

Efficient implementation of content models with numerical occurrence constraints

Henry S. Thompson

University of Edinburgh
World Wide Web Consortium
Markup Technology Ltd.

ht@inf.ed.ac.uk

Introduction

W3C XML Schema allows numerical occurrence ranges in the regular expressions over element declarations and wildcards which it uses to express content models, to e.g. to allow between 2 and 10 occurrences of some element. In [ThomTob03] we introduced an approach to translating such regular expressions into deterministic finite-state automata by extending the standard translation for regular expressions without numerical occurrence ranges [KThom68], [AhoUllman77] to *unfold* terms with numerical occurrence ranges. Although straightforward, the Thompson and Tobin algorithm is expensive in terms of space. When numerical occurrence ranges are nested, the resulting FSA has a number of states proportional to the *product* of the length of the ranges. In the worst case, this renders the algorithm unusable.

In this paper I present a new approach to implementing such models without unfolding. It uses FSAs augmented with ranges, and is space-efficient even when nested ranges are involved. The deterministic interpretations (greedy or modest) of such FSAs are also time-efficient, but fail to recognise languages involving certain configurations of nested ranges correctly. Implementers

may none-the-less find the construction described herein useful, as the configurations which are not recognised correctly by the deterministic interpretations are a) detectable at conversion time and b) extremely rare in practice, so falling back to a non-deterministic interpretation in such cases is probably acceptable.

In this paper FSAs with ranges are first defined and an implementation using counters described, then algorithms for conversion from regular expressions with numerical occurrence ranges and for simplification are presented, in a form paralleling the presentation in [ThomTob03].

Finite-state automata with ranges

Traditionally a finite-state automaton with respect to a given alphabet is a four-tuple consisting of a set of states, a subset thereof which are *start* states, another subset which are *final* states, and a set of transitions which are themselves a triple of start state, symbol from the alphabet (or λ , for null transitions) and end state. If

- a) There is only one start state;
- b) There are no null transitions and
- c) The relation over states defined by the transitions is functional

then the automaton is *deterministic*.

We define a finite-state automaton *with ranges* (rFSA for short) with respect to a given alphabet as a five-tuple:

- 1) A set of states;
- 2) A subset thereof which are *start* states;
- 3) A subset thereof which are *final* states;
- 4) A set of *ranges*;
- 5) A set of transitions, each of which is a five-tuple:
 - a. A state, the *start* state;
 - b. A symbol from the alphabet, or λ , or a range (in which case we call this an *increment* transition);
 - c. A set of ranges, the *minGuards*;
 - d. A set of ranges, the *maxGuards*;
 - e. A state, the *end* state.

Ranges are half-open, as follows:

- 1) A non-negative integer, the inclusive *minimum*;
- 2) The exclusive *maximum*, a positive integer or *, for infinity.

For example, one rFSA for the language consisting of a sequence of 'a's between 5 and 10 in length, inclusive, would be as follows:

$$\begin{aligned} &< \{1, 2, \$\}, \{1\}, \{\$\}, \{c: [5, 10]\}, \\ &\{ < 1, a, \{\}, \{c\}, 2 > \\ &\quad < 2, c, \{\}, \{\}, 1 > \\ &\quad < 1, \bullet, \{c\}, \{\}, \$ > \} > \end{aligned}$$

We can diagram this rFSA as follows:

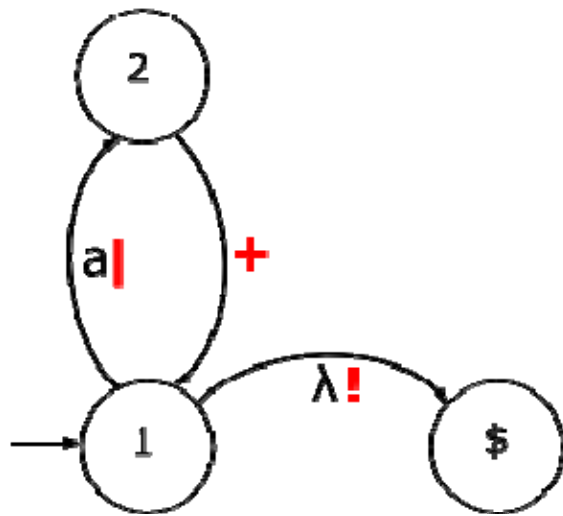


fig1.png

Figure 1. An rFSA for a^{5-10}

The above figure introduces graphical conventions used throughout this paper: Ranges are distinguished by colour, exclamation marks (!) indicate a minGuard and plus-signs (+) indicate an increment and vertical bars (|) indicate a maxGuard.

The interpretation of rFSAs is defined in terms of conditions on a sequence of transitions.

Define the *count* for a range at a transition in a sequence of transitions as the number of increment transitions for that range before that transition and after the most recent transition with that range in its minGuards, if any.

A string is in the language accepted by a rFSA if and only if

- 1) there is a sequence of transitions beginning with a start state and ending with a final state such that the concatenation of all the alphabet edge's symbols is the string and

- 2) all the guards of all the transitions in the sequence are satisfied:
 - a. A range in a minGuards is *satisfied* iff its count is greater than or equal to the range's minimum.
 - b. A range in a maxGuards is *satisfied* iff its count is less than the range's maximum.

Note that the maximum is treated as exclusive since the guard is checked *before* any increment. The utility of this formulation will become apparent later on.

Consider the regular expression $(a^{1-2}, b^?)^2$. The diagram for one possible rFSA for the same language is given below.

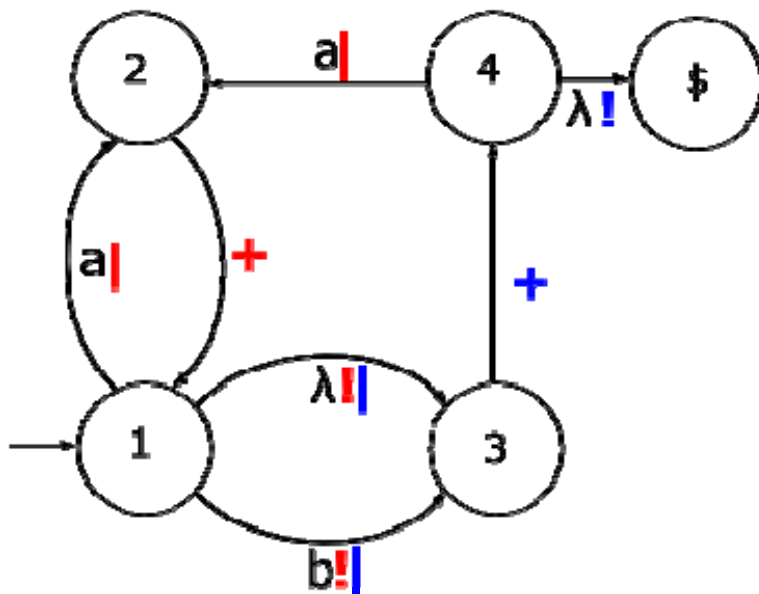


fig2.png

Figure 2. An rFSA implementing nested ranges

One sequence of transitions accepting the string *aaa* would then be

- 1 a | 2, 2 + 1,
- 1 λ! | 3, 3 + 4,
- 4 a | 2, 2 + 1, 1 a | 2, 2 + 1,
- 1 λ! | 3, 3 + 4, 4 λ! \$

Note in particular that the 3rd transition on the 3rd line satisfies its maxGuard of 2 with respect to the inner (red) range despite there being two relevant (red) increments earlier in the sequence, because only one of these is after the most recent relevant (red) minGuard.

The interpretation of the ranges and guards can be stated procedurally in terms of counters:

- 1) Every range has a counter associated with it, initialised to zero;

- 2) Guards inhibit transitions if their associated counter isn't within their range. That is, for a transition to be allowed
 - a. For each range in the maxGuards, the associated counter must be less than the maximum;
 - b. For each range in the minGuards, the associated counter must be greater than or equal to the minimum.
- 3) Increment transitions increment the counter associated with their range.
- 4) A successful transition sets the counters associated with all the ranges in its minGuards to zero. This is only relevant when minGuarded transitions are inside loops—these function as exits from an inner loop, and thus their counter needs to be reset in case the outer loop iterates again.

Conversion from regular expressions

This section addresses only the translation of occurrence indicators (i.e. '?', '+', '*' and numeric ranges). The reader is referred to *[ThomTob03]* for the details of the appropriate translations of XML Schema choices, sequences and terms ('all' groups are discussed below) or to *[KThom68]* or *[AhoUllman77]* for the corresponding aspects of traditional regular expressions. In what follows the phrase *term machine* (abbreviated TM) will be used to refer to the translations of such elementary components of XML Schema content models and traditional regular expressions.

We depend on, and preserve, the invariant that all term machines have only one start state and one final state, and that no transitions leave the final state.

Given a term machine TM for a regular expression without occurrence indicators, with start state TM_s and final state TM_f , we translate that regular expression *with* occurrence indicators as follows:

Optionality (?)

Add a λ transition from TM_s to TM_f , which remain the start and final states respectively.

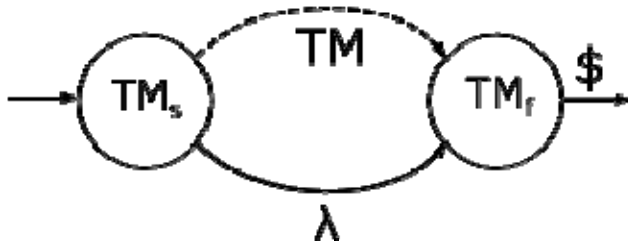


fig3.png

Figure 3. Making a term machine optional

Repetition (*)

Add λ transitions from TM_f to TM_s and from TM_s to a new state, which replaces TM_f as the final state. TM_s remains the start state.

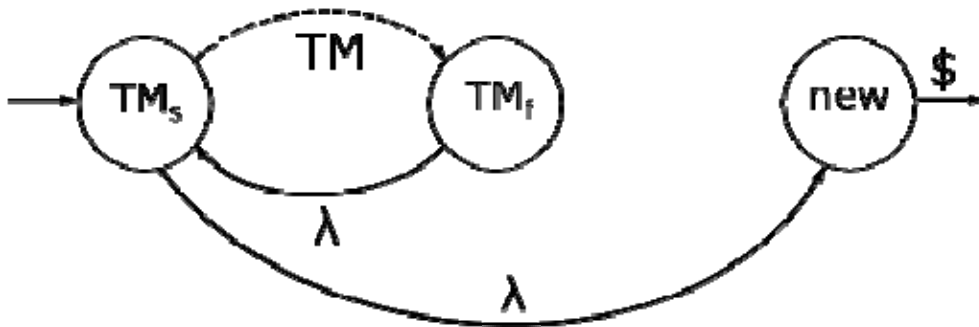


fig4.png

Figure 4. Optionally repeating a term machine

Repetition (+)

Add a λ transition from TM_f to TM_s and from TM_f to a new state, which replaces TM_f as the final state. TM_s remains the start state.

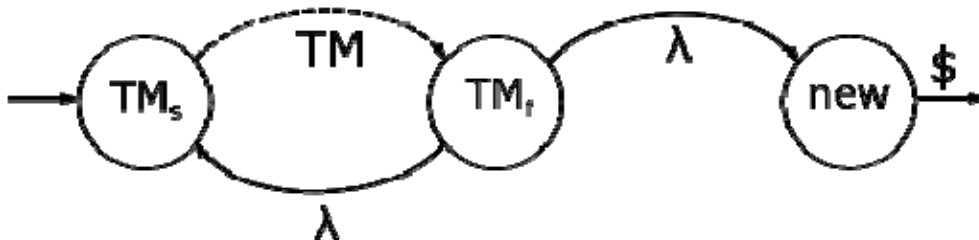


fig5.png

Figure 5. Repeating a term machine at least once

Numeric occurrence range (n-m)

Create a new range c of the form $[n,m)$ and three new states $n1$, $n2$ and $n3$. $n1$ is the new start state and $n3$ the new final state. Add new transitions as follows:

- $\langle n1, \lambda, \{\}, \{c\}, TM_s \rangle$
- $\langle TM_f, c, \{\}, \{\}, n2 \rangle$
- $\langle n2, \lambda, \{\}, \{\}, n1 \rangle$
- $\langle n1 \text{ if } n=0 \text{ otherwise } n2, \lambda, \{c\}, \{\}, n3 \rangle$

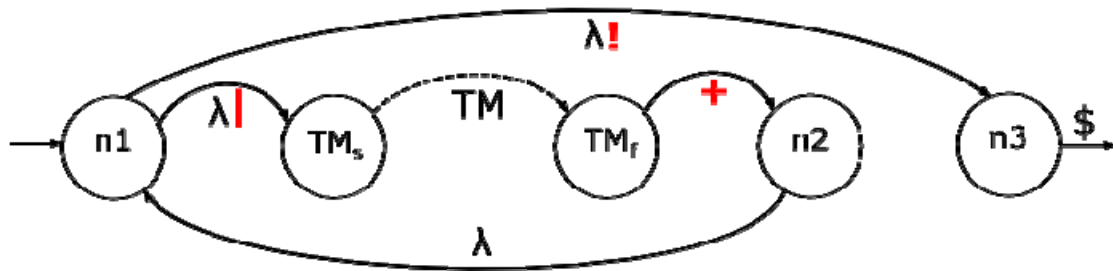


fig6.png

Figure 6. Repeating a term machine per a numeric range of the form $[0,m)$

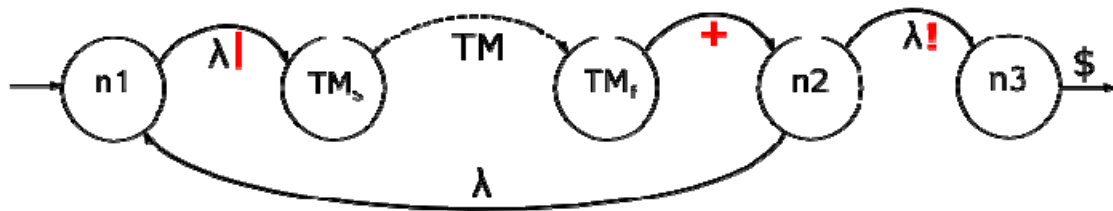


fig7.png

Figure 7. Repeating a term machine per a numeric range $[n>0,m)$

Note that the recent figures use a different convention for notating the final state, using a transition labelled '\$'. Indeed in what follows we will change the formulation of an rFSA to remove the notion of final state altogether, and add a fourth variety of transition, which we will call an exit transition, formally a triple of start state, minGuards and maxGuards, which we will write as

- $\langle \text{state}, \$, \text{minGuards}, \text{maxGuards} \rangle$

It should be evident how each of the above conversions achieves the correct semantics. In particular it should be clear that on reaching state $n3$, the counter will always have a value in the appropriate range. The details of *why* the guarded transitions and the increment are placed where they are with respect to the term machine will emerge in the next section.

Simplifying FSAs with ranges

If the approach described here is to be used for W3C XML Schema, following [ThomTob03] we need to simplify the verbose automata which result from the above conversion process, so that we can read violations of the Unique Particle Attribute constraint off the result. What is in fact required is to eliminate all empty transitions.

This is accomplished using a variation of the closure and subset constructions given in [AhoUllman77]. The variations are needed to avoid losing the impact of any guards on empty transitions.

First, we modify the closure construction so that every state in the closure of a state is annotated with *all* the guards on any empty edges which were traversed in the process of adding it to the closure. In checking closure identity, annotations *are* taken in to account.

Second, we modify the subset construction so that whenever a transition is added from a closure-labelled state, it acquires all the guards which annotate its original start state in that closure.

Using the same kind of pseudo-code as [AhoUllman77], here are the closure and subset constructions for rFSAs:

```
begin // CLOSURE(s) wrt rFSA R
  push s[\{\}, \{\}] onto STACK;
  add s[\{\}, \{\}] to CLOSURE(s);
  while STACK not empty do
    begin
      pop p[ii, aa], the top element of STACK, off of STACK;
      for each empty transition from p in R ( of the
        form  $\langle p, \lambda, i, a, q \rangle$  ) do
        begin
          if q[ii  $\cup$  i, aa  $\cup$  a] is not in CLOSURE(s) do
            begin
              add q[ii  $\cup$  i, aa  $\cup$  a] to CLOSURE(s);
              push q[ii  $\cup$  i, aa  $\cup$  a] onto STACK;
            end
          end
        end
      end
    end
  end
```

Figure 8. Computation of CLOSURE(*s*) for rFSAs

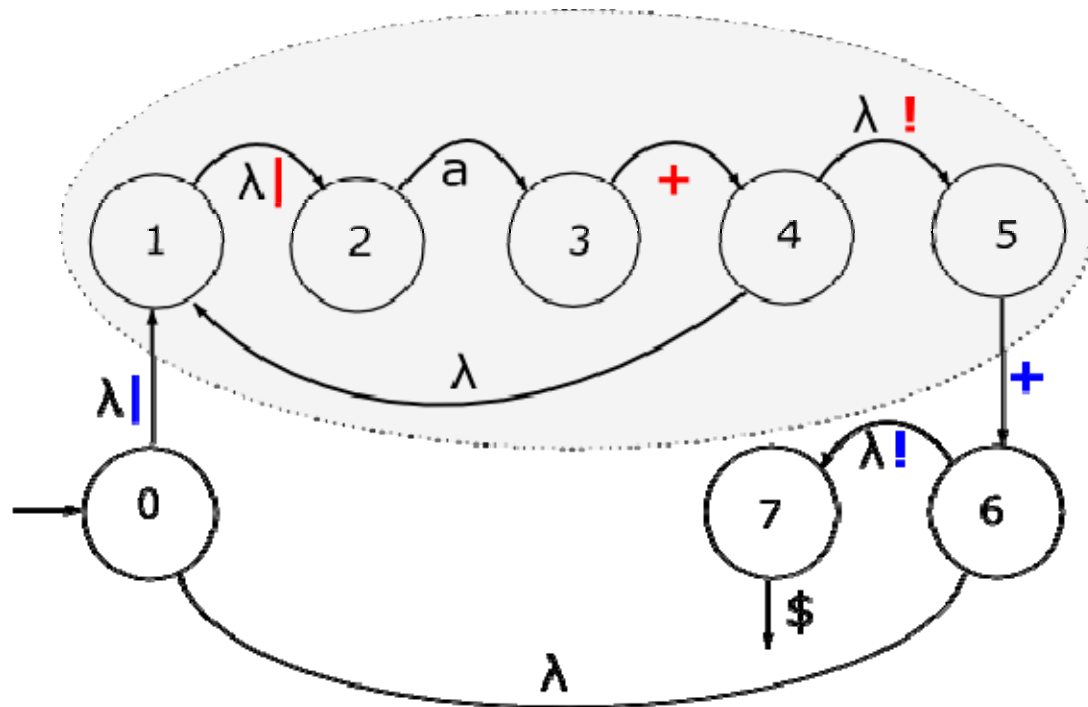
```

begin // SUBSET(rFSA R with start state  $s_0$ )
  D := an rFSA with the same alphabet as R and start
        state CLOSURE( $s_0$ ) wrt R, unmarked;
while there is an unmarked state x of D do
  begin
    mark x;
    for each  $s[ii,aa]$  in x do
      begin
        for each non-empty, non-final transition from s
          in R ( of the form  $\langle s, \alpha, i, a, t \rangle$  or
                 $\langle s, c, i, a, t \rangle$  ) do
          begin
            y := CLOSURE(q) wrt R;
            if y has not yet been added to the
              set of states of D then
              make y an "unmarked" state of D;
              add a transition to D:
                 $\langle x, \alpha, iiUi, aaUa, y \rangle$  or
                 $\langle x, c, iiUi, aaUa, y \rangle$ , as appropriate;
            end
          for each final transition from s
            in R ( of the form  $\langle s, \$, i, a \rangle$  ) do
            begin
              add a transition to D:
                 $\langle x, \$, iiUi, aaUa \rangle$ ;
            end
          end
        end
      end
    return D
  end

```

Figure 9. The subset construction for rFSAs

To illustrate the operation of these constructions, consider the regular expression $(a^{1-2})^2$. The figure below shows both the initial rFSA as produced by the conversion process described above, and the simplified result of the constructions.



[1,2)
[2,2)

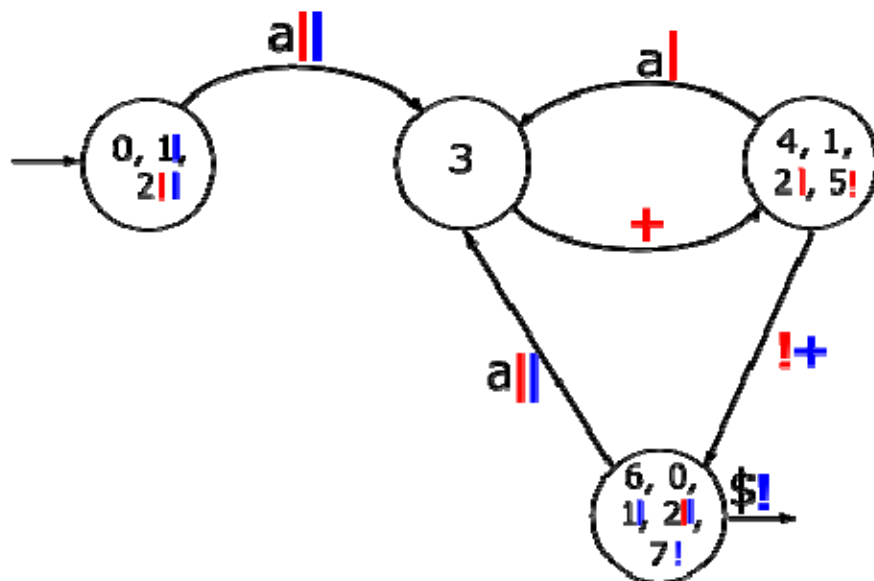


fig10.png

Figure 10. rFSAs for $(a^{1-2})^2$, before and after simplification

The above figure illustrates why we moved to using an exit transition, because it allows for the use of guards. Note that the states in the second, simplified, rFSM above are labelled with a representation of the corresponding closure, with the coloured vertical bars and exclamation marks giving the annotations. For example, the red exclamation mark on the '5' in the upper-right-most

state traces back to the red exclamation mark on the empty transition from state 4 to state 5 in the original automaton, because it is that empty transition which includes state 5 in $\text{CLOSURE}(4)$. This annotation in turn explains the red exclamation mark on the blue increment transition, since that increment transition originally came from state 5. In terms of the functioning of this red-exclamation-mark/blue-plus transition, it is essentially the exit from the inner loop—it checks that at least one ‘a’ has been seen by the inner loop, resets the inner (red) counter and increments the outer (blue) one.

Determinism issues

As presented, the algorithms given above do not produce deterministic automata, for two different reasons, one corrigible and the other not.

First of all, as presented the subset construction does not take account of multiple transitions from the same state with the same label. As noted above, this is because for W3C XML Schema purposes, where per *[ThomTob03]* transitions are labelled with element declarations or wildcards, and given the conversion rules used, there can *be* no such transitions in automata converted from XML Schema content models. It would be straightforward to further simplify automata with simple alphabetic labels, by modifying the subset construction given above to include the relevant check from the original construction as given in *[AhoUllman77]*.

But the second reason goes deeper. Consider again the regular expression $(a^1-2)^2$. This could be said to accept the string `aaa` in two different ways. This is reflected in the simplified automaton shown in Figure 10 by a non-determinism at the state labelled `4, 1, 2|, 5!`. Consider the situation after the first ‘a’ has been consumed, the red counter incremented, and we are in that state. We have two choices: we can be greedy, and take the ‘a’ transition back to state 3. Or we can be modest and exit the inner loop by taking the guarded increment transition on to state `6, 0, 1||, 2|, 7!`.

Whichever choice we make, if we make it as a uniform implementation strategy, we will make mistakes. If we’re always greedy (that is, always prefer alphabet edges to increment edges), we will fail to recognise `aa`. If we’re always modest (that is, always prefer increment edges to alphabet edges), we’ll fail to recognise `aaaa`. (Both strategies need to rate exit edges lowest of all.)

Full generality, then, requires an implementation without access to a parallel substrate to manage the non-determinism in some way, for example by recording and, if necessary, revisiting backtracking checkpoints, or by pseudo-parallelism.

W3C XML Schema implementation choices

Given that the perceived cost of general solutions is high, does this mean the approach set out herein is unlikely to prove attractive for W3C XML Schema implementations? I think not. Let's look a little harder at the circumstances in which the problem actually arises.

A greedy deterministic implementation will perform correctly for any content model (or regular expression) which obeys the following constraints:

- Vulnerable particles within strong repeated particles must either have $\{\text{minOccurs}\} = \{\text{maxOccurs}\}$ or $\{\text{minOccurs}\} = 0$.
- Vulnerable particles within weak repeated particles must either have $\{\text{minOccurs}\} = \{\text{maxOccurs}\}$ or $\{\text{minOccurs}\} = 0$ or $\{\text{maxOccurs}\} = \text{unbounded}$.
- A *repeated particle* is one whose $\{\text{minOccurs}, \text{maxOccurs}\}$ is other than $\{0, 1\}$ or $\{1, 1\}$
- A repeated particle is *weak* if $\{\text{minOccurs}\} \leq 1$, otherwise it's *strong*
- A particle is *vulnerable* if everything both before and after it in its enclosing particle is optional, that is, the input it accepts may be all that its enclosing particle accepts.

Our long-running example of $(a^{1-2})^2$ can be seen to be at risk because the internal particle is vulnerable (it's the only thing within its enclosing particle), the enclosing particle is strong, and the vulnerable (enclosed) particle has neither $\text{min} = \text{max}$ nor $\text{min} = 0$.

I say 'at risk' for content models which don't satisfy the invariants, because some models which don't will none-the-less always be handled correctly by a greedy implementation, for example $(a^{1-2})^*$. A precise definition of the conditions under which a greedy implementation will fail requires moderately complex number theory, but improving the coverage of the invariants given above should be possible none-the-less.

In practice, nested numeric ranges are rare in W3C XML Schema content models. Only a handful of the examples in my private schema document collection (approx. 1750 schema docs) contain numeric exponents at all, and *all* of them satisfy the constraints set out above. I converted XSV, the W3C XML Schema validator [*XSV*] to use this strategy and give an error if it encounters any content model violating the above constraints, and I've had only one contact from someone who encountered a problem as a result, and

this occurred in a schema document specifically designed to explore implementations of nested numeric ranges.

Given that ‘at risk’ models can be detected at conversion time, the in-principle best solution would seem to be to use the greedy deterministic implementation for all models which satisfy the constraints, and invoke some form of non-deterministic implementation.

Related Work

I have been unable to locate any existing work on FSAs with ranges or counters. As Kilpeläinen and Tuhkanen remark in their excellent work on the subject [*KilpTuh03*], this is paralleled by a dearth of published work on regular expressions with numerical occurrence indicators. Both this work and [*Kilp04*] discuss the use of such regular expressions in W3C XML Schema, and establish some interesting complexity results. It is worth noting that the result of most direct relevance to the approach discussed here, namely that checking language inclusion between two regular expressions with numerical occurrence indicators is NP-hard, depends on a construction which produces expressions which do not satisfy the constraints set out in the preceding section, i.e. correct implementation of recognition for the equivalent rFSAs would involve backtracking or some other mechanism for handling the necessary non-determinism therein.

Bibliography

[AhoUllman77] Aho, A. and J. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.

[KilpTuh03] Kilpeläinen, P. and R. Tuhkanen, “Regular expressions with numerical occurrence indicators—preliminary results”. In *Proc. of the Eighth Symp. on Programming Languages and Software Tools*, pages 163–173. University of Kuopio, 2003. Available online at <http://www.cs.uku.fi/tutkimus/publications/reports/A-2003-1/page163.pdf>

[Kilp04] P. Kilpeläinen, *Inclusion of Unambiguous #REs is NP-Hard*, 2004. Available online at http://www.cs.uku.fi/~kilpelai/numRE_incl_is_hard.pdf

[KThom68] K. Thompson, “Regular Expression Search Algorithm”, *Communications of the ACM*, 11(6):419–422, June 1968.

[ThomTob03] Thompson, H. S. and R. Tobin, "Using Finite State Automata to Implement W3C XML Schema Content Model Validation and Restriction Checking", *Proceedings of XML Europe 2003*, IDEAlliance, London, 2003. Available online at http://www.idealliance.org/papers/dx_xml03/papers/02-02-05/02-02-05.html (imperfectly rendered) or at http://www.ltg.ed.ac.uk/~ht/XML_Europe_2003.html.

[XSV] Thompson, H. S. and R. Tobin, XSV: XML Schema Validator, 2000. Description and pointer to form-based interface available online at <http://www.ltg.ed.ac.uk/~ht/xsv-status.html>