NAME

Lintian::Relation – Lintian operations on dependencies and relationships

SYNOPSIS

```
my $depends = Lintian::Relation->new('foo | bar, baz');
print "yes\n" if $depends->implies('baz');
print "no\n" if $depends->implies('foo');
```

DESCRIPTION

This module provides functions for parsing and evaluating package relationship fields such as Depends and Recommends for binary packages and Build-Depends for source packages. It parses a relationship into an internal format and can then answer questions such as ''does this dependency require that a given package be installed'' or ''is this relationship a superset of another relationship.''

A dependency line is viewed as a predicate formula. The comma separator means ''and'', and the alternatives separator means ''or''. A bare package name is the predicate ''a package of this name is available''. A package name with a version clause is the predicate ''a package of this name that satisfies this version clause is available.'' Architecture restrictions, as specified in Policy for build dependencies, are supported and also checked in the implication logic unless the **new_noarch()** constructor is used. With that constructor, architecture restrictions are ignored.

CLASS METHODS

new(RELATION)

Creates a new Lintian::Relation object corresponding to the parsed relationship RELATION. This object can then be used to ask questions about that relationship. RELATION may be `undef` or the empty string, in which case the returned Lintian::Relation object is empty (always satisfied).

parse_element

new_norestriction(RELATION)

Creates a new Lintian::Relation object corresponding to the parsed relationship RELATION, ignoring architecture restrictions and restriction lists. This should be used in cases where we only care if a dependency is present in some cases and we don't want to require that the architectures match (such as when checking for proper build dependencies, since if there are architecture constraints the maintainer is doing something beyond Lintian's ability to analyze) or that the restrictions list match (Lintian can't handle dependency implications with build profiles yet). RELATION may be `undef` or the empty string, in which case the returned Lintian::Relation object is empty (always satisfied).

new_noarch(RELATION)

An alias for new_norestriction.

and(RELATION, ...)

Creates a new Lintian::Relation object produced by AND'ing all the relations together. Semantically it is the similar to:

```
Lintian::Relation->new (join (', ', @relations))
```

Except it can avoid some overhead and it works if some of the elements are Lintian::Relation objects already.

INSTANCE METHODS

duplicates()

Returns a list of duplicated elements within the relation object. Each element of the returned list will be a reference to an anonymous array holding a set of relations considered duplicates of each other. Two relations are considered duplicates if one implies the other, meaning that if one relationship is satisfied, the other is necessarily satisfied. This relationship does not have to be commutative: the opposite implication may not hold.

restriction_less()

Returns a restriction-less variant of this relation (or this relation object if it has no restrictions).

implies(RELATION)
: Returns true if the relationship implies RELATION, meaning that if the Lintian::Relation object is satisfied, RELATION will always be satisfied. RELATION may be either a string or another Lintian::Relation object.

  By default, architecture restrictions are honored in RELATION if it is a string. If architecture restrictions should be ignored in RELATION, create a Lintian::Relation object with **new_noarch()** and pass that in as RELATION instead of the string.

implies_element
implies_array
implies_inverse(RELATION)
: Returns true if the relationship implies that RELATION is certainly false, meaning that if the Lintian::Relation object is satisfied, RELATION cannot be satisfied. RELATION may be either a string or another Lintian::Relation object.

  As with **implies()**, by default, architecture restrictions are honored in RELATION if it is a string. If architecture restrictions should be ignored in RELATION, create a Lintian::Relation object with **new_noarch()** and pass that in as RELATION instead of the string.

implies_element_inverse
implies_array_inverse
unparse()
: Returns the textual form of a relationship. This converts the internal form back into the textual representation and returns that, not the original argument, so the spacing is standardized. Returns undef on internal failures (such as an object in an unexpected format).

matches (REGEX[, WHAT])
: Check if one of the predicates in this relation matches REGEX. WHAT determines what is tested against REGEX and if not given, defaults to VISIT_PRED_NAME.

  This method will return a truth value if REGEX matches at least one predicate or clause (as defined by the WHAT parameter – see below).

  NOTE: Often ''implies'' (or ''implies_inverse'') is a better choice than this method. This method should generally only be used when checking for a ''pattern'' package (e.g. phpapi-[\d\w+]+).

  WHAT can be one of:

  VISIT_PRED_NAME
  : Match REGEX against the package name in each predicate (i.e. version and architecture constrains are ignored). Each predicate is tested in isolation. As an example:

```
my $rel = Lintian::Relation->new ('somepkg | pkg-0 (>= 1)');
# Will match (version is ignored)
$rel->matches (qr/^pkg-\d$/, VISIT_PRED_NAME);
```

  VISIT_PRED_FULL
  : Match REGEX against the full (normalized) predicate (i.e. including version and architecture). Each predicate is tested in isolation. As an example:

```
my $vrel = Lintian::Relation->new ('somepkg | pkg-0 (>= 1)');
my $uvrel = Lintian::Relation->new ('somepkg | pkg-0');

# Will NOT match (does not match with version)
$vrel->matches (qr/^pkg-\d$/, VISIT_PRED_FULL);
# Will match (this relation does not have a version)
$uvrel->matches (qr/^pkg-\d$/, VISIT_PRED_FULL);

# Will match (but only because there is a version)
$vrel->matches (qr/^pkg-\d \(.*\)$/, VISIT_PRED_FULL);
```

```
# Will NOT match (there is no version in the relation)
$uvrel->matches (qr/^pkg-\d  \(.*\)$/, VISIT_PRED_FULL);
```

VISIT_OR_CLAUSE_FULL
: Match REGEX against the full (normalized) OR clause. Each predicate will have both version and architecture constrains present. As an example:

```
my $vpred = Lintian::Relation->new ('pkg-0 (>= 1)');
my $orrel = Lintian::Relation->new ('somepkg | pkg-0 (>= 1)');
my $rorrel = Lintian::Relation->new ('pkg-0 (>= 1) | somepkg');

# Will match
$vrel->matches (qr/^pkg-\d(?: \([^\)]\))?$/, VISIT_OR_CLAUSE_FULL);
# These Will NOT match (does not match the "|" and the "somepkg" part)
$orrel->matches (qr/^pkg-\d(?: \([^\)]\))?$/, VISIT_OR_CLAUSE_FULL);
$rorrel->matches (qr/^pkg-\d(?: \([^\)]\))?$/, VISIT_OR_CLAUSE_FULL);
```

visit (CODE[, FLAGS])
: Visit clauses or predicates of this relation. Each clause or predicate is passed to CODE as first argument and will be available as `$_`.

  The optional bitmask parameter, FLAGS, can be used to control what is visited and such. If FLAGS is not given, it defaults to VISIT_PRED_NAME. The possible values of FLAGS are:

  VISIT_PRED_NAME
  : The package name in each predicate is visited, but the version and architecture part(s) are left out (if any).

  VISIT_PRED_FULL
  : The full predicates are visited in turn. The predicate will be normalized (by ''unparse'').

  VISIT_OR_CLAUSE_FULL
  : CODE will be passed the full OR clauses of this relation. The clauses will be normalized (by ''unparse'')

    Note: It will not visit the underlying predicates in the clause.

  VISIT_STOP_FIRST_MATCH
  : Stop the visits the first time CODE returns a truth value. This is similar to first, except visit will return the value returned by CODE.

  Except where a given flag specifies otherwise, the return value of visit is last value returned by CODE (or `undef` for the empty relation).

empty
: Returns a truth value if this relation is empty (i.e. it contains no predicates).

unparsable_predicates
: Returns a list of predicates that were unparsable.

  They are returned in the original textual representation and are also sorted by said representation.

AUTHOR

Originally written by Russ Allbery <rra@debian.org> for Lintian.

SEE ALSO

lintian (1)