ReDIF specification file documentation

Thomas Krichel and Ivan Kurmanov

December 1999

1 Introduction

This document is available on the web at `http://openlib.org/acmes/root/docu/spefor.html`.
It gives details of how the ReDIF format is being encoded in a special specification file. This file is normally called "redif.spec". This documentation explains how to make changes to a ReDIF specification. One can make enhancements to an existing specification or design a new specification from scratch.
Before trying to understand the specification of ReDIF, one should understand the ReDIF data format documentation well, because this document is like a dark side of the moon for ReDIF:it is better look at the bright side first.
The ReDIF specification file contains a ReDIF specification in a custom-built format called the "SPEcification FOrmat for Redif", abbreviated spefor. The spefor file is loaded each time any data reading starts. It defines:

- template types and cluster types
- attributes (fields) names and other properties in each template type or cluster type
- attribute data types: checking and preprocessing

Each cluster and template has a finite set of allowed attributes, some of which may be required to appear in every template of the type. An attribute in a template or a cluster must have a defined (complete) name and may have some additional properties set. The additional properties of an attribute are: type of value, the permission to omit and the prohibition to repeat the attribute in a template.

2 Syntax

2.1 General aspects of the syntax

The spefor syntax borrows some features of Perl code syntax. Comment lines start with the hash char # symbol. Repetitions of whitespace (i.e. blanks and tabulation characters) are ignored.
Spefor files make extensive use of blocks of text. Blocks start with the open curly brace `{` character and end with the close curly brace `}` character. Unlike in C or Perl, each block closes with a line consisting of a single `}` character, not taking spaces and tab chars into account.
This means that in the following example:

```
A {
```

block's code here

```
}
 B {
```

other data or code `}`
block `A` is correctly closed, but the block `B` is not. The difference is a newline character before closing brace `}`. This imposes a limitation on the contents of each block: you can not put a line with single `}` inside a block. Be careful with Perl code in blocks.
The ReDIF specification is specified by a number of definitions. In other words, definitions are the main structural elements of the spefor file.

2.2 Definition syntax

A definition states what is being defined and how it will be named and then gives the contents of the definition in a block. The most common syntax is
def_type = *name block* '\n'
'\n' designates a newline character here. In some cases, you will also see definitions of the syntax
def_type = *name*[/ *parameter*] *block* '\n'
The item in square brackets is optional here.
There are three types of definitions. Therefore *def_type* variable can take only three values. These are `template`, `cluster` and `type`. These values are case-insensitive.

2.3 Template type definition

A template type definition lists attributes of a template and sets their properties. The *name* variable of the definition line gives the template type name in full, like `ReDIF-Paper 1.0` for example. All its attributes are listed inside the definition *block*. Each attribute is listed on a separate line. Each non-empty line defines one attribute of the template. Its syntax is
attribute_name[:[*type*][:[*subtype*][:[*required_flag*][:[*non_repeatable_flag*]]]]]
Whitespace characters are not allowed inside this string. They are allowed at the beginning or end of the string. The colon character is a delimiter here. The only part of the line which is required is the *attribute_name*. *type*
and *subtype* have several different uses for different cases, but in general they specify content properties of the attribute. In most cases only *type*
is given, the *subtype* parameter has less usage.
The meaning of the *required_flag* is to answer the following question: Is this attribute necessary to consider template or cluster valid or it is optional? The value `1` in the *required_flag*
means that the attribute is required.
The meaning of *non_repeatable_flag* is to answer the following question: Can this attribute appear several times in one template or cluster? By default attributes can be repeated, but if you set *non_repeatable_flag* to `1`, then the attribute will become non-repeatable.
A special format exists for key attributes of template and cluster types. If in the *required_flag* field of the attribute definition there is an asterisk '*' (instead of `1` or anything else), then this attribute will become the key attribute of the template or the cluster.
In all other cases *required_flag* and *non_repeatable_flag* may contain any digits, but normal value for it is 1 or nothing by default.
An example of a valid template type definition:

```
template = ReDIF-Person 1.0 {
Template-Type:::*
        Handle:::1:1
        Name-Full:::1:1
        Name-First
        Name-Last
        WorkPlace:cluster:Organization::
        Email
        Fax
        Postal
        Phone
        Homepage:URL
      # Homepage-Publications:URL
      # Classification-Jel
}
```

This example defines the template type `ReDIF-Person 1.0`. The attribute `Template-type` is the key attribute of the template, the `Handle` and `Name-Full` attributes are required and non-repeatable. The attribute `Workplace` is of type `cluster` and subtype `Organization`. The attribute `Homepage` is of type `URL`. Lines with `Homepage-Publications` and `Classification-Jel` attributes are commented out. They are not included into the template

type.

2.4 Cluster type definition

Clusters are groups of attributes. They are defined outside the template types and may be used as a group in various templates to build composite elements. For example consider the following cluster definition

```
cluster = address {
 Country:::*
 state:
 county
 zipcode
 city:::1:1
 street
}
```

Similarly to template type definitions, a cluster type definition contains a list of attribute names with their properties. This particular example (above) defines a cluster type named `address`.
Elements of a cluster type (attributes) can not be used directly in any template. To use a cluster in some template type, we need to introduce it there through an ordinary attribute. The attributes of the cluster then can appear in the data being prefixed by that attribute and a dash.
For example, this means that if we introduce a cluster of type `address` into some template through an attribute `Home` then following data lines may be valid inside such templates:

```
Home-Country: U.S.A.
Home-State: CA
Home-City: Santa Barbara
```

More detailed explanation about how to introduce a cluster into a template will be found in one of the following sections.
Once defined a cluster type can be used in any template or even cluster type. The scope of the cluster type definition's is global and clusters can be nested.

3 Attribute types

When we list attributes in a template or a cluster type definition we can specify each attribute as having a certain *`type`*. There is one special type of attributes which will be described right in the next section. It is the type `cluster`.

3.1 Attribute type `cluster`

If the *`type`* property of an attribute line takes value `cluster`, then the *`subtype`* shall point to a specific cluster type. For example if we have defined cluster type `course`, then in a template type definition for a teacher we can put following attribute line:

```
teaching:cluster:course
```

In this case the attribute `teaching` in the template type will represent a cluster of type `course`. Attributes of this cluster type may appear in any template of this type, being prefixed by `teaching-`. For example, look at a template

```
Template-Type: ReDIF-Teacher 1.0
Name: Thomas Krichel
Teaching-Course-Name: Dynamic Macroeconomics
Teaching-Course-Class-Size: 25
Teaching-Course-Name: Open Economy Macroeconomics
Teaching-Course-Class-Size: 20
Handle: RePEc:per:teachr:T_Krichel
```

Such a template is possible if the cluster type `course` has been defined as

```
cluster = course {
   course-name:::*
   course-class-size
}
```

Exactly the same mechanism can be used to include clusters into other clusters, i.e. clusters can be nested.

4 User-defined value data types

All we dealt with up to this moment are the ReDIF template structures. Having defined a template structure is important for parsing and validating data, but sometimes more deep checking of data is desired. And this is what this section is about.
We have shown how an attribute specified in a template type or a cluster type definition can bring a cluster into a corresponding structure. We use a special value of `cluster` of the `type` property of an attribute for that.
In other cases, the type of an attribute is used to specify the attribute's value data type. These types are types of value that this attribute may take. In most cases this property is empty which means that no certain data type has been assigned, and any value is possible.
For example, we can define a data type `date` for our date-related attributes and then declare those attributes as having value type `date`. This will impose some checking and processing rules of the `date` data type on all appearances of the attributes or, to be more correct, on the values of the attributes. To set those checking and processing rules we use the Perl programming language. We will be more elaborate on that subject in the following sections.

4.1 Defining data value types

Value types define rules by which the attribute values will be processed when a ReDIF data file is being read or checked. Attributes may be associated with a value data type through a cluster or template type definition, as it has been shown in the previous sections.
The value data type definition syntax is
`type =` *`type_name`*`[/` *`type_aspect`*`]` *`block`*`'\n'`
The *`type_aspect`* specifies how to interpret the definition's content block. This parameter may take three different values

check-regex a regular expression for checking value; this is the default.

check-eval a piece of Perl code for checking value

preproc-eval a piece of Perl code for preprocessing value

4.2 Type aspects: check-regex

For example, let us look at a following definition:

```
type = handle / check-regex {
     ^RePEc\:[a-zA-Z]{3}(\:[a-zA-Z0-9]{6}(\:[^\s\n]+)?)?$
}
```

It defines a `check-regex` aspect of data type `handle`. As a result, all values of `handle` type attributes will be checked to match the Perl regular expression in the braces above.
Thus the `check-regex` aspect allows to set the syntax of attribute values in the form of a Perl regular expression. Values that don't match the Perl regular expression would be rejected and the corresponding template will be invalid.

4.3 Type aspects: check-eval

The `check-eval` aspect allows to set checking routine code for values of the data type in Perl language. The code is evaluated at run-time. Depending on the return value (or last expression, evaluated in the block), the attribute is accepted or rejected.
In the code one has access to the value which we need to be check by a scalar variable *`$value`*. Let's look at an example

```
type = pubstat / check-eval  {
   if (   $value !~ /^Forthcoming/i or
          $value !~ /^Published/i ) {
            return 0;
   } else { return 1; }
}
```

This type is for the Publication-Status attribute, whose value must start with one of `Forthcoming` and `Published` case-insensitive.
Here again let us remind that one cannot use single closing brace `}` on a line inside a block. So if we write the same piece in a different way

```
type = pubstat / check-eval     {
   if (   $value  !~ /^Forthcoming/i
       or $value !~ /^Published/i ) {
             return 0;
   } else {
             return 1;
   }
}
```

this will cause an error at spefor file (redif.spec) reading.
There is another one interesting feature in *`check-eval`* aspect of types. An author or maintainer of data often wants to know why this or that template is rejected. The *rech* data checking script for this purpose uses so-called messages.
A message has a text (content) and a status. A status is one of several levels: from a debugging note (lowest) to a fatal error message (highest). But the two most often used levels are: a warning and an error. For further technical details please consult *rech*, *rr.pm* and *messages.pm* Perl files.
If during reading a template somewhere appears a message with status of error or higher, then the template will be rejected. Depending on the invocation of *rech*, the message content may be printed to the standard output.
Should a type checking aspect *check-regex* or *check-eval* executed for a particular value produce a negative result, the parsing software will generate a message of a general form, stating that `value XXX of type YYY is invalid` or alike.
If one wants to make for clever and friendly checking for your data, he may want to change this kind of message to a different one, showing exactly what is the problem of the data. To make this in *check-eval* aspect definition you may yourself initiate a message. To initiate a message you simply call a function *msg*. But to exclude error duplication, if you initiate an error message in a *check-eval* block, be sure to return a positive value at the end.
The *msg* subroutine accepts the message text as the first parameter, and a number - the status level as the second parameter. The status level of 2 corresponds to a warning, 3 - to an error.
For example, the code in the following type definition may generate an error message when the corresponding value is incorrect.

```
 type = integer / check-eval {

     if ($value =~ /^\d+$/) {
msg "($file, $line): Value <$value> is not an integer", 3;
return 1;    # as if it was OK
     ;}
}
```

Thus function *msg* sends a message. The *$file* and *$line* variables will be equal to the data filename and the line number position in the file.
Apart from figuring out and reporting errors, you can also give user a warning about data. Let's look into another example

```
type = URL / check-eval {
   if (     $value =~ /^(ftp|http):\/\//i
        and $value !~ /^(ftp|http):\/\// ) {
          msg "($file, $line): URL capitalization is bad <$value>", 2;
```

```
          $value =~ s/^Ftp/ftp/i;
          $value =~ s/^Http/http/i;
   ;}

   $value =~ /^(URL:)?
      (ftp|http|gopher)\:
      \/\/[a-zA-Z0-9\_\.\-]+
      (\:[0-9]+)?
      (\/?|[\~\&\%\^\@\!\'\"\'\(\)\|\#\:\;\/A-Za-z\-\_0-9\?\=\*\\\+\,\.]+)$/x;
}
```

In the first part of this *check-eval* we check the capitalization of the starting part of a URL (protocol, like http or ftp). The line

```
msg "($file, $line): URL capitalization is bad \"$value\"", 2;
```

creates a message with status 2 (Warning). The template will not be ignored. Then we correct the capitalization of the value and go further. In the finishing part of the block we check the value to match a big regular expression (Perl5 extended syntax - note "/x" in the end). The logical value calculated in the result of this matching operator will trigger an error if the value does not match.

4.4 Type aspects: preproc

Another useful function that a ReDIF parsing software (which is behind *rech*, *rere* or *rr.pm*) may perform is in to convert attribute values from the form in which the author has written it to some other (possibly, standardized) form.
Historically within the RePEc project, a need for this feature arose with the use of the subject classification codes defined by the Journal of Economic Literature, the JEL codes. For example, we could do (and actually did) so, that a sample value of JEL classification attribute `a21; b3, X3` would be automagically converted to `A21 B3` at the data reading stage, 'on the fly'. This conversion includes: change to the uppercase letters, separation by single space characters, and ignoring (filtering out) the invalid code `X3`.
It is implemented using a `preproc` value data type aspect. It works very much like `check-eval`. The `preproc` type block is `eval`-uated in Perl while reading data, but the resulting (returned) value does not matter.
Consider another following example:

```
type = date / preproc {
   return if $value eq '';
  if ($value !~ /-/ ) {
      $value =~ s/^(\d{4})(\d{2})$/$1-$2/;
      $value =~ s/^(\d{4})(\d{2})(\d{2})/$1-$2-$3/; }
  if ($value !~ /^\d{4}(-\d\d){0,2}$/ ) {
      if ($rr::Options{'MessageOut'})  {
          include_attrline( );
          msg "($file, $line): A bad date value format: \"$value\"", 2;
      } else { $value = '' ; }
      return 1;
  ;}
}
```

This example is for attributes of the *date* type. Dates are converted to a standard form or cleared to `''`, if original value is of incorrect format. But if this code is executed in the data checking mode (for example, in *rech* script) which is checked by `$rr::Options{'MessageOut'}` then a warning message is generated. Also note a function call before creating a message:
`include_attrline( );`
This is necessary for correct rech output if you generate messages in `preproc`.

5 Checking sequence

Just to clear some possible questions, here we have a brief introduction about how the ReDIF data is being parsed and how the spefor file is being used.
When a ReDIF parsing engine behind *rr.pm*, *rere* and *rech* reads an attribute, it checks some control values i.e. it asks several questions about an attribute. If any of a question leads to a negative result, an error is being created. And such an error causes the whole template to be treated as invalid. The template is ignored and the processing program (the parser's client) cannot see it.
The first thing checked about attribute is whether the attribute is correctly placed. Is it allowed for the current template and/or cluster type(s)? If the attribute is repeated, is it allowed to be repeated here?
If the attribute fits well into the template structure (does not violate the template type), the parser then checks the value of the attribute.
The engine then checks what type of attribute it is (as it is defined in the template or cluster type). If its type is `cluster`, then the engine *opens* a new cluster. If it is of some data type, that type's `preproc` aspect is checked and executed (`eval`-uated), if it has been defined in spefor. Then `check-regex` is matched if defined. Next step is `check-eval` `eval`-uation, you guessed, if defined.
If at the `check-regex` or the `check-eval` stage a negative result has been received, then the engine generates an error and stops further processing of the template. Actually the same happens if an error appears in the `pre-proc` aspect.
When a new `Template-type` attribute is encountered or an end of file is reached, the template is checked to contain all *required* attributes of the template type. Processing the template goes on successfully only if this check returns positive result. After that the template is considered to be complete and valid.