The QuickRDA 3.0 Platform

Erik Eidt

Version 3.0

April, 2011

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

[Introduction 5](#_Toc335636519)

[Basic Structure 6](#_Toc335636520)

[Everything is a Concept 6](#_Toc335636521)

[The Concept Cache 7](#_Toc335636522)

[Class to Instance Cached Linkages 7](#_Toc335636523)

[Class to Class Cached Linkages 8](#_Toc335636524)

[Statement Cached linkages 8](#_Toc335636525)

[Property to Statement Cached Linkages 9](#_Toc335636526)

[Property to Domain & Range Cached Linkages 9](#_Toc335636527)

[Subject and Object Usage Cached Linkages 10](#_Toc335636528)

[Performance 11](#_Toc335636529)

[Lists as Vectors 11](#_Toc335636530)

[Concept and Statement and Serialization 11](#_Toc335636531)

[Memory 12](#_Toc335636532)

[Time 12](#_Toc335636533)

[Reasoning 13](#_Toc335636534)

[Built-In Reasoning 13](#_Toc335636535)

[Basic Reasoning 13](#_Toc335636536)

[Contextual Conditionality Cached Linkages 13](#_Toc335636537)

[Output Reasoning 14](#_Toc335636538)

[Visibility 14](#_Toc335636539)

[Diagramming Reasoning 14](#_Toc335636540)

[RDA-specific Reasoning 14](#_Toc335636541)

[Containment Reasoning 15](#_Toc335636542)

[Filtering Language 16](#_Toc335636543)

[Abstraction Language 17](#_Toc335636544)

[Reporting Language 18](#_Toc335636545)

[Additional Capabilities 18](#_Toc335636546)

[General Operation 19](#_Toc335636547)

[Graph 20](#_Toc335636548)

[Subgraphs 20](#_Toc335636549)

[View 20](#_Toc335636550)

[Concept Manager 21](#_Toc335636551)

[The Builder 22](#_Toc335636552)

[Bootstrapping 22](#_Toc335636553)

[Base Vocabulary 22](#_Toc335636554)

[Loading Metamodels 22](#_Toc335636555)

[Loading Domain Models 22](#_Toc335636556)

[Generating Output 22](#_Toc335636557)

[References & Other Readings 23](#_Toc335636558)

*Table of Figures*

[Figure 1. Concept, Serialization, and Statement 6](#_Toc335636559)

[Figure 2. Class-to-Instance Cached Linkages 7](#_Toc335636560)

[Figure 3. Class to Class Cached Linkages 8](#_Toc335636561)

[Figure 4. Subject, Verb, Object Cached Linkages 8](#_Toc335636562)

[Figure 5. Property-to-Statement Cached Linkages 9](#_Toc335636563)

[Figure 6. Property-to-Domain & Range Cached Linkages 9](#_Toc335636564)

[Figure 7. Subject and Object Usage Cached Linkages 10](#_Toc335636565)

[Figure 8. Attach and Group Cached Linkages 15](#_Toc335636566)

[Figure 9. Builder Environment 19](#_Toc335636567)

*Table of Tables*

**No table of figures entries found.**

# Introduction

This document gives a high level overview of the architecture of the QuickRDA 3.0 code base, which provides a platform for information capture, modeling, and reasoning. It serves as a platform for two different kinds of language:

The first kind is characterized as domain languages, which are declarative languages for modeling domain-specific concerns. Domain languages themselves are expressed as models, i.e. they are metamodels — models that define the language for domain modeling. Domain languages are defined in terms of an underlying metamodel that provides a number features easing development of metamodels, such as expressing containment, abstraction, and diagramming properties. At present, there is a set of domain languages supporting Role-based Domain Architecture.

The second kind is characterized as reasoning languages capable of query and manipulation of models. Ideally, reasoning languages are general purpose, being properly abstracted to work across all the domain languages. However, reasoning languages can be developed as hard-coded to a specific domain language, perhaps as a first step toward more general applicability. Also ideally, the reasoning languages are high level and directly usable by domain modelers rather than inferencing specialists. Thus our languages are higher level than SPARQL or SPIN. At present, there is a set of languages for filtering, abstraction, containment, and reporting.

Statements within one reasoning language compose (e.g. filtering is done with multiple statements), and languages compose in the sense that you can filter after filtering, or abstract before and/or after filtering. The reasoning languages are implemented through imperative code and have access to the rich information cache of the underlying metamodel and internal construction of the QuickRDA platform. An area for possible research is language supporting the construction of higher level reasoning languages out of lower level reasoning languages, eliminating the need for imperative coding for each new reasoning language; this might be done by supporting something like SPARQL or SPIN; though at present we’re concentrating on the higher level reasoning languages.)

At present, QuickRDA 3.0 is a core framework that does not provide an editing experience.

QuickRDA 3.0 provides persistence capabilities; it can read from multiple inputs sources, combining them for visualization or creating new output, as well as keeping them separate for persisting changes in those individual input sources. Persistence is in the form of Domain Model Interchange (DMI). DMI defines a Unit of Interchange, which is a data container for logic statements. In addition to providing for the raw expression of statements, the container also gives context to those statements: by grouping statements to be taken together in a paragraph analogy; by defining externally referenced (and reference-able) concepts; and by separating asserted vs. non-asserted statements. More details on the specifics of the container format can be found in QuickRDA Domain Model Interchange.

# Basic Structure

This section describes the internal design from a data/object perspective.

## Everything is a Concept

The first thing to understand is that it treats everything as a Concept. There are two specializations of Concept, namely Serialization, and Statements.



Figure 1. Concept, Serialization, and Statement

Concepts represent the notions of domain concepts, such as Role or Responsibility from Business Context Architecture of Role-base Domain Architecture (RDA).

Serializations represent typed values serialized as strings and are used to express constants such as names.

Statements represent an association of a subject, verb, and object into a relationship. DMI defines a set of well-known concepts (such as Class) and verbs (such as Is-An-Instance-Of) for a logic foundation that is known as the Underlying Metamodel.

## The Concept Cache

It is important to note that in DMI, Concepts are understood solely through the statements that describe them. Thus, in the DMI unit of interchange container, only serializations and statements are represented!

Statements are used to associate concepts with concepts, such as to indicate that one concept is an instance of another concept. Statements are also used to associate concepts with serializations, such as to indicate the name of a concept.

The QuickRDA 3.0 code base provides necessary understandings of concepts by interpreting the logic statements and assembling them into an understanding represented in memory as a Cache that is associated with each Concept.

The following sections detail the most significant cached information from the perspective of information modeling rather than as object orientation; however, to be sure, all of this cached information is accessible via methods on the core concept. Much of the information is in list form, which can be directly obtained, or in some cases that are known to be common patterns, searched for matches given some search criteria.

### Class to Instance Cached Linkages

The underlying metamodel provides the logic foundation for classes and instances, and verbs that express class to instance relationships, for example, Is-An-Instance-Of. One feature of the Cache is the in-memory linkage between Concepts and their Classes. This linkage is manifest bi-directionally.



Figure 2. Class-to-Instance Cached Linkages

All concepts are individuals in the sense that they are instances of types (all concepts are instances of the primordial concept, Concept). Thus, all concepts have a list of classes that represent their types. A concept that is known to be a type is an instance of the primordial class, Class. Concepts representing classes have a list of instances.

### Class to Class Cached Linkages

The complete, transitive list of super classes and sub classes is also maintained.



Figure 3. Class to Class Cached Linkages

### Statement Cached linkages

The logic foundation provides a notion of statement, which is an association of a subject, verb, and object. *(Note that this is an area of extensibility; other kinds are of statements possible, though not yet specified, for example: subject, verb; and subject, verb, direct object, and indirect object.)*

A statement refers to one each of subject, verb, and object; these are captured as attributes of the Statement specialization of concept, and cached as linkages.



Figure 4. Subject, Verb, Object Cached Linkages

### Property to Statement Cached Linkages

Statements have subjects, verbs, and objects. Note that the term used to refer to a verb outside of its usage in a statement is: Property. A property is used in the verb position of a statement (in RDF a verb is called a predicate).

The underlying metamodel provides the logic foundation for properties and statements, which is that concepts used as verbs in statements are properties, and further, that properties are classes whose instances are statements. Thus, the class-to-instance cache linkage shown above applies directly to properties and statements and they take on meaning as follows: the list of types for a statement concept is the list of properties that apply to that statement. The list of instances for a property is the list of statements in which the property is used as a verb.



Figure 5. Property-to-Statement Cached Linkages

(Note that the code base captures only the class-to-instance cache linkages and that the property-to-statement cache linkages shown above are understood meanings when the class is a property and the individual is a statement; these understood meanings are codified in the underlying metamodel.)

### Property to Domain & Range Cached Linkages



Figure 6. Property-to-Domain & Range Cached Linkages

### Subject and Object Usage Cached Linkages

In the previous section, we described the cached linkages that arise from the verb of a statement. Further linkages of concepts to statements come from the subject and as object. Each usage of a concept as a subject and each usage of a concept as an object is cached as shown in the diagram below.



Figure 7. Subject and Object Usage Cached Linkages

## Performance

### Lists as Vectors

The cached linkages lists described above are stored using an internal Vector object. Performance and memory utilization are closely related to the Vector implementation.

### Concept and Statement and Serialization

The primordial concepts of Concept and Statement and Serialization (each is a class) are heavily stressed since everything is a concept: the list of instances for Concept contains every concept, serialization, and statement in the graph. The list of instances for Statement contains every statement in the graph, and similar for Serializations.

Because of this, and since this doesn’t really capture usefully needed information, the code can run in a mode where it omits generation of the cached linkages for these special and primordial concepts of Concept, Statement, and Serialization. If you query a concept to see if it is an instance of Concept or a statement if it is an instance of Statement, the answer will be yes as a special case handled in this mode; however, the list of types will not contain Concept, and Concept’s list of instances will not contain all the concepts in the graph.

### Memory

Any concept can be an individual, a class, a property, a statement or all of these at the same time. Thus, we need an accommodating design. Currently, a place holder (reference) for each list of cached linkage is allocated for each concept, making it simple to change an object representing an individual concept to one representing a class concept.

(Polymorphism could be used to reduce memory requirements, but morphing would be needed and references would have to be updated. Alternatively, class and property oriented lists could be stored in an addendum data structure not allocated until needed, as a space over speed tradeoff.)

Generally speaking, the vectors are only constructed when the first element is added to a list, so, for example, a concept that is not used as a class does not have lists (and instead just the placeholder reference) associated with classes (such as instances, super classes, and sub classes). The query Interfaces return a shared or common default empty list when asked, allowing vector creation to be fully delayed until the first element of the list is to be added.

### Time

The base algorithms typically spend time checking for duplicates when adding to the vectors. Thus, when vectors are frequently iterated over and are sufficiently long they are automatically converted to hash tables having the same interface. Most of the cache is constructed in an additive manner that doesn’t require removing an element from a list.

However, some of the algorithms (not the cache itself) use the vectors and do remove elements from them. Removing an element from a vector destroys the hash table, which is regenerated if the criteria are met again; this is automatic, so callers don’t have to worry about it to some extent. Generally speaking cleaning up a vector after removal of an element is a delayed operation having performance facilitated by leaving a null in the place; lists having nulls can subsequently be trimmed to remove them, which is more expensive. For this reason, algorithms that remove elements from lists accommodate nulls, and trim en-mass after running.

In particular, the list of statements associated with the visible and the non-visible subgraphs use vectors, and hiding or revealing a concept is a matter of moving it from one subgraph to another, which is implemented as removing an element from one list and adding it to another. This introduces a list removal operation, which as mentioned is done simply by replacement with null and subsequently trimmed en-mass later.

# Reasoning

The QuickRDA 3.0 code base supports a number of different kinds of reasoning.

Some of the reasoning is built-in, and, further reasoning is that more generic is supported by reasoning languages.

## Built-In Reasoning

### Basic Reasoning

Construction of the cache describe above is done by built-in reasoning over statements involving the well-known concepts and relationships of the underlying metamodel (such as Class and Concept) and relationships (provided by Statements). Concepts such as properties, domains, ranges, and associated properties, Has Domain, Has Range, etc… are reasoned over with the objective of constructing the cache. In short, each statement is considered one at a time as to what information is added to the cache.

#### Contextual Conditionality Cached Linkages

The current implementation caches information regarding asserted statements. However, for explicit versioning, and for time-based variation (4D modeling), multiple truths need to co-exist. To support multiple co-existing and yet potentially conflicting truths, a notion of conditionality is designed and can be introduced into the cache. This conditionality is architected as follows. In list caches, the list of concept references is replaced with a list of pairs; these pairs consist of a condition and the original concept reference. During construction of the cache, the building environment will maintain the notion of a condition (as an internal object); these conditions can represent either versions or points in time in which a single consistent truth applies. During query of the cache, such as for the set reasoning described herein, a condition must be provided that represents the context (version or point in time) for which truth is sought, providing a condition to match in evaluating the relevance and applicability of each cached linkage pair to the given context.

### Output Reasoning

#### Visibility

In order to output a diagram, some additional reasoning is necessary. Only visible elements will be diagrammed. Thus, the various metamodels will not appear in diagrammed output; various filtered and abstracted concepts will not appear in the diagrammed output. Visibility is a property of the subgraph, which is detailed in the section on general operation. It only makes sense to output statements whose subjects and objects are also visible, and so, non-conforming statements are moved to an invisible subgraph. (The verb becomes the label of the relationship between subject and object. The verb itself does not need to be visible in order for the statement to be output.)

#### Diagramming Reasoning

#### RDA-specific Reasoning

At present, there is some RDA specific reasoning. When a responsibility is decomposed into other responsibilities, and is assigned to the same role that as its parent responsibility is, and then we choose not to diagram the additional assigned to relationships as they just add clutter to the graph. It would be preferable to make this an automatic feature of the metamodel rather than specific, hard-coded logic applied for RDA.

### Containment Reasoning

One concept may attach (other) concepts, and, a concept may group (other) concepts. Further, when one concept, X, groups another concept, Y, some of the relationships that Y has with yet other concepts, Z, should also be grouped within X. These relationships are expressed in a metamodel by having relevant properties establishing concept relationships appropriately subclass from the well-known properties of attachment, grouping, and group following.

Containment reasoning establishes cached linkages between concepts by maintaining, for each concept, a list of attached concepts, and a list of grouped concepts. These lists are bi-directionally maintained, such that for each concept there is also a list of attaching concepts and a list of grouping concepts.



Figure 8. Attach and Group Cached Linkages

Construction of the containment cache is optional, and controlled by a build switch. When this cache is present, the diagramming capability will automatically apply visually indications of this containment when applicable. (It will not diagram containment when the underlying graphic language does not support it, such as the case for AT&T’s DOT language when the same element is in two different groups or the same element is attached to two different concepts.)

Containment reasoning is on by default, and as well, by default containment reasoning also hides the properties that cause containment from the visible graph, leaving the visualization of the containment to be diagrammed rather than the relationship that caused it. Of course, that hiding can be disabled, so we can visually see the containment and the relationship causing the containment on the same diagram.

## Filtering Language

The filtering language identifies concepts (non-statement concepts, and statement concepts) that should be revealed (made visible) to the output. There are a number of statements that reveal a single node or adjacency.

However, the Reach statement searches for all available paths between the specified source and target sets. The algorithm has a notion of a reaching path candidate, which is a list of elements that are directly connected in a path, alternations of concepts and relationships that connect them. Multiple reaching path candidates are concurrently tracked. The algorithm iterates; each iteration looking for successful paths and also creating new path candidates for the next iteration. The algorithm is complete and terminates when no new candidate paths have been created for the next iteration.

The initial reaching path candidates are populated from the source set: one path candidate (of initial length one) from each element in the source set. The source set is specified by the language statements, commonly a Select statement; the target set is specified by the Reach statement.

Successful path candidates are those reaching any element of the target set; more specifically, those whose end-most concept in the candidate path is actually in the target set. Upon detection of success, the concepts along the candidate path are revealed to the visible graph (the main function of filtering), and the candidate path is retired from the algorithm (in the sense that it does not serve as a base from which to generate new candidate paths).

Ever longer new path candidates are generated from the old candidate paths; the old candidate paths being retired in favor of the new candidate paths. Zero or more new candidate paths can be generated from each one of the old candidate paths. The new candidate paths are all one unit longer than in the previous iteration. Each new candidate path is has one more element on the end; this new element is directly reachable from the last (end) element of the old candidate path. New path candidates are created within constraints, namely the new element must not match those specified in Avoid statements and in addition, must not already on the old candidate path, which is to say that cycles are not allowed in a candidate path.

Its performance is relative to the size of the graph that it will reveal. There is some redundancy in the candidate paths; this redundancy would be difficult to eliminate, since what needs to be revealed is all possible paths, even when the paths cross or overlap!

## Abstraction Language

The abstraction language provides a simpler view by eliminating concepts from the visible graph, and promoting relationships to concepts that are to be abstracted to their parents. The language is simple; it specifies the concept(s) for targets of abstraction. The key to the success of abstraction is in expressing composition and decomposition in a metamodel or domain language.

For example, the metamodel for RDA’s Business Contextual Architecture needs to capture that the concept of Responsibilities represents decomposition of the concept of Roles. This is captured by declaring the Is-Assigned-To property, which relates Responsibilities to Roles, as a subclass of the well-known Decomposes property.

An additional statement for abstracting connections between like concepts (such as Roles interacting over artifacts) needs to be developed.

*(Note: For those familiar with RDF: the RDF notion of sub property is, in DMI, subsumed by the notion of sub class, since in DMI, properties are classes and thus have the full expressive power of classes.)*

## Reporting Language

The reporting language is a work in progress. It currently uses a derivation of Excel formulas to assess the quality and completeness of various portions of domain models. The formulas are specified on the various concepts to be evaluated in the report.

## Additional Capabilities

# General Operation

This section describes additional implementation classes involved in reading, reasoning over and manipulating, and writing logic statement in DMI.



Figure 9. Builder Environment

## Graph

A Graph represents *merged* inputs. A Graph object collects subgraphs.

## Subgraphs

A Subgraph contains a collection of serializations, concepts (that aren’t serializations or statements), and statements. Each subgraph has a set of Boolean properties, such as for indicating whether or not it contains domain model information or metamodel information.

The subgraph provides an interface to:

* Add a Serialization given a concept reference, a value, and type
* Add a Concept given concept reference
* Add a Statement given a concept reference, a subject, verb, and object.

The subgraph routines require a new concept reference generated by the caller; and in contrast to concept manager routines, they always add to the (sub) graph. The subgraph also provides the basic inferencing to build cached linkages; by default it does this as statements are added to the subgraph; however, it can be run in a mode where basic inferencing can be delayed until explicitly requested such as after loading the subgraph.

## View

A View represents a selection of concepts and relationships from a graph — this selection is a manifestation of the desired visible part of the graph to use as input to reasoning and/or as input to generation of export or diagramming. A view can be taken to the DMI textual representation and shared or persisted; a view can be taken to a diagramming language such as DOT for GraphViz.

## Concept Manager

The Concept Manager manages names for concepts through the name cache. The concept manager also provides for redundancy elimination via duplicate detection and prevention.

The concept manager provides an interface to:

* Lookup concepts by name
* Enter Serializations by string value and type
* Enter Concepts by name
* Enter Statements by subject, verb, and object

Concept can also be searched without entering them. The Enter routines check for and prevent redundancy in concepts and statement. Regardless of this duplicate elimination, the Enter routines return a reference to the concept (whether serialization, regular concept, or statement). Newly entered concepts by name are cached for subsequent lookup.

## The Builder

The builder class coordinates reading, reasoning, and writing. It creates the other classes as needed, including the main graph object and subgraphs.

## Bootstrapping

The QuickRDA 3.0 code base inflates the underlying metamodel, associating the primordial concepts and relationships with their in-memory representations as Concepts.

## Base Vocabulary

Bootstrapping provides convenient access to underlying metamodel concepts and relationships such as Class, Concept, Property, etc. stored in a base vocabulary object.

## Loading Metamodels

Metamodels are loaded into metamodel tagged subgraphs. They supply the definitions of concepts and relationships for subsequent domain models to use.

## Loading Domain Models

Domain models are loaded into domain tagged subgraphs.

## Generating Output

A default view is constructed based by the builder; build switches are processed, and govern subsequent actions. Filters are run other reasoning is performed; diagram output is generated; DMI output is generated.

# References & Other Readings

##### QuickRDA Domain Model Interchange