CoParsing of RDF & XML

Jeremy J. Carroll
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2001-292
November 26th , 2001*

E-mail: jjc@hpl.hp.com

parsing,
coroutines,
XML, RDF,
inversion

RDF/XML is defined using two grammars, that for XML over text, and that for RDF over XML nodes. Conway's classic pipeline compiler architecture is extended to have two coparsers one for each grammar. A direct implementation of this architecture using Java threads for coroutines is presented. An alternate implementation, with the second parser inverted is discussed. Implications for other XML dialects are suggested.

1. Introduction

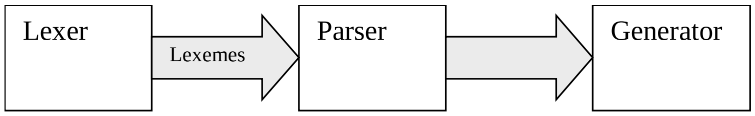

Figure 1: Traditional parsing

Traditional parser architectures consist of a dataflow pipeline with a single parser component (figure 1). This paper, in contrast, discusses an architecture for RDF/XML parsing consisting of a dataflow pipeline with two peer *coparsers*, one for XML and one for RDF (figure 2).

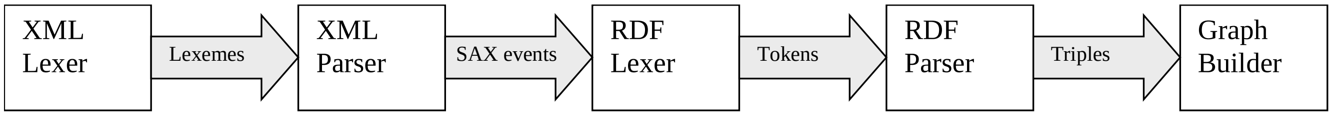

Figure 2: The top-level dataflow for RDF/XML parsing

This is the architecture of the author's RDF/XML parser: ARP [7].

The language of RDF/XML documents is defined by means of a grammar over XML Information Set [12] items. XML [6] in turn is defined by means of a grammar over text. Most uses of XML have a similarly two-layered specification often by means of a document type definition or an XML Schema ([4][13] and [28]).

The RDF grammar is specified by Model&Syntax [18] and rearticulated in [3].

A parser architecture reflecting this, uses a general-purpose XML parser for the initial analysis, which is then followed by analysis using the RDF/XML grammar.

1.1 Parsers and coroutines

From the beginning (Conway [11] in 1963), parser architectures have been conceptualized as dataflow pipelines between coroutines which are all first class processors, e.g. a lexer, a parser, a code generator, as in figure 1. Conway defines a coroutine as "an autonomous program which communicates with adjacent modules as if they were input or output subroutines" alternatively "coroutines are subroutines all at the same level, each acting as if it were the master program". Typically, during design, the parser becomes the dominant routine and the other coroutines are downgraded through an inversion transformation to be subroutines of the parser. This is exemplified by lex and yacc ([17] and [20]). Conceptually both systems can be used to write filters that take an input stream and produce an output stream. However, a basic design decision is that by default a yacc parser forms the main loop and the `yylex` function corresponding to the lex specification is a subroutine.

It is possible to have other realizations of Conway's basic coroutine architecture, e.g. by inverting the parser, and making it a subroutine to the lexer, but such approaches are not documented.

Many XML parsers follow the high level design of a yacc parser, with the parser being invoked as the main program, which in turn invokes both the lexer and subsequent stages as subroutines. In particular, the SAX event interface [22] follows this pattern, with the post-XML parsing phases being subordinate to the parser when considering the flow of control.

Conway's dataflow when applied to RDF/XML is naturally extended to have two parsers: one for parsing the tokenized input text using the XML grammar, and one for parsing the XML Infoset items so found using the RDF grammar. The full pipeline has five coroutines as in figure 2. This architecture is realized in ARP. We focus on the initial ARP implementation, which implements asynchronous coroutines using Java threads. We also discuss an alternative single threaded implementation. In this version, one of the parsers is inverted so that it can function as a subroutine to the other.

2. RDF/XML Parsers

We briefly survey the following three RDF/XML parsers, all written in Java: SiRPAC [27], VRP [29][30], RDFFilter [21]. Both SiRPAC and RDFFilter are built on top of SAX XML parsers. As such the main method of the program invokes the SAX parser. As various XML syntactic constructs are recognized the SAX parser invokes the callback methods known as SAX events. These methods are the link into the RDF parser.

In RDFFilter the parsing of RDF loosely follows an Augmented Transition Network design ([2] and [31]). The RDF parser switches between various states and has a single pushdown stack and arbitrary code augmenting the finite state machine. The apparent simplicity of such an approach belies the complexity of the code, which suffers from both too many states for easy maintenance and debugging, yettoo few for effective discrimination between the various cases.

SiRPAC was designed as a much more direct implementation of Model & Syntax [18], with code corresponding to the various paragraphs of the specification. Once again the requirement that the RDF parser be subordinate to the XML parser prevented the use of well-understood design such as top-down recursive descent. This too has proved to be difficult to maintain.

VRP is better structured as an RDF parser than either of the other two. It uses an explicit RDF/XML grammar which is compiled with CUP [14] into Java code. This grammar attacks both the RDF and XML parts of the specification at the same level and in one pass. CUP, like yacc, generates LALR(1) parsers through the use of shift & reduce tables (see [1] for definition of LALR(1)). VRP does not use an off-the-shelf XML parser, but instead contains its own XML parsing. This requires a duplication of effort, and suffers from some unnecessary defects in its XML processing, e.g. with respect to character references, in which it only supports the built-in references of XML. The LALR(1) parser is also somewhat awkward. It lacks the ability to easily pass information between the various productions; recursive descent parsers as generated with LL(k) systems such as JavaCC [24] and ANTLR [26] are usually easier to understand. Compared to SiRPAC and RDFFilter, VRP is a model of clarity.

3. A Dataflow Architecture

In contrast with the other RDF parsers we are starting from the explicit pipeline architecture shown in figure 2, in which there are two coparsers, one for XML parsing, the other for RDF parsing.

At the highest level of abstraction there are five processes. An XML lexer segments the input text into lexemes; these are parsed according to the XML grammar and a sequence of SAX events is generated; an RDF lexer recognizes those SAX events that are meaningful to the RDF grammar and generates a token sequence; these are parsed according to the RDF grammar, and a sequence of triples is generated; these are passed to a graph builder.

In common with other RDF/XML parsers we invert both the XML lexer and the graph generator. However we leave both parsers as first class coroutines. The RDF lexer is also inverted; within ARP it is placed as an output subroutine of the XML parser. This choice permits easy integration with SAX events. This is shown in figure 3, in which the arrows show dataflow, and containment shows subroutine relationships.

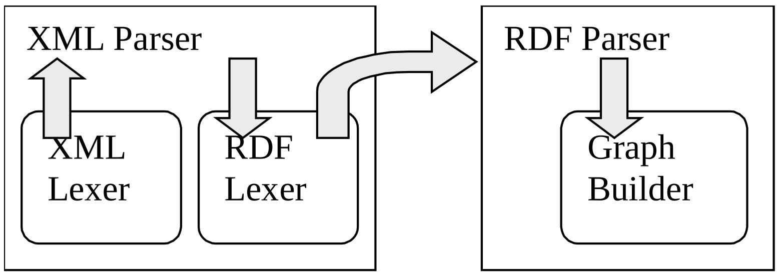

Figure 3: Coroutines and subroutines

We will refer to the XML and RDF parsers as coparsers. Having determined this top-level architecture the next level refinement involves determining quite what tokens are passed along the dataflow link between the two coparsers, and hence the nature of the RDF lexer.

Clues come from XML Information Set [12], the XPath [9] data model and from grammars for RDF like [3] and [8] which are explicitly in terms of items from such information models. In particular the XPath data model element, attribute and text nodes are constrained by an RDF grammar.

The architectural choice of a pipe linking the coparsers requires a linearization of the tree-like XML data model. This has two aspects:

- An end element token is required at the end of each element (rather like SAX events)
- Some order may be imposed on the unordered attribute nodes.

These steps are performed by the RDF lexer, which in addition, recognizes key qualified names [5] (*qnames*) on elements and attributes, such as `rdf:Description` and `rdf:about`.

4. Design with Java and JavaCC

Key design decisions concern the implementation of coroutines and the general approach to both parsers. Furthermore the RDF lexer needs further refinement with a complete definition of the dataflow between the coparsers.

A further design decision, omitted from this paper, is the triple production interface by which the RDF parser communicates with the rest of the RDF system. See Melnik's "Stanford API" [23] for an example approach.

4.1 Coroutines within Java

The Java Virtual Machine is a multithreaded environment without support for coroutines. As such, a natural, if expensive, approach to coroutines is to model each coroutine as a Java Thread. The sequence of execution is determined implicitly by the dataflow. The coroutines are linked using a pipe, e.g. Lea's bounded buffer [19].

The choice of pipe length has a dramatic impact on performance. The author's experiments for ARP suggested a surprisingly long optimal length of about 800 objects in the pipe.

Some programming languages have native coroutine support with explicit switching. These are likely to be significantly more efficient than using heavier weight concurrent programming constructs like threads and monitors.

Within Java, and within singly threaded environments, an alternative approach, explored below, is to invert one of the coparsers to make it a subroutine of the other one.

4.2 The XML Parser

The architecture is in part motivated by the availability of off-the-shelf XML parsers. The corresponding design decision is to identify a category of XML parsers. ARP uses a SAX2 compliant XML parser.

4.3 The RDF Parser

The architecture describes the RDF parser as a coroutine, i.e. “acting as if it were the master program”. At the design level this encourages the use of a standard parser design, e.g. a top-down recursive descent parser or an automatically generated one. Within Java the choice of JavaCC [24] or AntLR [26] as the parser generator allows the automatic generation of a top-down recursive descent parser, getting most of the benefits from both features.

During the development of ARP one version was coded with three different parser generators: JavaCC, CUP [14] and BYACC/Java [16].

CUP and BYACC/Java are very similar, principal differences being:

- CUP is the better known Java LALR(1) system
- BYACC/Java is a straight port of the very well-known yacc [17] LALR(1) system.
- CUP uses one stack internally, whereas BYACC/Java uses two.
- CUP is slightly faster.
- BYACC/Java uses the yacc $N convention for referring to the values of partial trees in action code, CUP allows names to be used.

With these observations BYACC/Java was eliminated from consideration.

JavaCC is better known than CUP, and has a significantly different design being an LL(1) parser with the ability for additional arbitrary lookahead where needed. The generated code is top-down recursive descent (see [1] for definition), with a Java method corresponding to each non-terminal in the grammar. Every non-terminal may be annotated with an arbitrary number of attributes that behave just like arguments to Java methods. This last feature allowed the annotation of the grammar rules to provide a clearer account of triple generation than with CUP which has no equivalent feature.

The most striking example of the use of this is with the property element productions. Each property element production describes the predicate and object of a triple. The subject of the triple is described in the parent element. With a JavaCC implementation this subject resource is passed as an argument into the property element production, and so the correct triple is generated in the logical place, within the property element production. Hand coded recursive descent code would also allow this. One possible CUP implementation is to pass the subject resource on an explicit stack that sits outside the grammar. Another, as in VRP [29], is for each property element to generate a [predicate, object] resource pair. A set of such pairs is passed back up to the parent production which then generates many triples using the subject resource and each pair from the set.

Speaking more formally, the semantics of the property element production is that each property element production corresponds to a function mapping a resource to a set of triples (usually a singleton set). Within CUP it is necessary to either have some explicit representation of this function (e.g. as a set of pairs) or to use a stack to extend CUP's abilities. Within JavaCC it is possible to code up this function directly in Java, passing the argument to the function in at the natural points within the code.

This feature of JavaCC determined that it was chosen as a key part of the design of the RDF coparser within ARP.

Further advantages of JavaCC were the following:

- It appeared to generate slightly faster parsers on like-for-like comparisons
- It has support for the EBNF [15] grammar constructs for repeated and optional elements.

4.4 The RDF Lexer

The functions of the RDF lexer are:

- To convert SAX events into tokens which are fed into the pipe from the XML parser to the RDF parser.
- To recognize key XML element tag qnames [5], attribute qnames, and attribute values.
- To appropriately order the attributes of each element according to a given partial order.

4.4.1 SAX Events

The SAX events that are pertinent to RDF parsing are:

- **`startElement`**

  Generating the following tokens in the pipe

  - A start element tag token

  And for each attribute:

  - An attribute name token
  - And an attribute value token

- **`endElement`**

  Generating a single fixed end element token in the pipe.

- **`characters`**

  Generating a single token in the pipe.

It may be noted that these tokens are rather closer to the XML Information Set [12] than the initial SAX events.

4.4.2 Key nodes

Element tags from the **`rdf`** namespace that are specifically recognized are: **`rdf:RDF`**, **`rdf:Description`**, **`rdf:li`** and **`rdf:_1, rdf:_2, ….`**

Attribute qnames from the **`rdf`** namespace that are specifically recognized are: **`rdf:ID, rdf:about, rdf:aboutEach, rdf:bagID, rdf:resource, rdf:parseType, rdf:type, rdf:_1, rdf:_2, ….`**

The two legal values of **`rdf:parseType`**, **`“Literal”`** and **`“Resource”`** are also specifically recognized.

Moreover **`rdf`** attributes are ordered in that ordered, after all attributes fromn the **`xml`** namespace and before other attributes.

We note that this choice of order means that the property attributes all come after any attribute that modifies the processing of that property attribute.

5. Error Handling

Error handling is often problematic in parsers. This particular architecture presents additional difficulties:

- Errors occur in both coroutines, but error handling needs to be coordinated across both. In particular, the invoking process can handle at most one catastrophic failure.
- The RDF Lexer detects certain types of errors (like unqualified attributes), but these are not errors in certain grammatical contexts. Hence the RDF lexer must coordinate with the RDF parser over error handling (which is in a different coroutine, and in the Java implementation in an asynchronous thread). In ARP, this coordination is achieved by the use of an error token inserted into the token stream flowing from the RDF Lexer to the RDF parser. The parser is then free to discard the error or to process it depending on grammatical context.

In addition there is the standard problem of making syntax error messages intelligible. The ARP approach is to have a look-up table for the follow sets of the most common syntax errors (Follow sets are defined in [1]). The look-up tables give a less cryptic message than JavaCC’s default message.

6. The Cost of Coroutines in Java

During the development, at the point where a choice was being made between JavaCC and CUP, another possibility was considered. This was to use CUP and invert the RDF parser to have a single threaded design with control relationships as in figure 4, again using containment to show subroutines, and arrows to show dataflow.

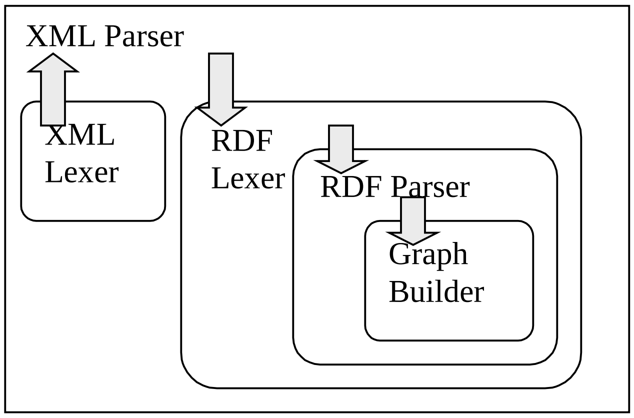

Figure 4: Inverting the RDF Parser

This reads: “The XML parser invokes the XML lexer repeatedly which returns a lexeme on each invocation. SAX events are handled by the RDF lexer which invokes the RDF parser with each token produced; this, in turn, invokes the graph builder with each triple produced.”

The CUP parser is an LALR(1) table driven parser which can be inverted straightforwardly within the object that encapsulates it. The inverted parser can be repeatedly invoked with the next lexical token, as opposed to the standard parser which repeatedly gets the next lexical token from the RDF lexer. This reverses the control relationship between the RDF lexer and the RDF parser. This process of inversion is not available to a JavaCC parser (or any recursive descent parser) because of the more extensive use of the call stack by a JavaCC parser.

Doing this inversion allowed a like-for-like comparison between the CUP version with coroutines and the inverted version with a subroutine. We note that the crucial difference from the perspective of the Java virtual machine, is that the coroutine version is multithreaded, whereas the subroutine version is singly threaded.

This single change saved approximately a quarter of the execution time. For the current version of ARP, which is slower because it spends more time on error checking, an estimated comparable figure is 15% of execution time. This is all spent on threading overhead. This is not the sort of inefficiency that the advocates of coroutines over the years have had in mind, and reflects the inadequacies of threads as the dominant concurrent programming paradigm.

6.1 Single Threaded Implementations and Compiler Compilers

We have seen that a singly threaded implementation requires the inversion of one of the coparsers. Given the desire to use an off-the-shelf XML parser, we must invert the RDF parser.

Top-down recursive descent parsers cannot be inverted without encapsulating the call-stack in the inversion. Hence the RDF parser must be implemented using a state machine of some sort.

It is well known that hand-coding state machines is error prone (RDFFilter [21], which follows exactly this design, is an example). Some state machine compiler is highly desirable. A good example of advanced state machine compilation is LALR(1) compiler compilers, which, with the CUP example, can be seen to be appropriate for the job. Ideally, we would use a compiler compiler that also processed arguments to grammar symbols as if they were procedural arguments, as JavaCC [24] and AntLR [26] do.

Inversion is also much harder if more than one token lookahead is permitted. With such grammars a backtracking trail must be maintained in the inversion object. This desire for table driven parsing with only one token lookahead strongly suggests LALR(1) as opposed to the more fashionable LL(k) grammars [25].

7. Applicability to other XML grammars

This dataflow architecture should be considered alongside the streaming SAX [22] XML processing model.

XML is extensible, and needs extension by a second level grammar, DTD or schema to make meaningful documents. For an arbitrary second level grammar we can relabel figure 2 to get figure 5.

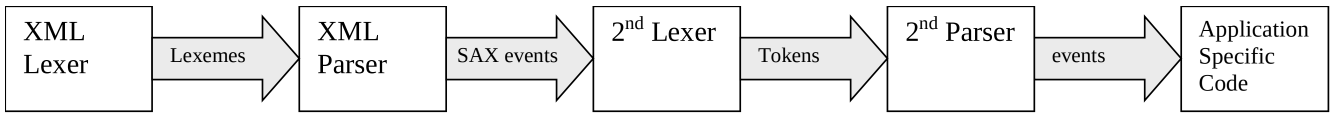

Figure 5: Two-level XML parsing

In a SAX-based approach, in which architectural considerations have not been clearly articulated, it is often difficult to distinguish those parts of the code that have areas of responsibility corresponding to the distinct functions of the Second Lexer, Second Parser and application code, of figure 5. For many XML dialects the SAX event handling code ends up as a somewhat muddled mix of:

- looking for special elements and attributes,
- state machine, often miscoded as a number of stateful variables that interact in an unclear fashion,
- application specific code and escapes

In fact, not dissimilar to RDFFilter [21].

Using this architecture to clean up such a system involves a clear separation between these three components; preferable with the state machine being clearly articulated alongside the higher level grammar (e.g. by means of a compiler compiler).

Further work may show how to compile annotated DTDs, XML Schema ([4], [13] & [28]) or Relax NG [10] grammars into this architecture and allow schema specific events or document models to be the application programmer's view of the XML document.

8. Conclusion

The clear architectural picture (figure 2) has allowed the development of the cleanest implementation of the RDF/XML grammar. Implementations of other XML dialects could benefit from an equally clear articulation of their architecture as a dataflow pipeline linking two distinct coparsers, and their ancillary components.

9. References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman *Compilers: Principles, Techniques and Tools* (also known as *The Red Dragon Book*) Addison-Wesley 1986.

[2] M. Bates, "The theory and practice of augmented transition network grammars". In *Natural Language Communication with Computers*, L. Bolc, Ed., Springer-Verlag, 1978, pp. 191-259.

[3] Dave Beckett (ed), *Refactoring RDF/XML Syntax* W3C Working Draft 2001 http://www.w3.org/TR/2001/WD-rdf-syntax-grammar-20010906/

[4] Paul V. Biron, Ashok Malhotra *XML Schema Part 2: Datatypes* W3C Recommendation, 2001, http://www.w3.org/TR/xmlschema-2/

[5] Tim Bray, Dave Hollander and Andrew Layman *Namespaces in XML*, World Wide Web Consortium Recommendation, 1999, http://www.w3.org/TR/REC-xml-names.

[6] Tim Bray, Jean Paoli and C. M. Sperberg-McQueen *Extensible Markup Language (XML) 1.0*, World Wide Web Consortium Recommendation, 1998 http://www.w3.org/TR/1998/REC-xml-19980210

[7] Jeremy J. Carroll, *Another RDF Parser*, 2001, http://www-uk.hpl.hp.com/people/jjc/arp

[8] James Clark, e-mail subject: *WD-rdf-syntax-grammar-20010906 in RELAX NG* http://lists.w3.org/Archives/Public/www-rdf-comments/2001JulSep/0248.html

[9] James Clark & Steve DeRose, *XML Path Language* World Wide Web Consortium Recommendation, 1999 http://www.w3.org/TR/xpath

[10] James Clark and MURATA Makoto (eds) *Relax NG Specification* OASIS, 2001. http://www.oasis-open.org/committees/relax-ng/spec-20010811.html

[11] Melvin E.Conway, "Design of a Separable Transition-Diagram Compiler", *Communications of the ACM* Vol. 6, No. 7, 1963, pp 396-408.

[12] John Cowan and Richard Tobin *XML Information Set* W3C Recommendation 2001 http://www.w3.org/TR/xml-infoset/

[13] David C. Fallside *XML Schema Part 0: Primer* W3C Recommendation, 2001, http://www.w3.org/TR/xmlschema-0/

[14] Scott E Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, *CUP Parser Generator for Java* (v0.10j) http://www.cs.princeton.edu/~appel/modern/java/CUP/

[15] ISO/IEC 14977:1996(E) *Information technology — Syntactic metalanguage — Extended BNF*

[16] Bob Jamison, *BYACC/J Java extension* http://troi.lincom-asg.com/~rjamison/byacc/

[17] S. C. Johnson and R. Sethi, ``Yacc: A parser generator'', *Unix Research System Programmer's Manual*, Tenth Edition, Volume 2

[18] Ora Lassila & Ralph R. Swick *Resource Description Framework (RDF) Model and Syntax Specification*, World Wide Web Consortium, 1999, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[19] Doug Lea *Concurrent Programming in Java: Design Principles and Patterns* Second edition Addison-Wesley, 1999.

[20] John Levine, Tony Mason & Doug Brown *lex & yacc*, 2nd Edition, O'Reilly, 1992.

[21] David Megginson, *RDFFilter* http://rdf-filter.sourceforge.net/

[22] David Megginson *SAX2* http://sax.sourceforge.net/

[23] Sergey Melnik, *RDF API Draft* (also known as *The Stanford API*) http://www-db.stanford.edu/~melnik/rdf/api.html

[24] MetaMata, Inc & Sun Microsystems *JavaCC* http://www.webgain.com/products/java_cc/

[25] T. J. Parr, *Obtaining Practical Variants Of LL(k) And LR(k) For k>1 By Splitting The Atomic k-Tuple*, Ph.D. Dissertation, School of Electrical Engineering, Purdue University, 1993.

[26] Terrence Parr and Russell Quong "ANTLR: A Predicated-LL(k) Parser Generator" *Journal of Software Practice and Experience*, Vol. 25(7), 789-810 (1995).

[27] Janne Saarela, *SiRPAC*, Last version: http://www.w3.org/RDF/Implementations/SiRPAC/SiRPAC-1.17.jar

[28] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn *XML Schema Part 1: Structures* W3C Recommendation, 2001, http://www.w3.org/TR/xmlschema-1/

[29] Karsten Tolle, *Analyzing and Parsing RDF*, Master's Thesis, Universität Hannover & University of Crete, 2000.

[30] Karsten Tolle and Vassilis Christophides, "ICS – VRP: a Tool for Parsing and Validating RDF Metadata & Schema", *ERCIM News* No.41, 2000, http://www.ercim.org/publication/Ercim_News/enw41/tolle.html

[31] Woods, W.A., "Transition network grammars for natural language analysis", *Communications of the ACM* 13:10, 1970, pp. 591-606.