The QuickRDA 4.0 Patterns Guide

Erik Eidt

Version 4.0

April, 2011

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

[Introduction 5](#_Toc335636569)

[The QuickRDA 4.0 Build Table Commands for Pattern Language 6](#_Toc335636570)

[PatFile 6](#_Toc335636571)

[Apply 7](#_Toc335636572)

[The QuickRDA 4.0 Default Patterns 8](#_Toc335636573)

[The QuickRDA 4.0 Pattern Language 9](#_Toc335636574)

[Pattern Language BNF 11](#_Toc335636575)

[Appendix A — A Note on Repetitive Patterns vs. Recursive Patterns 12](#_Toc335636576)

*Table of Figures*

**No table of figures entries found.**

*Table of Tables*

**No table of figures entries found.**

# Introduction

QuickRDA 4.0 provides a new pattern language capability, which supports more powerful queries than available with previous versions.

Patterns are defined and loaded from text files. Patterns can be used from build table commands to query the graph and focus diagrams and other generation. QuickRDA 4.0 provides a default set of patterns, which are located in QuickRDA.txt in the QuickRDA home folder. Additional pattern definitions defined in other files can be loaded via build table commands.

# The QuickRDA 4.0 Build Table Commands for Pattern Language

QuickRDA 4.0 introduces two new build table commands as follows:

## PatFile

PatFile is the build command to load additional pattern definitions from text files. Once loaded, the file’s pattern definitions are available for use in subsequent build table commands. PatFile takes one argument, which is the simple name of the file (without file extension). The files can be located in the same folder as the Excel workbook holding the build table (searched first), or, in the QuickRDA home folder where the Bits are located (searched second). The argument is given as a file name without file extension. For example the following will load patterns from the file “My Patterns.txt”

/PatFile=My Patterns

## Apply

Apply is used to access named patterns (which come from pattern files). Apply takes one or more arguments. The first argument is the name of the pattern to use. Subsequent arguments provide initial values for variables in the patterns. For example, the following access the pattern Role2Role2 and applies it to the current graph.

/Apply=Role2Role2,Manager,Employee

Since the Role2Role2 pattern searches for certain well-defined relationships between two roles, it takes two parameters. In the above example, the two parameters are the “Manager” role and the “Employee” role; application to this pattern will reveal the consumes-and-provides relationships (as defined by the Role2Role2 pattern) between the Manager and Employee role. Fewer arguments may also be supplied, as in the following example.

/Apply=Role2Role2,Manager

The above sample provides the pattern with only one argument: the effect is to leave the second argument as a wild card. This example will result in revealing the consumes-and-provides relationships between the Manager and any other role. Similarly, omission of all arguments has expected effect of wild carding them all:

/Apply=Role2Role2

The above sample provides no arguments, wild carding all of them.

It is also possible to omit the first argument, and provide a value for the second, as in the following:

/Apply=Role2Role2,,Employee

Since the Role2Role2 pattern is defined symmetrically, this will yield the same results as the following:

/Apply=Role2Role2,Employee

Note that as with previous filtering in QuickRDA, Apply patterns reveals information in the graph, and thus in order for Apply to show noticeable results, source tabs should be “invisibly” included (using i) instead of “visibly” included (using x).

Also note that as with previous filtering, more than one apply pattern can be used in a single build column.

# The QuickRDA 4.0 Default Patterns

Role2Role — searches for direct linkage between two roles, such linkage is constituted as one role providing an artifact via a responsibility that the other role consumes via a responsibility.

Role2Role2 — same as above, with the addition of the indirection that an artifact is provided and a second artifact is consumed, and there is a defined relationship between the two artifacts.

Role2Role3 — changes Role2Role2 such that the first Role is revealed with responsibilities providing and consuming artifacts, whereas the second role is revealed without responsibilities.

# The QuickRDA 4.0 Pattern Language

The pattern language supports matches that can be combined via operators (and, or), with the use of parenthesis for achieving explicit operator association and precedence.

Recursion among patterns is currently not supported, thus, only (lexically) already-defined patterns may be used in subsequent pattern definitions (such usage called sub patterns). However, there is a repetition capability, which allows a sub pattern to be multiply matched provided the matches form a chain over the first two parameters.

The following is an example taken from the built-in pattern definitions:

define Art2Art <?A1, ?A2> =

reveal where

( ?A1 ? ?A2 | ?A2 ? ?A1 ) &

?A1 "Is An Instance Of" "Artifact" &

?A2 "Is An Instance Of" "Artifact"

.

define Role2Role2 <?Role1, ?Role2> =

reveal where

?Resp1 "Is Assigned To" ?Role1 &

(

(

( ?Resp1 "Consumes" ?ArtA | ?Resp1 "Consumes (Provides Input)" ?ArtA ) &

( ?Resp2 "Provides" ?ArtA2 |?Resp2 "Provides (Consumes Input)" ?ArtA2 )

) |

(

( ?Resp1 "Provides" ?ArtA | ?Resp1 "Provides (Consumes Input)" ?ArtA ) &

( ?Resp2 "Consumes" ?ArtA2 | ?Resp2 "Consumes (Provides Input)" ?ArtA2 )

)

) &

[] Art2Art <?ArtA, ?ArtA2> &

?Resp2 "Is Assigned To" ?Role2

.

The above pattern matches relationships in which one role provides an artifact via a responsibility to a potentially other consuming responsibility of a potentially other role (and vice versa). It uses the pattern match repetition capability to specify that either ?ArtA should be one and the same as ?ArtA2 (with zero repetitions of the Art2Art sub pattern) or, the Art2Art sub pattern should be repeated one or more times, forming a chain connecting ?ArtA eventually to ?ArtA2 by having, ?A2 in one repetition of Art2Art connecting to ?A1 in another repetition of Art2Art, and so on.

Regarding named patterns, there is also a restriction that all statement-match-expressions in the pattern must interconnect via variables; in some sense the pattern itself (not the eventual match) must form a single network or graph. For example, the following simple pattern is illegal because it forms two networks that are not interconnected via variables:

define illegal < ?1, ?2 > =

reveal where

?1 “Provides” ?3 &

?2 “Consumes” ?4 .

This pattern, illegal, is effectively specifying two unrelated match expressions, and could be considered somewhat nonsensical as a single pattern, though might make sense taken as a union of two completely independent patterns. Were such desired, however, it can be achieved by separating each of the independent statement match expressions into their own different pattern; all of them can be applied using consecutive build table commands.

## Pattern Language BNF

The pattern language is defined in BNF-like syntax as follows:

Pattern-file ::= Pattern \*

Pattern ::=

“define” Pattern-Name “<” Formal-Arg [ “,” Formal-Arg ]\* “>” “=”

“reveal” “where”

Match-Expression

“.”

Pattern-Name ::= identifier-or-number

Match-Expression ::=

Statement-Match-Expression |

Repetition-Application-Expression |

Match-Expression Operator MatchExpression

“(” Match-Expression “)”

Statement-Match-Expression ::=

Var-or-Const Var-or-Const Var-or-Const |

Variable Var-or-Const Var-or-Const Var-or-Const

Variable ::=

Formal-Arg |

Local-Variable

Var-or-Const ::=

Variable |

Constant-Value

Formal-Arg ::= “?”identifier-or-number

Local-Variable ::= “?”identifier-or-number

Constant-Value ::= Double-Quoted-String

Operator ::= “&” | “|”

Repetition-Application-Expresssion ::=

Repetition-Spec Pattern-Name “<” Variable [ “,” Variable ]\* “>”

Repetition-Spec ::= “[]” (see note)

*Note:* [] *is currently supported, is the same as* [0:\*] *meaning zero or more, others are planned, namely*

[0:1] *— meaning optional*

[1:1] *— meaning exactly once (i.e. simple invocation), and,*

[1:\*] *— meaning at least once*

# Appendix A — A Note on Repetitive Patterns vs. Recursive Patterns

To get a better understanding of the repetitive capability, we re-examine the repetitive pattern shown in Role2Role:

[0:\*] Art2Art <?ArtA, ?ArtA2>

This particular use of the Art2Art sub pattern allows multiple copies of the sub pattern to match in a chain linking ?ArtA and ?ArtA2.

Some of the matching capabilities of the repeating patterns are hard-coded. To get more depth, we’ll show a roughly equivalent pattern in pseudo-code with some additional constructs to work alongside the original patterns, and another pattern, Art2Art’, that is written in the recursive style, making use of the original Art2Art as well as itself.

( ?ArtA = ?ArtA2 |

Art2Art’ <?ArtA, ?ArtA2> )

Where Art2Art’ is defined as follows:

define Art2Art’ < ?A1, ?A2 > =

reveal where

Art2Art < ?A1, ?A2 > |

( Art2Art’ < ?A1, ?A3 > & Art2Art’ < ?A3, ?A2 > )

This pseudo code is more expressive, i.e. making explicit what is hard-coded in the repetitive capability.

First, note that the repetitive pattern allows a simple form where ?ArtA and ?ArtA2 are the same artifact (also note that this equality operator is pseudo code rather than supported expression). This equivalence test is an important part of the built-in repetitive pattern match mechanism.

Second, note the introduction of the recursive composition involving two Art2Art matches connected by a third variable ?A3. This is also an important part of the built-in repetitive pattern match mechanism.

While the recursive mechanisms shown above would be more expressive in allowing other matches beyond the one stated, it would also suffers from recursive termination problems: namely, that this form — as is — will match unintended matches (those having loops in the graph of artifacts, which are unintended matches), and also effectively infinitely loop during matching.

What we’d need to express beyond the simple recursive match is a restriction preventing unintended matches, for example, that a new ?A3 does not match any other ?A1 or ?A2 in the recursion. One way this could be expressed (again in the pseudo code) is to add some values and testing of them to the recursion as follows:

define Art2Art’ < ?A1, ?A2, {Values} > =

reveal where

Art2Art < ?A1, ?A2 > |

( Art2Art’ < ?A1, ?A3, {Values}+{?A3} > &

Art2Art’ < ?A3, ?A2, {Values}+{?A3} > &

{?A3} is not a subset of {Values} )

This would be initially invoked using:

( ?ArtA = ?ArtA2 |

Art2Art’ <?ArtA, ?ArtA2, {?ArtA}+{?ArtA2} > )

(Note that a similar restriction could be applied to statement matches instead of variable matches.)

This illustrates some of the additional complexity required to support recursive declarative pattern matching, and, is one of the reasons that at present, QuickRDA 4.0 supports a specific subcase (using the notion of repetitive matches) of what could be expressible with recursion.