shows namespace qualified names using **Simple Name** and **Qualified Name**.
4. **Property** is the class for defining properties. Domain Properties are instances of this class. The Property class is involved in two associations: **has Domain** and **has Range**, meaning that we expect instances of Property (which are Domain Properties) to manifest their personalization of these Associations as Links.
5. **Relation** is the class that begins to formalize the notion of quads — the DMI equivalent to RDF triples. The Relation class is involved in two associations: **has Subject** and **has Object**; instances of Relation (which are Statements) will have their personalization of these associations.
6. Each **Domain Property** is an instance of Property, as well as subclasses of Relation. From being an instance of Property, Domain Properties personalize the Domain and Range associations by specifying links: **has Domain** and **has Range**. From being a subclass of Relation, Domain Properties are classes having instance members with a Subject and Object. Domain Properties apply of the type in their links **has Domain** and **has Range** in refining the class or type of Subject and Object provided by Relation.
7. A **Statement** is an instance of Domain Properties. Since Domain Properties are subclasses of Relation, which is a subclass of Concept, thus, Statements are instances of Concepts. We note that this formalization of Statements as Concepts, which is a unique contribution of DMI over RDF, allows Statements to be referred to. Thus, the four elements forming a quad are: the subject Link, and the object Link exhibited from being instances of Relation, their type specification as an instance of a Domain Property (the predicate), and the ability to refer to them by being instances of Concept.
8. The notion of sub properties is obtained through the use of the subclass association applied between user-defined properties. There is no separate or other notion of sub or super property is necessary.
9. It may be appreciated that additional classes similar to Relation could be introduced to allow for Statements of other, more complex forms beyond subject-predicate-object.
10. Note: Metamodel Associations describing min and max cardinality for Property need to be added.

#### Values & Serialization

The notions of Values & Value Types are not fully described as yet. The intent of DMI is to clearly separate the concerns of conceptual value definition from their serialization.

XML Schema, for example, conflates the notion of values in a value space with the serialization of such, making it difficult to both properly and usefully define the notion of equality among values. RDF builds on this conflation by limiting serialization to a single string, while adding additional characters to that string for the XSD type and internationalization specification (then requiring specialized query extensions to access these second-class embedded fields). As DMI is specifically designed as an interchange format, it addresses this conflation by separating the notion of the definition values and their value spaces from the serialization format used.

#### Capabilities of Concepts

##### As Instance

All entities are an instance of the class Concept. As an instance of Concept, an item has one or more types, which are Classes.

##### As Class

Any entity can operate as a class, having Class as its type. As an instance of Class, an item can be involved in sub and super class relationships, and can have members, instances, and can commonality among those instances in the class can be described via properties.

A Domain Property is also a class representing the statements using it as their Class.

Even Values that are instances of Value Types can themselves be classes; when they are they refer to they are singletons that refer to the value. Thus, for example, one property can have a Range of Boolean, and, another can have the Range of True.

##### As Property

A Domain Property is an instance of Property and a subclass of Relation. As a class, their members are instances, which are statements that use the property as a predicate.

## Basic Entities

The following concepts for entities are defined in the metamodel.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Qualifier*** | ***Name*** | ***Type*** | ***Description*** |
| DMI | Concept | Class | The base type for everything |
| DMI | Nothing | Concept | The non-value |
| DMI | Name | Class | The type for externally visible names |
| DMI | Literal | Class | The base type for defining value spaces(\*tbd) |
| DMI | Class | Class | The type for classes |
| DMI | Property | Class | The type for properties |
| DMI | Relation | Class | The type for relations, which are instances of properties |

Table 1. Basic Entities of the Underlying Metamodel

The concept Property itself has type Class; instances of Property are properties; properties themselves have instances, which are relations. Each of the properties is itself also a class that represents its instances (which are relations). Sub-properties are specified as subclasses.

## Basic Properties

These properties are defined in the metamodel — they are qualified by com, hp, DMI (e.g. in the com.hp.DMI namespace).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Qualifier*** | ***Name*** | ***Domain*** | ***Range*** | ***Description*** |
| DMI | Simple Name | Name | Literal | Associates Subject with an externally visible simple name |
| DMI | Qualifier | Name | Name | Associates Subject with an externally visible qualifier name |
| DMI | Has Name | Concept | Name | Associates Subject with Name |
| DMI | Description | Concept | Literal | A textual description of the subject |
| DMI | Instance Of | Concept | Class | Subject is an instance of Object class |
| DMI | Subclass Of | Class | Class | Subject is subclass of Object |
| DMI | Domain | Property | Class | Subject is a property whose domain is Domain |
| DMI | Range | Property | Class | Subject is a property whose range is Range |

Table 2. Basic Properties of the Underlying Metamodel

### On Names vs. Concepts

Concepts have zero or more names; names are used refer to concepts, especially across units of interchange.

Names themselves are also first-class concepts. Since names are concepts, names can be related to other names, to other concepts; thus, names can have a rich structure and participate in taxonomies, independent of the concepts they refer to.

A concept can have multiple names, within and especially across the variety of human languages.

The same name can be used for multiple concepts; though if used, one should be aware that at present, DMI provides no additional explanation as to which meaning is intended to be associated with which usage of an external name beyond that of simple matching of external names.

### On Internal Names vs. External Names

DMI distinguishes the token used to refer to a concept within the unit of interchange from an external name for that concept. Since a token is internal to the unit of interchange, like blank nodes in RDF, tokens do not require global uniqueness. To be able to match concepts between two or more units of interchange, units of interchange must specify external names for the concepts, by formally relating the internal token to the external name.

To facilitate certain operations like debugging, concepts, which are referred to by internal tokens, may be given a label using the label property (qualifier=MM, name=Label, Domain=Concept, Range=Literal). This gives the authors of units of interchange the choice of using readable strings for the internal tokens, or meaningless numbers while still allowing for a text identifier for the token. Only one such label per token seems sensible, though this is not mandated; there is no normative usage of such a label, either internal or external to the unit of interchange.

## Graph Properties

In addition, graphical qualities are also valuable descriptions of elements

* Logical Graphing
  + Dimension
* Physical Graphing
  + Style
  + Text Color
  + Fill Color
  + Line Weight
  + Position

# DMI: The Unit of Interchange as a Container

In DMI, quads representing relations (i.e. statements) are not automatically asserted. This feature is important for versioning and collaboration — multiple versions can coexist side-by-side in a unit of interchange, and for provenance — provenance can be asserted over unasserted imported statements, and to build complex asserted statements out of smaller unasserted statements.

Quads are asserted by reachability from a given root. A unit of interchange has a primary root node, which describes the assertions quads as well as the conditions for their assertion. Statements are grouped into classes to form anonymous or named versions; these versions can be unconditionally or conditionally asserted. Since versions are classes, versions participate in hierarchies; asserting a sub class also asserts its parent classes.

This is done through relations on the root node using the following properties.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Qualifier*** | ***Name*** | ***Type*** | ***Domain*** | ***Range*** | ***Description*** |
| DMI | Asserts | Property | Concept | Concept |  |
| DMI | Denies | Property | Concept | Concept |  |
| DMI | Sources | Property | Concept | Class | Subject asserts relations in the Object class. |
| DMI | Version | Class |  |  | The base class for versions. |

Table 3. Summary of Properties Relevant to the Container

## Container Overview

First, we assert the metamodel elements we’re using; this is similar to imports in DLLs. Then we assert the versions (as classes) and their memberships. Then we ascribe them to the root element, which is taken as asserted.

### Non-asserted section

### Asserted section

#### Matter of Fact Assertions

#### Strong Assertions

### Wrapping Not Necessarily Trusted Imports with Provenance

Graphic showing that asserted quads are de-asserted, collected (which is asserted as matter of fact) and strongly asserted with provenance about origination.

Graphic showing recursive application of this pattern.

## Assertion

Asserting a quad means that the statement made by the relation is taken to be true. Asserting a class means taking all instance statements. (Note that one must assert each instance of a class separately.)

TBD: What does asserting an instance or a class mean?

Relationship between assertion and non-assertion.

## Versions

As classes that are asserted.

## Collaboration using notion of Proposals

Version composed of base+ changes (additions/deletions)

## Provenance

Authorship

Source File

Asserts…

Diagram/View

Differences between diagram/view, source file, version.

## DMI Container Boot Strapping

Fields of quads (that don’t refer to literals) refer to internal tokens that have no meaning outside of the unit of interchange. Shared terminology must be “imported” — this includes even the DMI metamodel terminology.

This presents a bootstrapping problem, which is solved by taking the special empty name to have a meaning — actually, several meanings, depending on the context. If blank appears in the verb position of a quad whose object is a literal, then the verb is taken to be the metamodel property “Simple Name”.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Name*** | ***Position*** | ***Quad Form*** | ***Description*** |
| Blank | Verb | Object | Blank refers to the “Qualifier” property of the metamodel. |
| Blank | Verb | Literal | Blank refers to the “Simple Name” property of the metamodel. |
| Blank | Type | Literal | Blank refers to the “String” class of the metamodel. |

Table 4. Special Tokens

Separation of names vs. concepts.

# Appendix A: Differences with DMI For Those Familiar with RDF

## Quads vs. Triples

In RDF, statements are triples. Statements express relations; however, statements themselves are not a first-class construct: relations cannot be involved in relations. To put a relation into a relation, a parallel construct (usually using classes and instances) to the statement mechanism must be created. Even then, there is then no specific way to relate the asserted statement to this parallel construct. Class and instance (individually first-class notions) form a valuable pair; however, no similar pairing notion exists between properties (a first-class notion) and triples.

RDA DMI uses quads, which are relations, a first-class construct. Relations can be the source or target of other relations. In the elevation of relations to first-class, we add an identifier to relations. To complete the first-class integration of relations, we formalize the property-relation pair, by tying them to the class-instance constructs. A relation is a usage of a property. There is a class that represents the set of relations that are usages of a given property. (This class expands to include sub-properties, and usual class-instance reasoning applies.)

## Consequences of Making Relations First-Class

As a result of promoting relations to a first-class concept, properties now have instances, which are relations that use the property. Relations have a type, which is that of the property; properties are classes that represent collections of relations — in the same way that any class represents the collection of its instances.

Given that the notion of first-class relations turns properties into classes, the notion of sub-property is subsumed by the notion of subclass. A subclass of a property is a property.

However, a parent of a property is not necessarily also a property; the parent is not automatically a property merely due to being the parent of some property — to be a known as a property, the parent (or any property) should exhibit qualities of properties, such as having a domain and range or being a subclass of a known property.

## Assertion

Statements in RDF are asserted. The only way to make un-asserted statements is through the ill-defined and cumbersome reification mechanism. A graph mechanism where all statements are assumed to be true makes it difficult to do versioning, to share graphs of statements that are not necessarily true but were known to be obtained by some source.

In RDA DMI, statements are not automatically asserted. The graph must contain explicit assertion, which is accomplished by membership in root set. Only statements reachable from the root are asserted, and, they are only asserted in the context from which they are reached. Conditional (declarative) reachability is supported.

We can store versioned information side-by-side; since statements are only asserted through reachability, the one version of information does not conflict with another.

We can make statements like “the sky is blue during the day”, using two quads, and only asserting the 2nd qualified form; thus not universally declaring that the sky is blue.

We can give provenance to statements, asserting only the provenance, but not the statements themselves; this can be applied recursively.

## Use of Blank Nodes vs. IRI/URI

In RDF, all references are done using IRI/URIs that have meaning not just within a graph but between graphs; this allows one graph to easily be merged with another graph. However, versioning of information then requires the introduction of parallel identifiers.

In RDA DMI, all references within the quads are done using the RDF equivalent of blank nodes that are internal to a unit of exchange. These references are not persisted between units of sharing (graphs). Sharing and graph merging is done by explicitly giving external, persistent names (in namespaces) to references. This creates a level of indirection between the use of nodes within the graph and the external sharing of persistent names. This level of indirection is used to support versioning and collaboration.

In RDA DMI, the element naming is a first-class notion, and thus features of names, such as namespaces, can be expressed in the same logically structured way that other information is conveyed. This means we can search for elements in a namespace — this is not possible in RDF without resorting to regular expression matching on the IRI/URI value.

## Separation of Names vs. Concepts

DMI provides a separation of concerns elevates names to a first-class construct and as a result have an official type and structure independent of the concepts to which they refer.

First-class naming supports the use of more than one naming scheme, for example, one of which could be is URI and another of which could be namespaces. Even the URI could naming scheme can be expressed with structure instead of simply being an opaque and cryptic long string. As a result, there is a formal mechanism for searching names instead of RDF’s informal searching thru the text of URIs.

Schemes can be defined that allows first-class access to both the simple and the namespace qualifier for a name; thus the hierarchy in qualified names can be visible to the metamodel rather than outside the metamodel as in RDF where URI are captured as strings with domain names and slashes. Domain-specific naming scheme can be used. Further, translation is supported in that 1) a concept can have more than one name, and, 2) names themselves can be put in relations (e.g. with other names).

## Explicitly Typed Literals

In RDF, certain literals are represented using a string; various type information may be encoded within and along with actual value in the string; this includes a language identifier for strings, and a type specification for numbers and dates.

In RDA DMI, literal type information is separated from the actual value. This separation allows us to refer to the actual value of literals with less parsing and encoding. It also supports extensibility: for example, future versions may support expression of unit, precision, and accuracy for numeric literals without accidentally misinterpreting previously existing encoded strings (or other literals?).

## Summary

Flexibility is provided by

* the use of blank nodes instead of universal ids,
* the use of assertion by reachability instead of automatic assertion,
* the use of relations (quads) as a first-class construct

These features combine to supports provenance, versioning and collaboration, and a variety of other interesting expressions.

# References & Other Readings

BLOGIC, Pat Hayes (Talk on what can be improved in RDF)

<http://videolectures.net/iswc09_hayes_blogic/>

Axiomatic Triples in RDF

<http://www.w3.org/TR/2009/REC-owl2-rdf-based-semantics-20091027/>

IKL

<http://www.ihmc.us/users/phayes/IKL/GUIDE/GUIDE.html>

Common Logic

<http://common-logic.org/>

RDF

<http://www.w3.org/standards/techs/rdf#w3c_all>

OWL

<http://www.w3.org/standards/techs/owl#w3c_all>

Lazysoft Associative Model, Sentences Database

<http://www.lazysoft.com/docs/other_docs/amd_whitepaper.pdf>