rnginline Documentation

Release 0.0.0

Hal Blackburn

March 29, 2015

Contents

rnginline is a Python library and command line tool for flattening multi-file RELAX NG schemas into a single file, taking care not to change the semantics of the schema.

It works by implementing just enough of the RELAX NG simplification rules to replace `<include href="...">` / `<externalRef href="...">` elements with the content of the external files they reference.

It can be used:

- As part of a build workflow to merge RELAX NG schemas ahead of time
- At runtime as a Python library to load multi-file schemas stored as data files in Python packages (lxml doesn't support loading multi-file schemas from anything other than the filesystem.)

CHAPTER 1

Contents

1.1 Quickstart

Install with pip:

```
$ pip install rnginline
```

You can use it from Python like this:

```
>>> import rnginline
>>> rnginline.inline('my-nested-schema-root.rng')
<lxml.etree.RelaxNG object at ...>
```

You can load a multi-file schema from a Python package's data like this:

```
>>> import rnginline
>>> from rnginline.urlhandlers import pydata
>>> url = pydata.makeurl('rnginline.test',
...                      'data/testcases/external-ref-1/schema.rng')
>>> url
'pydata://rnginline.test/data/testcases/external-ref-1/schema.rng'
>>> rnginline.inline(url)
<lxml.etree.RelaxNG object at ...>
```

You can use it from the command line like this:

```
$ rnginline my-nested-schema-root.rng flattened-output.rng
```

See Using rnginline From the Command Line or Using rnginline From Python for more details.

1.2 Installation

1.2.1 Python & OS Support

rnginline supports Python 2.6, 2.7, 3.3 and 3.4 and PyPy (but not PyPy3, as it doesn't yet support lxml).

rnginline works on Unix/Linux, OS X and Windows.

1.2.2 Install rnginline

Install via pip:

```
$ pip install rnginline
```

1.3 Using rnginline From the Command Line

Firstly, ensure you've installed rnginline.

1.3.1 Basic Usage

Basic usage is:

```
$ rnginline my-nested-schema-root.rng flattened-output.rng
```

If the second argument is not given, output goes to stdout, so you can pipe it through `xmllint --format` or similar:

```
$ rnginline schema.rng | xmllint --format -
<grammar xmlns="http://...
```

`rnginline -h` gives help output as you'd expect, listing all available options:

```
$ rnginline -h
Flatten a hierachy of RELAX NG schemas into a single schema by recursively
inlining <include>/<externalRef> elements.

usage: rnginline [options] <rng-src> [<rng-output>]
       rnginline [options] --stdin [<rng-output>]

[...]
```

1.3.2 Full Example

In this example we create `schema.rng`, which references `external.rng` and inline them into one file, tidying the output with `xmllint`:

```
$ cat > schema.rng <<'EOF'
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="external.rng">
    <!-- Override foo -->
    <define name="foo">
      <element name="foo">
        <value>abc</value>
      </element>
    </define>
  </include>
  <start>
    <element name="root">
      <ref name="foo"/>
      <ref name="bar"/>
    </element>
  </start>
```

```
</grammar>
EOF

$ cat > external.rng <<'EOF'
<rng:grammar xmlns:xyz="x:/my/ns" xmlns:rng="http://relaxng.org/ns/structure/1.0">
  <rng:define name="foo">
    <rng:element name="foo">
      <rng:notAllowed/> <!-- No foo for you! -->
    </rng:element>
  </rng:define>
  <rng:define name="bar">
    <rng:element name="xyz:bar">
      <rng:text/>
    </rng:element>
  </rng:define>
</rng:grammar>
EOF
```

Flatten `schema.rng` and everything it includes into a single XML document, re-indent it with `xmllint` and save the output in `out.rng`:

```
$ rnginline schema.rng | xmllint --format - > out.rng

$ cat out.rng
<?xml version="1.0"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <div>
    <rng:div xmlns:xyz="x:/my/ns" xmlns:rng="http://relaxng.org/ns/structure/1.0">
      <rng:define name="bar">
        <rng:element name="xyz:bar">
          <rng:text/>
        </rng:element>
      </rng:define>
    </rng:div>
    <!-- Override foo -->
    <define name="foo">
      <element name="foo">
        <value datatypeLibrary="">abc</value>
      </element>
    </define>
  </div>
  <start>
    <element name="root">
      <ref name="foo"/>
      <ref name="bar"/>
    </element>
  </start>
</grammar>
```

The resulting schema acts as expected — the `foo` definition from `external.rng` has been overridden:

```
$ xmllint --relaxng out.rng - <<'EOF'
<root>
    <foo>abc</foo>
    <bar xmlns="x:/my/ns">123</bar>
</root>
EOF
<?xml version="1.0"?>
<root>
```

```
    <foo>abc</foo>
    <bar xmlns="x:/my/ns">123</bar>
</root>
- validates
```

1.3.3 Advanced Usage

This section describes some less common use cases.

Passing the input on stdin

You can send the root schema on stdin, but doing so means rnginline won't know the location of the file, which means it can't resolve relative references in the input without extra information. To prevent casual use of stdin without realising the issues, the `--stdin` option must be passed in place of the input file.

To tell rnginline where the schema on stdin is from, use the `--base-uri` option. If you don't specify a base, the paths of included files will be relative to the current directory.

Here's a (contrived) example of pre-processing the input before passing it on stdin:

```
$ xmllint --format - < /tmp/schema.rnc | rnginline --base-uri /tmp/schema.rnc --stdin
<grammar xmlns="http://...
```

1.4 Using rnginline From Python

Firstly, ensure you've installed rnginline.

1.4.1 Basic Usage

The example files used here are shown at the *bottom of this section*.

RELAX NG files on the filesystem can be referenced by path:

```
>>> import rnginline, os
>>> sorted(os.listdir('.'))
['external.rng', 'schema.rng']
>>> rnginline.inline('schema.rng')
<lxml.etree.RelaxNG object at ...>
```

But `lxml` can already do that; the real utility is in loading multi-part schemas from locations other than the filesystem, which `lxml` can't do.

We can load multi-part schemas stored in Python packages (which may be stored in zip files on disk). Here's how to load one of the schemas from rnginline's test suite:

```
>>> import rnginline
>>> from rnginline.urlhandlers import pydata
>>> url = pydata.makeurl('rnginline.test',
...                      'data/testcases/external-ref-1/schema.rng')
>>> url
'pydata://rnginline.test/data/testcases/external-ref-1/schema.rng'
>>> rnginline.inline(url)
<lxml.etree.RelaxNG object at ...>
```

1.4.2 Data Sources

The first argument to `inline()` defines the location to load the top-level schema from. It can be a filesystem path, a URL, a file-like object or an `lxml.etree` element.

If you don't want `inline` to guess which input you're providing, you can pass the input as a specific type using one of the `url`, `path`, `file` or `etree` keyword args instead.

URLs

When you use a URL as the input, it's retrieved using the same machinery that fetches external schemas during the inlining process. By default two types of URLs are supported. `file:` URLs referencing the local filesystem, and `pydata:` URLs referencing data in a Python package. Note that the `pydata:` scheme is a proprietary/unregistered scheme created for use in rnginline.

Note: You can add support for URLs other than these, see *the URL Handlers section* for details.

```
>>> rnginline.inline('pydata://rnginline.test/data/testcases/include-1/schema.rng')
<lxml.etree.RelaxNG object at ...>
>>> from rnginline import urlhandlers
>>> url = urlhandlers.pydata.makeurl('rnginline.test', 'data/testcases/include-1/schema.rng')
>>> url
'pydata://rnginline.test/data/testcases/include-1/schema.rng'
>>> rnginline.inline(url)
<lxml.etree.RelaxNG object at ...>
```

Filesystem Paths

When you pass a filesystem path, it's converted into a scheme-less URL path which is resolved against the *default base URL*, which by default is the current working directory.

```
>>> os.link('schema.rng', 'Not valid URL path.rng')
>>> rnginline.inline('Not valid URL path.rng')
<lxml.etree.RelaxNG object at ...>
>>> url = urlhandlers.file.makeurl('Not valid URL path.rng')
>>> url
'Not%20valid%20URL%20path.rng'
>>> rnginline.inline(url)
<lxml.etree.RelaxNG object at ...>
>>> os.unlink('Not valid URL path.rng')
```

File-like Objects

You may pass a file-like object as the input source. URLs inside the input's schema document will be relative to the *default base URI* (current directory) unless you use the `base_uri` keyword arg to `inline()` to specify a the base URI of the file object.

```
>>> os.mkdir('foo')
>>> os.chdir('foo')
>>> with open('../schema.rng') as f:
...     # schema.rng references external.rng, which would fail to
...     # resolve unless we provide a base URI
...     rnginline.inline(f, base_uri='../schema.rng')
<lxml.etree.RelaxNG object at ...>
```

```
>>> os.chdir('..')
>>> os.rmdir('foo')
```

`lxml.etree` Element

You can pass pre-parsed XML as an `lxml.etree` element. The base URI of the elements in the document is respected, and you should most likely ensure it's defined to be something sensible to allow references to external files to be resolved correctly.

The base URI of an element is by default the URL of the location the parser read the document from. It can be overridden from within the XML document using the `xml:base` attribute as well.

```
>>> os.mkdir('foo')
>>> os.chdir('foo')

>>> from lxml import etree
>>> doc = etree.parse('../schema.rng')
>>> doc.docinfo.URL
'../schema.rng'
>>> rnginline.inline(doc)
<lxml.etree.RelaxNG object at ...>

>>> with open('../schema.rng') as f:
...     schema_content = f.read()
>>> elem = etree.fromstring(schema_content)
>>> elem.getroottree().docinfo.URL is None
True
>>> rnginline.inline(elem, base_uri='../schema.rng')
<lxml.etree.RelaxNG object at ...>

>>> elem = etree.fromstring(schema_content, base_url='../schema.rng')
>>> rnginline.inline(elem)
<lxml.etree.RelaxNG object at ...>

>>> os.chdir('..')
>>> os.rmdir('foo')
```

Note: If you use `etree.XML()`/`etree.fromstring()`, the XML won't have a base URI set unless you use the `base_url` keyword arg.

1.4.3 URL Handlers

URLs encountered in `<include href="...">` / `<externalRef href="...">` elements are fetched using the URL Handlers registered with the Inliner whose `inline()` method has been called. As mentioned above, handlers for `file:` and `pydata:` URLs are provided and activated by default.

Handlers for other URL schemes can be created and used quite easily. Say you wanted to inline a schema referencing sub parts via HTTP. You could do it like this:

```
>>> from rnginline.urlhandlers import ensure_parsed
>>> import requests  # http://python-requests.org
>>> class HTTPUrlHandler(object):
...     def can_handle(self, url):
...         print('Calling can_handle() w/ {}'.format(url))
...         return ensure_parsed(url).scheme == 'http'
```

```
...
...     def dereference(self, url):
...         print('Calling dereference() w/ {}'.format(url))
...         return requests.get(url).content

>>> from http.server import HTTPServer, SimpleHTTPRequestHandler
>>> import threading
>>> # Start an HTTP server serving the schemas in our cwd
>>> httpd = HTTPServer(('localhost', 8000), SimpleHTTPRequestHandler)
>>> threading.Thread(target=httpd.serve_forever).start()

>>> rnginline.inline('http://localhost:8000/schema.rng',
...                  handlers=[HTTPUrlHandler()])
Calling can_handle() w/ http://localhost:8000/schema.rng
Calling dereference() w/ http://localhost:8000/schema.rng
Calling can_handle() w/ http://localhost:8000/external.rng
Calling dereference() w/ http://localhost:8000/external.rng
<lxml.etree.RelaxNG object at ...>

>>> httpd.shutdown()
```

1.4.4 Example Files

The preceding examples in this section assume the following files exist in the directory the examples are run from.

Listing 1.1: schema.rng

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="external.rng">
    <!-- Override foo -->
    <define name="foo">
      <element name="foo">
        <value>abc</value>
      </element>
    </define>
  </include>
  <start>
    <element name="root">
      <ref name="foo"/>
      <ref name="bar"/>
    </element>
  </start>
</grammar>
```

1.5 URI Resolution

`rnginline` uses URI Resolution to map relative paths like `common/name.rng` to absolute URIs, such as `file:/a/b/proj/common/name.rng` which can be handled by a URL Handler.

1.5.1 Resolution 101

A quick crash course in URI resolution. Check out RFC 3986 if you want all the details.

Listing 1.2: external.rng

```
<rng:grammar xmlns:xyz="x:/my/ns" xmlns:rng="http://relaxng.org/ns/structure/1.0">
  <rng:define name="foo">
    <rng:element name="foo">
      <rng:notAllowed/> <!-- No foo for you! -->
    </rng:element>
  </rng:define>
  <rng:define name="bar">
    <rng:element name="xyz:bar">
      <rng:text/>
    </rng:element>
  </rng:define>
</rng:grammar>
```

URI Types

There are two types of URIs we need to know about. Absolute URIs and relative URIs. In simple terms, Absolute URIs have a *scheme*, which is the part of a URI before the first colon, e.g. in `http://example.org` `http` is the scheme. Relative URIs don't have a scheme. e.g. `file.txt`, `//example.org/foo` and `/tmp/file.txt` are all relative URIs.

Resolving

URI Resolution is the process of merging two URIs. A *base* URI which is always absolute, and a *reference* URI which can be absolute or relative.

There are two aspects of URI resolution we need to know about. Resolving schemes and resolving paths.

Schemes

If we resolve a relative reference URI against an absolute base URI, the resulting URI has the scheme of the base URI, with other parts overridden by the reference URI:

```
>>> from rnginline import uri
>>> uri.resolve('file:', 'somefile.txt')
'file:somefile.txt'
```

Resolving an absolute reference URI results in the base being replaced by the reference:

```
>>> uri.resolve('file:somefile.txt', 'other:blah')
'other:blah'
```

Paths

The path component of a relative reference URI is resolved against the path component of the base URI:

```
>>> uri.resolve('file:/some/dir/', 'other/dir/file.txt')
'file:/some/dir/other/dir/file.txt'
```

If the reference URI's *path* is absolute (starts with a `/`) then it replaces the base URIs path:

```
>>> uri.resolve('file:/some/dir/', '/tmp/foo.txt')
'file:/tmp/foo.txt'
```

If the reference URI is absolute, its path replaces the base URIs path, regardless of whether or not the reference URIs *path* is absolute or not:

```
>>> uri.resolve('file:/some/dir/', 'file:file.txt')
'file:file.txt'
```

Also, note that trailing slashes are significant for path resolution. Without a trailing slash, the base's final path segment is replaced when resolving paths:

```
>>> uri.resolve('file:/some/dir', 'other/dir/file.txt')
'file:/some/other/dir/file.txt'
```

1.5.2 URI Resolution in rnginline

When the Inliner sees a URI like `common/name.rng`, it needs to resolve it to to absolute URIL, such as `file:/a/b/proj/common/name.rng` in order to determine the location it points to, and which URL Handler can fetch the URL.

To do this, `rnginline` uses a heirachy of URIs:

1. **The *default base URI* — The catch-all, top-most URI. This has to be** absolute. By default it's `file:<current-dir>` but can be set to anything.
2. **The current document's base URI — The location the current document was** loaded from.
3. **Any `xml:base` attributes on, or on ancestors of the an** `<inline>/<externalRef>` XML element.
4. **The URI value of the `href` attribute of the `<inline>/<externalRef>`** element.

The absolute URI of an `href` attribute is resolved by resolving 2 against 1, then 3 against the result of that, then 4 against the result of that.

Because the default base URI is `file:<current-dir>`, relative paths/URIs passed to `inline()` get resolved to `file:` URLs, which are handled by the filesystem handler without having to specify the `file:` scheme on the input to `inline()`.

Similarly, following what we've learnt above, if an input's base URI is `pydata://my.pkg/some/dir/a.rng` and an `href` attribute in `a.rng` contains the value `b.rng`, it will be resolved to the absolute URL `pydata://my.pkg/some/dir/b.rng`, and therefore handled by the pydata handler, not the filesystem handler.

1.6 rnginline API

This is the Python API reference for rnginline.

1.6.1 rnginline

rnginline.**inline**(*source-arg*[, *optional-kwargs*])

Load an XML document containing a RELAX NG schema, recursively loading and inlining any `<include href="...">`/`<externalRef href="...">` elements to form a complete schema in a single XML document.

URLs in `href` attributes are dereferenced to obtain the RELAX NG schemas they point to using one or more URL Handlers. By default, handlers for `file:` and `pydata:` URLs are registered.

Keyword Arguments

- **src** – The source to load the schema from. Either an `lxml.etree` `Element`, a URL, filesystem path or file-like object
- **etree** – Explicitly provide an `lxml.etree` `Element` as the source
- **url** – Explicitly provide a URL as the source
- **path** – Explicitly provide a filesystem path as the source
- **file** – Explicitly provide a file-like object as the source
- **handlers** – An iterable of `UrlHandler` objects which are, in turn, requested to fetch each `href` attribute's URL. Defaults to the *rnginline.urlhandlers.file* and *rnginline.urlhandlers.pydata* in that order.
- **base_uri** – A URI to override the base URI of the schema with. Useful when the source doesn't have a sensible base URI, e.g. passing a file object like `sys.stdin`
- **postprocessors** – An iterable of `PostProcess` objects which perform arbitary transformations on the inlined XML before it's returned/ loaded as a schema. Defaults to the result of calling *rnginline.postprocess.get_default_postprocessors()*
- **create_validator** – If True, a validator created via `lxml.etree.RelaxNG()` is returned instead of an lxml `Element`
- **default_base_uri** – The root URI which all others are resolved against. Defaults to `file:<current directory>` which relative file URLs such as `'external.rng'` to be found relative to the current working directory.
- **inliner** – The class to create the `Inliner` instance from. Defaults to *rnginline.Inliner*.
- **create_validator** – If True, an lxml RelaxNG validator is created from the loaded XML document and returned. If False then the loaded XML is returned.

Returns A `lxml.etree.RelaxNG` validator from the fully loaded and inlined XML, or the XML itself, depending on the `create_validator` argument.

Raises `RelaxngInlineError` – (or subclass) is raised if the schema can't be loaded.

class `rnginline.`**`Inliner`**(*handlers=None*, *postprocessors=None*, *default_base_uri=None*)

Inliners merge references to external schemas into an input schema via their `inline()` method.

Typically you can ignore this class and just use *rnginline.inline()* which handles instantiating an `Inliner` and calling its `inline()` method.

`__init__`(*handlers=None*, *postprocessors=None*, *default_base_uri=None*)

Create an Inliner with the specified Handlers, PostProcessors and default base URI.

Parameters

- **`handlers`** – A list of URL Handler objects to handle URLs encountered by `inline()`. Defaults to the *rnginline.urlhandlers.file* and *rnginline.urlhandlers.pydata* in that order.
- **`postprocessors`** – A list of PostProcess objects to apply to the fully inlined schema XML before it's returned by `inline()`. Defaults to the result of calling *rnginline.postprocess.get_default_postprocessors()*
- **`default_base_uri`** – The root URI which all others are resolved against. Defaults to `file:<current directory>`

inline(*[src]*, ***kwargs*)
Load an XML document containing a RELAX NG schema, recursively loading and inlining any `<include>`/`<externalRef>` elements to form a complete schema.

URLs in <include>/<externalRef> elements are resolved against the base URL of their containing document, and fetched using one of this Inliner's urlhandlers.

Parameters

- **src** – The source to load the schema from. Either an lxml.etree Element, a URL, filesystem path or file-like object.
- **etree** – Explicitly provide an lxml.etree Element as the source
- **url** – Explicitly provide a URL as the source
- **path** – Explicitly provide a path as the source
- **file** – Explicitly provide a file-like as the source
- **base_uri** – A URI to override the base URI of the grammar with. Useful when the source doesn't have a sensible base URI, e.g. passing sys.stdin as a file.
- **create_validator** – If True, an lxml RelaxNG validator is created from the loaded XML document and returned. If False then the loaded XML is returned.

Returns A `lxml.etree.RelaxNG` validator from the fully loaded and inlined XML, or the XML itself, depending on the `create_validator` argument.

Raises `RelaxngInlineError` – (or subclass) is raised if the schema can't be loaded.

1.6.2 rnginline.urlhandlers

This module contains the built-in URL Handlers provided by `rnginline`.

URL Handler objects are responsible for:

- Saying if they can handle a URL — `can_handle(url)`
- Fetching the data referenced by a URL — `dereference(url)`

Default URL Handler instances

The following URL Handler objects are provided, ready to use:

rnginline.urlhandlers.**file**
The default instance of *FilesystemUrlHandler*

rnginline.urlhandlers.**pydata**
The default instance of *PackageDataUrlHandler*

They're also available via:

rnginline.urlhandlers.**get_default_handlers**()
Get a list of the default URL Handler objects.

URL Handler Classes

class rnginline.urlhandlers.**FilesystemUrlHandler**
A `UrlHandler` for `file:` URLs. This handler can resolve references to files on the local filesystem.

can_handle(*url*)
Check if this handler supports `url`.

This handler supports URLs with the `file:` scheme.

Parameters **url** – A URL as a string.

Returns True if `url` is supported by this handler, False otherwise

Return type bool

dereference(*url*)
Read the contents of the file identified by `url`.

Parameters **url** – A `file:` URL

Returns The content of the file as a byte string

Raises `DereferenceError` – if an `IOError` prevents the file being read

static **makeurl**(*file_path*, *abs=False*)
Create relative or absolute URL pointing to the filesystem path `file_path`.

(Absolute refers to whether or not the URL has a scheme, not whether the path is absolute.)

Parameters

- **file_path** – The path on the filesystem to point to
- **abs** – Whether the returned URL should be absolute (with a `file` scheme) or a relative URL (URI-reference) without the scheme

Returns A `file` URL pointing to `file_path`

Note: The current directory of the program has no effect on this function

Examples

```
>>> from rnginline.urlhandlers import file
>>> file.makeurl('/tmp/foo')
'/tmp/foo'
>>> file.makeurl('/tmp/foo', abs=True)
'file:/tmp/foo'
>>> file.makeurl('file.txt')
'file.txt'
>>> file.makeurl('file.txt', abs=True)
'file:file.txt'
```

static **breakurl**(*file_url*)
Decode a `file:` URL into a filesystem path.

Parameters **file_url** – The URL to decode. Can be an absolute URL with a `file:` scheme, or a relative URL without a scheme.

Returns The filesystem path implied by the URL

Examples

```
>>> from rnginline.urlhandlers import file
>>> file.breakurl('file:/tmp/some%20file.txt')
'/tmp/some file.txt'
>>> file.breakurl('some/path/file%20name.dat')
'some/path/file name.dat'
```

class rnginline.urlhandlers.**PackageDataUrlHandler**

A URL Handler which allows data files in Python packages to be referenced.

The URLs handled by instances of this class are layed out as follows:

```
pydata://<package-path>/<path-under-package>
```

For example `pydata://rnginline.test/data/loops/start.rng`.

can_handle(*url*)

Check if this handler supports `url`.

This handler supports URLs with the `pydata:` scheme.

Parameters **url** – A URL as a string.

Returns True if `url` is supported by this handler, False otherwise

Return type bool

dereference(*url*)

Get the contents of the data file identified by `url`

`pkgutil.get_data()` is used to fetch the data.

Parameters **url** – A `pydata:` URL pointing at a file under a Python package

Returns A byte string

Raises `DereferenceError` – If the data identified by the URL does not exist or cannot be read

classmethod makeurl(*package*, *resource_path*)

Create a URL referencing data under a Python package.

Parameters

- **package** – A dotted path you'd use to import the package in question
- **resource_path** – The path under the package to a data file

Returns A URL of the form `pydata://<package>/<resource_path>`

Return type

...

Example

```
>>> from rnginline.urlhandlers import pydata
>>> pydata.makeurl('mypkg.subpkg', 'some/file.txt')
'pydata://mypkg.subpkg/some/file.txt'
```

classmethod breakurl(*url*)

Deconstruct a `pydata:` URL into constituent parts.

Parameters **url** – A `pydata:` URL

Returns A 2-tuple of the package and path contained in the URL

Example

```
>>> from rnginline.urlhandlers import pydata
>>> pydata.breakurl('pydata://mypkg.subpkg/some/file.txt')
('mypkg.subpkg', 'some/file.txt')
```

1.6.3 rnginline.postprocess

rnginline.postprocess.**datatypelibrary = <rnginline.postprocess.PropagateDatatypeLibraryPostProcess object>**
The default instance of *PropagateDatatypeLibraryPostProcess*

rnginline.postprocess.**get_default_postprocessors**()
Get a list containing the default postprocessor objects.

Currently contains just *datatypelibrary*.

class rnginline.postprocess.**PropagateDatatypeLibraryPostProcess**
Implements the propagation part of simplification 4.3: datatypeLibrary attributes are resolved and explicitly set on each data and value element, then removed from all other elements.

This can be used to work around libxml2 not resolving datatypeLibrary attributes from div elements.

r

Symbols

B

C

D

F

G

I

M

P

R