
RNV — Relax NG Compact Syntax Validator in C

Version 1.7

Table of Contents

News since 1.6	1
New since 1.5	1
Aknowledgements	2
Package Contents	2
Installation	2
Invocation	3
Limitations	3
Applications	4
ARX	4
RVP	5
User-Defined Datatype Libraries	6
Datatype Library Plug-in	7
Scheme Datatypes	7
New versions	9

Abstract

RNV is an implementation of *Relax NG Compact Syntax*, <http://relaxng.org/compact-20021121.html>. It is written in ANSI C, the command-line utility uses Expat, <http://www.jclark.com/xml/expat.html>. It is distributed under BSD license, see `license.txt` for details.

RNV is a part of an on-going work, and the current code can have bugs and shortcomings; however, it validates documents against a number of grammars. I use it.

News since 1.6

The format for error messages is similar to that of Jing (file name, line and column are colon-separated). Entities and DTD processing is moved out of RNV, use XX, available from the same download location, to expand entities.

New since 1.5

Better reporting: required and permitted content is reported separately; it helps debug grammars. Several bugfixes; I relied on an acquired test suite and published schemata, but have found that I can make more bugs than they cover, thus a reworked an extended test suite is now used for testing. The code has also been cleaned up and simplified in places during porting to Plan9.

Acknowledgements

I would like to thank those who have helped me develop RNV.

Dave Pawson has been the first user of the program.

Alexander Peshkov helps me with testing and I have been able to correct very well hidden errors with his help.

Sebastian Rahtz encouraged me to continue working on RNV since the first release, and has helped me to improve it on more than one occasion.

Package Contents

Note

I have put `rnv.exe` and `arx.exe`, Win32 executables statically linked with a current version of Expat from <http://expat.sourceforge.net/>, into a separate distribution archive (with name ending in `-win32bin`). It contains only the program binaries and should be available from the same location as the source distribution.

The package consists of:

- the license, `license.txt`;
- the source code, `*.[ch]`;
- the source code map, `src.txt`;
- `Makefile.bsd` for BSD make;
- `Makefile.gnu` for GNU Make;
- `Makefile.bcc` for Win32 and Borland C/C++ Compiler;
- `tools/xck`, a simple shell script I am using to validate documents;
- `tools/*.rnc`, sample Relax NG grammars;
- `scm/*.scm`, program modules in Scheme, for Scheme Datatypes Library;
- the log of changes, `changes.txt`;
- this file, `readme.txt`.
- Other scripts, samples and plug-ins appear in `tools/` eventually.

Installation

On Unix-like systems, run **make -f Makefile.gnu** or **make -f Makefile.bsd**, depending on which flavour of make you have; `Makefile.bsd` should probably work on SysV, but, unfortunately, I have no place to check for the last couple of years. If you are using Expat 1.2, define `EXPAT_H` as `xmlparse.h` instead of `expat.h`.

On Windows, use `rnv.exe`. To recompile from the sources, use `Makefile.bcc` with Borland C/C++ Compiler, or create a makefile or project for your environment.

Invocation

The command-line syntax is

```
rnv {-q|-p|-c|-s|-v|-h} grammar.rnc {document1.xml}
```

If no documents are specified, RNV attempts to read the XML document from the standard input. The options are:

- q names of files being processed are not printed; in error messages, expected elements and attributes are not listed;
- n *<limit>* sets the maximum number of reported expected elements and attributes, -q sets this to 0 and can be overridden;
- p copies the input to the output;
- c if the only argument is a grammar, checks the grammar and exits;
- s uses less memory and runs slower;
- v prints version number;
- h displays usage summary and exits.

Limitations

- RNV assumes that the encoding of the syntax file is UTF-8.
- Support for XML Schema Part 2: Datatypes is partial.
 - ordering for `duration` is not implemented;
 - only local parts of `QName` values are checked for equality, `ENTITY` values are only checked for lexical validity.
- The schema parser does not check that all restrictions are obeyed, in particular, restrictions 7.3 and 7.4 are not checked.
- RNV for Win32 platforms is a Unix program compiled on Win32. It expects file paths to be written with normal slashes; if a schema is in a different directory and includes or refers external files, then the schema's path must be written in the Unix way for the relative paths to work. For example, under Windows, `mv` that uses `../schema/docbook.rnc` to validate `userguide.dbx` should be invoked as

```
rnv.exe ../schema/docbook.rnc userguide.dbx
```

Applications

The distribution includes several utilities built upon RNV; they are listed and described in the following sections.

ARX

ARX is a tool to automatically determine the type of a document from its name and contents. It is inspired by James Clark's schema location approach for nXML, <http://groups.yahoo.com/group/emacs-nxml-mode/message/259>, and is a development of the idea described in <http://relaxng.org/pipermail/relaxng-user/2003-December/000214.html>.

ARX is a command-line utility. The invocation syntax is

```
arx {-n|-v|-h} document.xml arx.conf {arx.conf}
```

ARX either prints a string corresponding to the document's type or nothing if the type cannot be determined. The options are:

- n turns off prepending base path of the configuration file to the result, even if it looks like a relative path (useful when the configuration file and the grammars are in separate directories, or for association with something that is not a file);
- v prints current version;
- h displays usage summary and exits.

The configuration file must conform to the following grammar:

```
arx = grammars route*
grammars = "grammars" "{" type2string+ "}"
type2string = type "=" literal
type = nmtoken
route = match|nomatch|valid|invalid
match = "=~" regexp "=>" type
nomatch = "!~" regexp "=>" type
valid = "valid" "{" rng "}" "=>" type
invalid = "!valid" "{" rng "}" "=>" type

literal=string in "'", '"' inside must be prepended by '\ '
regexp=string in '/', '/' inside must be prepended by '\ '
rng=Relax NG Compact Syntax
```

Comments start with # and continue till the end of line.

Rules are processed sequentially, the first matching rule determines the file's type. Relax NG templates are matched against file contents, regular expressions are applied to file names. The sample below associates documents with grammars for XSLT, DocBook or XSL FO.

```
    grammars {
docbook="docbook.rnc"
xslt="xslt.rnc"
xslfo="fo.rnc"
    }

    valid {
start = element (book|article|chapter|reference) {any}
any = (element * {any}|attribute * {text}|text)*
    } => docbook

    !valid {
default namespace xsl = "http://www.w3.org/1999/XSL/Transform"
start = element *-xsl:* {not-xsl}
not-xsl = (element *-xsl:* {not-xsl}|attribute * {text}|text)*
    } => xslt

    =~/.*\.\xsl/ => xslt
    =~/.*\.\fo/ => xslfo
```

ARX can also be used to link documents to any type of information or processing.

RVP

RVP is abbreviation for **R**elax **N**G **V**alidation **P**ipe. It reads validation primitives from the standard input and reports result to the standard output; it's main purpose is to ease embedding of a Relax NG validator into various languages and environment. An application would launch RVP as a parallel process and use a simple protocol to perform validation. The protocol, in BNF, is:

```
    query ::= (
quit
| start
| start-tag-open
| attribute
| start-tag-close
| text
| end-tag) z.
    quit ::= "quit".
    start ::= "start" [gramno].
    start-tag-open ::= "start-tag-open" patno name.
    attribute ::= "attribute" patno name value.
    start-tag-close ::= "start-tag-close" patno name.
    text ::= ("text"|"mixed") patno text.
    end-tag ::= "end-tag" patno name.
    response ::= (ok | er | error) z.
    ok ::= "ok" patno.
    er ::= "er" patno erno.
    error ::= "error" patno erno error.
    z ::= "\0" .
```

- RVP assumes that the last colon in a name separates the local part from the namespace URI (it is what one gets if specifies ‘:’ as namespace separator to Expat).
- Error codes can be grabbed from rvp sources by **grep_ER_*.h** and OR-ing them with corresponding masks from `erbit.h`. Additionally, error 0 is the protocol format error.
- Either **er** or **error** responses are returned, not both; `-q` chooses between concise and verbose forms (invocation syntax described later).
- **start** passes the index of a grammar (first grammar in the list of command-line arguments has number 0); if the number is omitted, 0 is assumed.
- **quit** is not opposite of **start**; instead, it quits RVP.

The command-line syntax is:

```
rvp {-q|-s|-v|-h} {schema.rnc}
```

The options are:

- `-q` returns only error numbers, suppresses messages;
- `-s` takes less memory and runs slower;
- `-v` prints current version;
- `-h` displays usage summary and exits.

To assist embedding RVP, samples in Perl (`tools/rvp.pl`) and Python (`tools/rvp.py`) are provided. The scripts use Expat wrappers for each of the languages to parse documents; they take a Relax NG grammar (in the compact syntax) as the command line argument and read the XML from the standard input. For example, the following commands validate `rnv.dbx` against `docbook.rnc`:

```
perl rvp.pl docbook.rnc < rnv.dbx
python rvp.py docbook.rnc < rnv.dbx
```

The scripts are kept simple and unobscured to illustrate the technique, rather than being designed as general-purpose modules. Programmers using Perl, Python, Ruby and other languages are encouraged to implement and share reusable RVP-based components for their languages of choice.

User-Defined Datatype Libraries

Relax NG relies on XML Schema Datatypes to check validity of data in an XML document. The specification allows the implementation to support other datatype libraries, a library is required to provide two services, `datatypeAllows` and `datatypeEqual`.

A powerful and popular technique is the use of string regular expressions to restrict values of attributes and character data. However, XML Schema regular expressions must be written as single strings, without

any parameterization; they often grow to several dozens of characters in length and are very hard to read or debug.

A solution for these problem would be to allow the user to define custom datatypes and to specify them in a high-level programming language. The user can then either use regular expressions as such, employ lex for lexical analysis, or any other technique which is best suited for each particular case (for example XSL FO datatypes would benefit from a custom datatype library). With many datatype libraries eventually implemented, it is likely that a clearer picture of the right language for validation of data will eventually emerge.

RNV provides two different ways to implement this solution; I believe that they correspond to different tastes and traditions. In both cases, a high-level language can be used to implement a datatype library, the language is not related to the implementation language of RNV, and RNV need not be recompiled to add a new datatype library.

Datatype Library Plug-in

A datatype plug-in is an executable. RNV invokes it as either

```
program allows type key value ... data
```

or

```
program equal type data1 data2
```

program is the executable's name, the rest is the command line; *key* and *value* pairs are datatype parameters and can be repeated. The program is executed for each datatype in library <http://davidashen.net/relaxng/pluggable-datatypes>; if the exit status is 0 for success, non-zero for failure.

Both RNV and RVP can use pluggable datatypes, and must be compiled with `DXL_EXC` set to 1 (**make `DXL_EXC=1`**) to support them, in which case they accept an additional command-line option `-d` with the name of the plugin as the argument. An implementation of XML Schema datatypes as a plugin (in C) is included in the distribution, see `xsdck.c`. For example,

```
rnv -d xsdck xslt-dxl.rnc $HOME/work/docbook/xsl/*/*.xsl
```

will validate all DocBook XSL stylesheets on my workstation against a grammar for XSLT 1.0 modified to use RNV Pluggable Datatypes Library instead of XML Schema Datatypes.

Scheme Datatypes

Another way to add custom datatypes to RNV is to use the built-in Scheme interpreter (SCM, <http://www.swiss.ai.mit.edu/~jaffer/SCM.html>) to implement the library in Scheme, a dialect of Lisp. This solution is more flexible and robust than the previous one, but requires knowledge of a particular programming language (or at least desire to learn it, and the result is definitely worth the effort).

To support it, SCM must be installed on the computer, and RNV or RVP must be compiled with `DSL_SCM` set to 1 (**make `DSL_SCM=1`**), in which case they accept an additional option `-e` with the name of a scheme program as an argument. The datatype library is bound to <http://davidashen.net/relaxng/scheme-datatypes>; a sample implementation is in `scm/dsl.scm`. For example,

```
rnv -e scm/dsl.scm xslt-dsl.rnc $HOME/work/docbook/xsl/*/*.xsl
```

check the stylesheets against an XSLT 1.0 grammar modified to use an RNV Scheme Datatypes Library implemented in `scm/dsl.scm`.

A Datatype Library in Scheme must provide two functions in top-level environment:

```
(dsl-equal? string string string)
```

and

```
(dsl-allows? string '((string . string)*) string)
```

To assist development of datatype libraries, a Scheme implementation of XML Schema Regular Expressions is included in the distribution as `scm/rx.scm`. The Regular Expression library is not just a way to re-implement the built-in datatypes. Owing to flexibility of the language it is much easier to write and debug regular expressions in Scheme, even if they are to be used with built-in XML Schema Datatypes in the end. For example, a regular expression for e-mail address, with insignificant simplifications, is:

```
pattern=
  "\\((([^\(\\)]|\\.)*\\) )?"
  ~ "[a-zA-Z0-9!#$%&'*+\\-/?\\^_`{|}~]+"
  ~ "\\.[a-zA-Z0-9!#$%&'*+\\-/?\\^_`{|}~+)*"
  ~ ""|"([^\(\\)]|\\.)*" ""
  ~ "@"
  ~ "[a-zA-Z0-9!#$%&'*+\\-/?\\^_`{|}~]+"
  ~ "\\.[a-zA-Z0-9!#$%&'*+\\-/?\\^_`{|}~+)*"
  ~ "\\([([^\(\\)]|\\.)*\\)"
  ~ "\\((([^\(\\)]|\\.)*\\)?"
```

which, even split into four lines, is ugly-looking and hard to read. Meanwhile, it consists of a few repeating subexpressions, which could easily be factored out, but the syntax does not have the means for that.

Using Scheme interpreter, it is as simple as

```
(define addr-spec-regex
  (let* (
    (atom "[a-zA-Z0-9!#$%&'*+\\-/?\\^_`{|}~]+")
    (person "\\([^\(\\)]|\\.)*")
    (location "\\([([^\(\\)]|\\.)*\\)")
    (domain (string-append atom "\\." atom)))
    (string-append
      (" domain "|" person ")
      "@"
      (" domain "|" location "))))
```

This code is much simpler to read and debug, and then the parts can be joined and added to the grammar for production use. Furthermore, it is easy to implement the parsing of structured regular expressions embedded into parameters of datatypes in Relax NG itself. `dsl.scm`, the sample datatype library, can handle parameter `s-pattern` with regular expressions split into named parts, and the example above becomes:

```
s-pattern=""  
  comment = "\\([^(\\|\\|)\\.)*\\"  
  atom = "[a-zA-Z0-9!#$%&'*+\\-\\/=?\\^_`{|}~]+"  
  atoms = atom "\\." atom "*"  
  person = "\\([^(\\|\\|)\\.)*\\"  
  location = "\\([^(\\|\\|)\\.)*\\"  
  local-part = "(" atom "|" person ")"  
  domain = "(" atoms "|" location ")"  
  start = "(" comment " )?" local-part "@" domain "( " comment " )?"  
  ""
```

addr-spec-dsl.rnc is included in the distribution.

New versions

Visit <http://davidashen.net/> for news and downloads.