Creole: Validating Overlapping Markup

Jeni Tennison, Jeni Tennison Consulting Ltd, *jeni@jenitennison.com, http://www.jenitennison.com*

Abstract

Overlapping structures are common in real-world documents. Structures such as comments, revisions and pages overlap with the main document structures of sections and paragraphs. There are many ways of representing overlapping structures: even within the confines of well-formed XML, we can use empty elements or processing instructions as milestones to indicate their limits, and non-XML syntaxes give even more flexibility. But validating them has always been a problem.

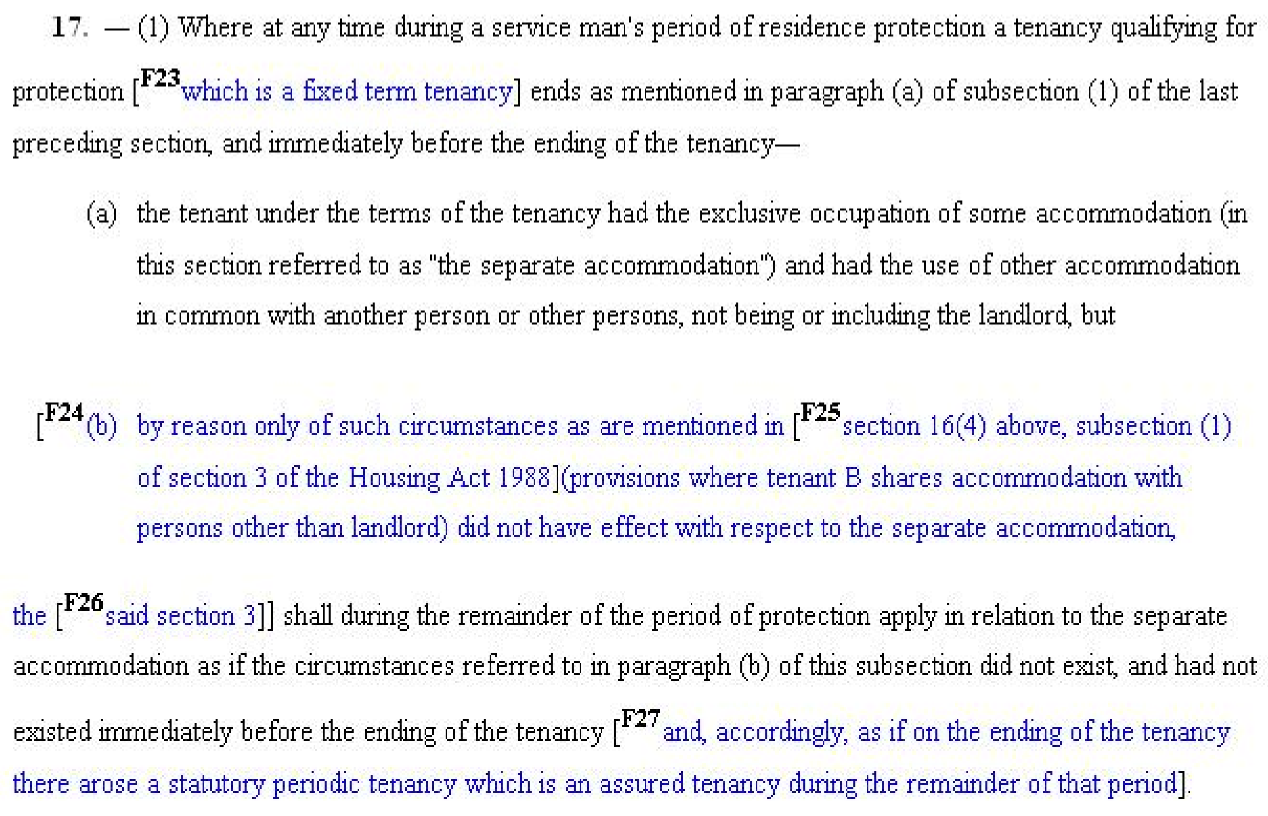
This paper introduces Creole (Composable Regular Expressions for Overlapping Languages etc.). Where previous methods of validating overlapping structures split documents into separate, well-formed, sub-documents for validation, Creole provides one grammar that defines exactly how the structures can overlap. As an extension of Relax NG, Creole has a simple, consistent implemention based on Brzozowski derivatives that can be applied to documents with overlapping structures, whatever their syntax. As such, Creole is a clear step forward in the support of mark-up in real-world documents.

Introduction

One of the fundamental rules of XML is that elements nest neatly inside each other: each element's end tag must appear before its parent's end tag. This restriction means that XML documents can be represented as a ordered, rooted tree, which is a very handy data structure for processing.

But ordered, rooted trees aren't a natural way of representing documents. Real-world documents contain structures that overlap each other.

Figure 1 shows a screenshot of some UK legislation from the Statute Law Database (www.statutelaw.gov.uk). This is consolidated legislation, which means that changes that have been made to the legislation by later Acts are highlighted, here in blue with square brackets, and explanations given in footnotes. There is no restriction on where changes can be made, and sometimes they overlap the main structure of the document. In the example in Figure 1, the change indicated by F24 starts at the beginning of list item (b) and continues into the paragraph that follows the list. In other words, the change overlaps a list and a paragraph.



17. — (1) Where at any time during a service man's period of residence protection a tenancy qualifying for protection [F23which is a fixed term tenancy] ends as mentioned in paragraph (a) of subsection (1) of the last preceding section, and immediately before the ending of the tenancy—

(a) the tenant under the terms of the tenancy had the exclusive occupation of some accommodation (in this section referred to as "the separate accommodation") and had the use of other accommodation in common with another person or other persons, not being or including the landlord, but

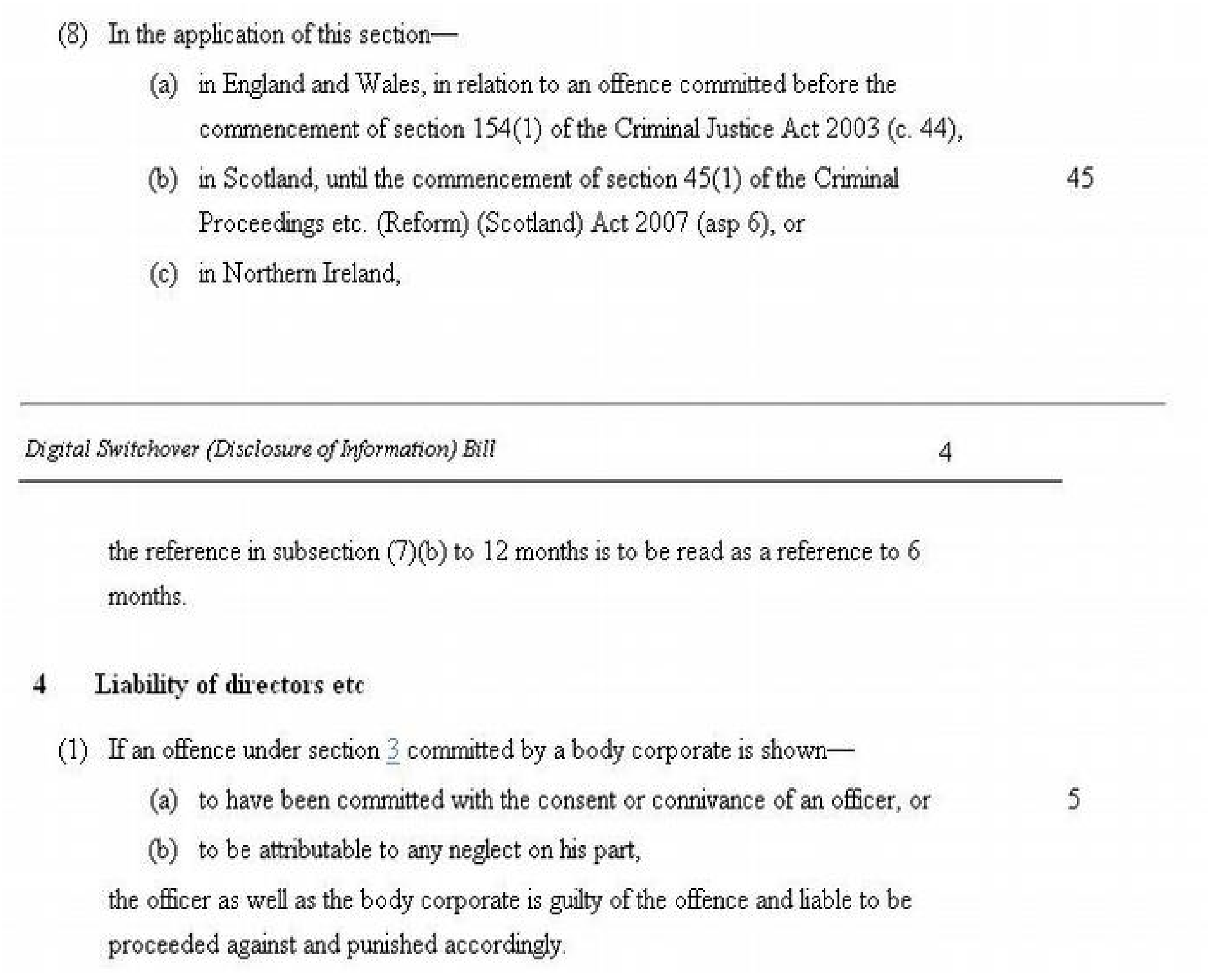
[F24(b) by reason only of such circumstances as are mentioned in [F25section 16(4) above, subsection (1) of section 3 of the Housing Act 1988](provisions where tenant B shares accommodation with persons other than landlord) did not have effect with respect to the separate accommodation,

the [F26said section 3]] shall during the remainder of the period of protection apply in relation to the separate accommodation as if the circumstances referred to in paragraph (b) of this subsection did not exist, and had not existed immediately before the ending of the tenancy [F27and, accordingly, as if on the ending of the tenancy there arose a statutory periodic tenancy which is an assured tenancy during the remainder of that period].

Figure 1. Amendments overlapping document structures

This kind of overlap occurs whenever annotations are made to a document that otherwise obeys a hierarchical structure. The most obvious examples are revisions, such as the legislation example above, and comments, which can overlap with both the main document structure and each other. This kind of overlap is very frequent in documents with multiple authors; during drafting; in long-lived documents; and so on.

Figure 2 shows some more legislation, this time a Bill going through the UK Parliament, available from the Parliamentary website (http://www.parliament.uk/business/bills_and_legislation.cfm). This is draft legislation, which is also available in printed form, and as such needs to be referenced by Members of Parliament in a consistent way: through page breaks and line numbers. In this example, the page break for page 4 overlaps item (c) in the list.



(8) In the application of this section—

(a) in England and Wales, in relation to an offence committed before the commencement of section 154(1) of the Criminal Justice Act 2003 (c. 44),

(b) in Scotland, until the commencement of section 45(1) of the Criminal Proceedings etc. (Reform) (Scotland) Act 2007 (asp 6), or 45

(c) in Northern Ireland,

Digital Switchover (Disclosure of Information) Bill 4

the reference in subsection (7)(b) to 12 months is to be read as a reference to 6 months.

4 Liability of directors etc

(1) If an offence under section 3 committed by a body corporate is shown—

(a) to have been committed with the consent or connivance of an officer, or 5

(b) to be attributable to any neglect on his part,

the officer as well as the body corporate is guilty of the offence and liable to be proceeded against and punished accordingly.

Figure 2. Pages and lines overlapping document structures

This example of overlap is moderately common in publishing. Publishers want to preserve information about page breaks in the XML created for their books, so that they can recreate paginated versions in eBook form and to facilitate searching. But a page can overlap practically anything: paragraphs, tables, lists and so on.

Finally, the hardcore of overlapping mark-up is where multiple hierarchies apply to the same document. The classic example is the Bible, which can be viewed as a Book/Chapter/Verse hierarchy or a more rhetorical Book/Section/Paragraph hierarchy. Multiple hierarchies also occur in the academic analysis of documents, particularly plays and poetry, which can contain physical, syntactic and rhythmic structures as well as commentary [BRUN06].

Representing Overlap

There are many ways to represent documents that contain overlapping structures. The most common methods in XML are to use processing instructions or empty elements as milestones, marking the start (and end) of the minor structures in the document.

For the purpose of discussion, I'll use the following short example that illustrates the three major classes of overlap outlined above:

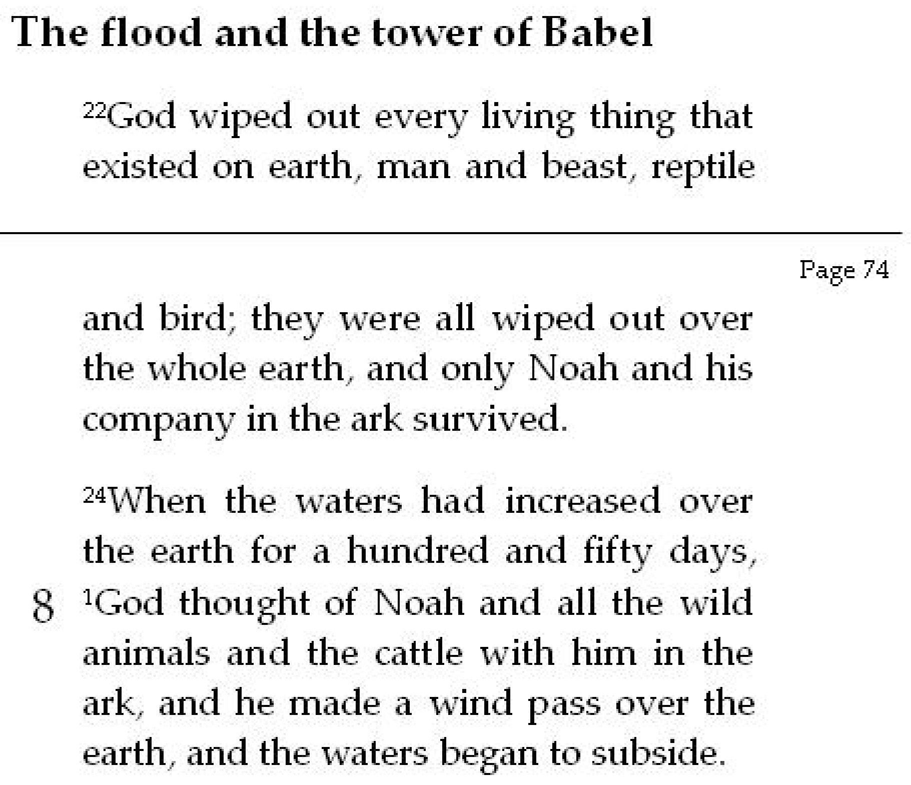
```
<book><?new-page no="1"?>
  <title>Genesis</title>
  ...
  <section>
    <heading>The flood and the tower of Babel</heading>
```

```
      ...
      <chapter no="7" />
      ...
      <para>
        ... <verse no="23" /><s>God wiped out every living thing
            that existed on earth, <index ref="i0037" mark="start" />
            man and <index ref="i0038" mark="start" />beast <index
            ref="i0037" mark="end" />, reptile <?new-page n="74"?>
            and bird<index ref="i0038" mark="end" />; they were all
            wiped out over the whole earth, and only Noah and his
            company in the ark survived.</s>
      </para>
      <para>
        <verse no="24" /><s>When the waters had increased over the
        earth for a hundred and fifty days, <chapter no="8" />
        <verse no="1" />God thought of Noah and all the wild
        animals and the cattle with him in the ark, and he made a
        wind pass over the earth, and the waters began to subside.
        </s>...
      </para>
      ...
    </section>
    ...
  </book>
```

This example contains:

- a section- and-paragraph-based hierarchy that is represented by normal elements
- a chapter-and-verse-based hierarchy that is represented by empty elements that indicate the start of new chapters and verses
- a set of pages, whose start (and end) are indicated by `<?new-page?>` processing instructions
- a set of index references, which can overlap with each other, which are represented by empty elements with `mark="start"` to indicate the start of a reference and `mark="end"` to indicate the end of a reference. The empty elements are linked to each other by sharing the same value for their `ref` attribute.

This example might be displayed as shown in Figure 3.


The flood and the tower of Babel

22God wiped out every living thing that existed on earth, man and beast, reptile

Page 74

and bird; they were all wiped out over the whole earth, and only Noah and his company in the ark survived.

24When the waters had increased over the earth for a hundred and fifty days,

8 1God thought of Noah and all the wild animals and the cattle with him in the ark, and he made a wind pass over the earth, and the waters began to subside.

Figure 3. Rendering of overlapping structures.

Using milestone processing instructions and elements isn't ideal. Structures that are indicated through milestones are less tangible than those that are indicated through proper elements: they are less amenable to querying and harder to process. For example, locating "the text referenced by index entry i0037", "the contents of page 74" or "Chapter 7, Verse 24" is much harder than identifying "the second paragraph of the section entitled 'The flood and the tower of Babel'".

In documents that contain multiple overlapping hierarchies, using milestones can give the erroneous impression that one hierarchy is more important or meaningful than another. In this example, sections and paragraphs form the primary document structure. But accessing biblical texts by chapter and verse is more common than via section, paragraph and sentence (which changes based on the translation), so the following markup, where the `<para>` and `<s>` elements are empty milestones, might make more sense:

```
<book><?new-page no="1"?>
  <title>Genesis</title>
  ...
  <section>
    <heading>The flood and the tower of Babel</heading>
  </section>
  ...
  <chapter no="7">
    ...
    <para />
    ... <verse no="23"><s />God wiped out every living thing that
        existed on earth, <index ref="i0037" mark="start" />man
        and <index ref="i0038" mark="start" /> beast<index
        ref="i0037" mark="end" />, reptile <?new-page n="74"?>and
        bird <index ref="i0038" mark="end" />; they were all
        wiped out over the whole earth, and only Noah and his
        company in the ark survived.</verse>
    <para />
    <verse no="24"><s />When the waters had increased over the
    earth for a hundred and fifty days, </verse>
  </chapter>
  <chapter no="8">
    <verse no="1">God thought of Noah and all the wild animals
    and the cattle with him in the ark, and he made a wind pass
    over the earth, and the waters began to subside.</verse>...
    ...
  </chapter>
  ...
</book>
```

As we'll see in the next sections, there are also many non-XML syntaxes for representing documents that contain overlap, such as TexMecs [HUIT06] and LMNL [TENN02].

Validating Overlap

So overlapping structures are common in real documents. Even using plain XML, we can represent them in a number of ways, but how do we validate them?

Milestones are hard to validate with existing schema languages. It's possible to indicate where elements are allowed, or even where processing instructions are permitted if you use Schematron, but any assertions about their content are hard to state.

For example, in the Bible extract above, we want to say:

- a book contains a title followed by one or more sections, each of which has a heading and a number of paragraphs, each of which contains a number of sentences
- a book also contains a title followed by one or more chapters, each of which contains a number of verses
- a book contains one or more pages
- page breaks cannot appear within the book title or a section heading
- every paragraph is made up of one or more verses
- a sentence or heading contains any number of index references, which can overlap with each other

When chapter and verse breaks are represented using milestones, we could use Relax NG to articulate these rules as follows:

```
start = book
book = element book { title, section+ }
title = element title { text }
section = element section { heading, para+ }
heading = element heading { text & index* }
para = element para { (chapter?, verse), s+ }
s = element s { text & (verse*, chapter?, verse*)
                    & index* }
chapter = element chapter { attribute no { text } }
verse = element verse { attribute no { text } }
index = element index { attribute ref { text },
                        attribute mark
                          { "start" | "end" } }
```

supplemented by some Schematron rules:

```
<sch:rule context="verse">
  <sch:assert test="preceding::chapter">
```

```
    Each verse milestone must come after a chapter milestone.
  </sch:assert>
</sch:rule>
<sch:rule context="s | heading">
  <sch:report test="index[@mark = 'start' and
                         not(@ref =
                             current()/index
                               [@mark = 'end']/@ref)]">
    Every index start anchor must be matched by an end anchor in
    the same sentence or heading.
  </sch:report>
</sch:rule>
<sch:rule context="processing-instruction('new-page')">
  <sch:report test="parent::heading or parent::title">
    Pages cannot start within a section heading or the book
    title.
  </sch:report>
</sch:rule>
```

These rules don't allow us to make assertions in an intuitive way about "chapters" or "pages" because such structures aren't easy to identify within the document. They also force us to constrain the markup language in unnatural ways, such as insisting that the first `<verse>` milestone must be the first child of a `<para>`.

Concur

SGML supported overlap through the CONCUR directive. With CONCUR, a document can have several DTDs associated with it, and each element within the document is associated with one or more of the DTDs. The document is validated against each of the DTDs, but only those elements associated with a particular DTD are visible during validation.

Our example using SGML's CONCUR is shown below.

```
<!DOCTYPE book.s SYSTEM "book.sections.dtd">
<!DOCTYPE book.c SYSTEM "book.chapters.dtd">
<!DOCTYPE book.p SYSTEM "book.pages.dtd">
<!DOCTYPE book.i1 SYSTEM "book.index.dtd">
<!DOCTYPE book.i2 SYSTEM "book.index.dtd">
<(book.s)book.s><(book.c)book.c><(book.p)book.p>
<(book.i1)book.i1><(book.i2)book.i2>
  <(book.p)page no="1">
  <(book.s)title>
    <(book.c)title>Genesis</(book.c)title>
  </(book.s)title>
  ...
  <(book.s)section
    heading="The flood and the tower of Babel">
  ...
  <(book.c)chapter no="7">
    ...
    <(book.s)para>...<(book.s)s><(book.i1)s><(book.i2)s>
      <(book.c)verse no="23"> God wiped out every living thing
      that existed on earth, <(book.i1)index ref="i0037"> man
      and <(book.i2)index ref="i0038">beast</(book.i1)index>,
      reptile</(book.p)page><(book.p)page no="74">and bird
      </(book.i2)index>; they were all wiped out over the whole
      earth, and only Noah and his company in the ark survived.
      </(book.c)verse> </(book.i2)s></(book.i1)s></(book.s)s>
    </(book.s)para>
    <(book.s)para><(book.s)s><(book.c)verse no="24">When the
    waters had increased over the earth for a hundred and fifty
    days, </(book.c)verse>
  </(book.c)chapter>
  <(book.c)chapter no="8">
    <(book.c)verse no="1">God thought of Noah and all the wild
    animals and the cattle with him in the ark, and he made a
    wind pass over the earth, and the waters began to subside.
    </(book.c)verse></(book.s)s>...</(book.s)para>
    ...
  </(book.c)chapter>
  ...
  </(book.s)section>
  ...
  </(book.p)page>
</(book.i2)book.i2></(book.i1)book.i1></(book.p)book.p>
</(book.s)book.s></(book.c)book.c>
```

Documents using CONCUR are treated as multiple separate documents for the purpose of validation. To validate the document against a particular DTD, all tags that don't belong to that DTD are stripped out and the result validated. The document as a whole is valid if it's valid against all the DTDs it defines. In this case, the five documents are section-based:

```
  <!DOCTYPE book.s SYSTEM "book.sections.dtd">
  <book.s>
    <title>Genesis</title>
    ...
    <section heading="The flood and the tower of Babel">
      ...
      <para>...<s>God wiped out every living thing that existed on
      earth, man and beast, reptile and bird; they were all wiped
      out over the whole earth, and only Noah and his company in
      the ark survived.</s></para>
      <para><s>When the waters had increased over the earth for a
      hundred and fifty days, God thought of Noah and all the
      wild animals and the cattle with him in the ark, and he
      made a wind pass over the earth, and the waters began to
      subside.</s>...</para>
      ...
    </section>
    ...
  </book.s>
```

chapter-based:

```
  <!DOCTYPE book.c SYSTEM "book.chapters.dtd">
  <book.c>
    <title>Genesis</title>
    ...
    <chapter no="7">
      ...
      <verse no="23">God wiped out every living thing that existed
      on earth, man and beast, reptile and bird; they were all
      wiped out over the whole earth, and only Noah and his company
      in the ark survived.</verse>
      <verse no="24">When the waters had increased over the earth
      for a hundred and fifty days, </verse>
    </chapter>
    <chapter no="8">
      <verse no="1">God thought of Noah and all the wild animals
      and the cattle with him in the ark, and he made a wind pass
      over the earth, and the waters began to subside.</verse>
      ...
    </chapter>
    ...
  </book.c>
```

page-based:

```
<!DOCTYPE book.p SYSTEM "book.pages.dtd">
<book.p>
  <page no="1">
    Genesis
    ...
    God wiped out every living thing that existed on earth, man
    and beast, reptile
  </page>
  <page no="74">
    and bird; they were all wiped out over the whole earth, and
    only Noah and his company in the ark survived.

    When the waters had increased over the earth for a hundred
    and fifty days, God thought of Noah and all the wild animals
    and the cattle with him in the ark, and he made a wind pass
    over the earth, and the waters began to subside.
    ...
  </page>
</book.p>
```

and there are two index-based documents, one for the first index term:

```
<!DOCTYPE book.i1 SYSTEM "book.index.dtd">
<book.i1>
  Genesis
  ...
  <s>God wiped out every living thing that existed on earth,
  <index ref="i0037">man and beast</index>, reptile and bird;
  they were all wiped out over the whole earth, and only Noah and
  his company in the ark survived.</s>

  When the waters had increased over the earth for a hundred and
  fifty days, God thought of Noah and all the wild animals and
  the cattle with him in the ark, and he made a wind pass over
  the earth, and the waters began to subside.
  ...
</book.i1>
```

and one for the second index term:

```
<!DOCTYPE book.i2 SYSTEM "book.index.dtd">
<book.i2>
```

```
    Genesis
    ...
    <s>God wiped out every living thing that existed on earth, man
    and <index ref="i0038">beast, reptile and bird</index>; they
    were all wiped out over the whole earth, and only Noah and his
    company in the ark survived.</s>

    When the waters had increased over the earth for a hundred and
    fifty days, God thought of Noah and all the wild animals and
    the cattle with him in the ark, and he made a wind pass over
    the earth, and the waters began to subside.
    ...
  </book.i2>
```

There are several problems with the CONCUR approach:

- CONCUR requires extensive, and verbose, markup of the document. Not only does this add to the size of the document, it also strongly binds the document closely to its DTDs.
- CONCUR does not allow co-constraints between hierarchies to be expressed, such as the fact that each paragraph contains one or more verses.
- All the text within the document must be covered by all hierarchies in the document. In the example above, I had to move the section heading to be an attribute so that the chapter-based hierarchy would validate.
- Elements that are common in several hierarchies must be repeated so that they appear in all the documents to be validated. The `<book>` and `<title>` elements are examples in the above.
- CONCUR does not extend well to self-overlap, where a single element can overlap with itself, because this would require an arbitrary number of DTDs to be associated with the document. In this example, because the `<index>` elements overlapped each other, I had to have two sub-documents. The more self-overlap there is, the more DTDs are required.

Rabbit/Duck Grammars

In [SPER06], Sperberg-McQueen proposed a solution similar to CONCUR, named rabbit/duck grammars. Like CONCUR, rabbit/duck grammars split a document into several virtual documents, each of which can be validated against a different grammar. But for the purpose of validation, rabbit/duck grammars classify elements into four categories:

- Normal elements
- Milestones, whose start and end tags can be matched individually
- Transparent elements, whose start and end tags are ignored
- Opaque elements, whose start and end tags, and content, are ignored completely

CONCUR treats all elements as either normal elements or transparent elements. In rabbit/duck grammars, milestones add the ability to do cross-grammar validation by limiting where the start and end tags of elements from other grammars can appear, and opaque elements mean that subtrees can be ignored.

Rabbit/duck grammars start from a document in which all elements are marked up in the same way, such as TexMecs [HUIT06]:

```
  <book|<page no="1"|
    <title|Genesis|title>
    ...
    <section|
      <heading|The flood and the tower of Babel|heading>
      ...
      <chapter no="7"|
        ...
      <para|...<s|<verse no="23"|God wiped out every living thing
      that existed on earth, <index~1 ref="i0037"|man and
      <index~2 ref="i0038"|beast|index~1>, reptile|page>
      <page no="74"|and bird|index~2>; they were all wiped out
      over the whole earth, and only Noah and his company in the
      ark survived.|s>|verse>|para>
      <para|<verse no="24"|<s|When the waters had increased over
      the earth for a hundred and fifty days, |verse>|chapter>
      <chapter no="8"|<verse no="1"|God thought of Noah and all
      the wild animals and the cattle with him in the ark, and
      he made a wind pass over the earth, and the waters began to
      subside.|verse>|s>...|para>
      ...
    |chapter>
    ...
  |section>
  ...
|page>|book>
```

The grammar for the chapter-based view is:

```
<!-- Chapter grammar -->
<!GRAMMAR book [
  <!ELEMENT book (title, chapter+)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT chapter (verse+)>
  <!ATTLIST chapter
    no NMTOKEN #REQUIRED>
  <!ELEMENT verse (#PCDATA)>
  <!ATTLIST verse
    no NMTOKEN #REQUIRED>
  <!ELEMENT heading IGNORE>
]>
```

Here, the `<book>`, `<title>`, `<chapter>` and `<verse>` elements are normal elements. There are no milestone elements. The `<page>`, `<section>`, `<para>` and `<index>` elements are transparent (they're not mentioned in the DTD so their tags are ignored), and the `<heading>` element is opaque (it and its content are ignored completely). In XML, the document to be validated would look like:

```
<book>
  <title>Genesis</title>
  ...
  <chapter no="7">
    ...
    <verse no="23">God wiped out every living thing that existed
    on earth, man and beast, reptile and bird; they were all
    wiped out over the whole earth, and only Noah and his company
    in the ark survived.</verse>
    <verse no="24">When the waters had increased over the earth
    for a hundred and fifty days, </verse>
  </chapter>
  <chapter no="8">
    <verse no="1">God thought of Noah and all the wild animals
    and the cattle with him in the ark, and he made a wind pass
    over the earth, and the waters began to subside.</verse>
    ...
  </chapter>
  ...
</book>
```

A grammar for the section-based view of the biblical structure could be:

```
<!-- Section grammar -->
<!GRAMMAR book [
  <!ELEMENT book (title, section+)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT section (heading, para+>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT para (#stag(verse),
                  (s+ & #tag(verse)*),
                  #etag(verse))>
  <!ELEMENT s (#PCDATA | #tag(verse))*>
]>
```

Under this view, the `<book>`, `<title>`, `<section>`, `<heading>`, `<para>` and `<s>` elements are normal elements: they have element declarations. The `<verse>` tags are classed as milestones: start tags are matched by `#stag(verse)`, end tags by `#etag(verse)` and any tag matched by the particle `#tag(verse)`. The `<chapter>` and `<index>` elements are transparent, and thus their tags are ignored. In XML, the document to be validated would look like the following (`sID` and `eID` attributes have been added to milestone elements to indicate the correct pairings).

```
<book>
  <title>Genesis<title>
  ...
  <section>
    <heading>The flood and the tower of Babel<heading>
    ...
    <para>...<s><verse no="23" sID="c7v23" />God wiped out every
    living thing that existed on earth, man and beast, reptile
    and bird; they were all wiped out over the whole earth, and
    only Noah and his company in the ark survived.</s><verse
    eID="c7v23" />
    </para>
    <para><verse no="24" sID="c7v24" /><s>When the waters had
    increased over the earth for a hundred and fifty days, <verse
    eID="c7v24" /><verse no="1" sID="c8v1">God thought of Noah
    and all the wild animals and the cattle with him in the ark,
    and he made a wind pass over the earth, and the waters began
    to subside.</s><verse eID="c8v1" />...</para>
    ...
  </section>
  ...
</book>
```

Using milestones for the `<verse>` elements means we can constrain how the two hierarchies interact. The declaration for the `<para>` element:

```
<!ELEMENT para (#stag(verse),
                (s+ & #tag(verse)*),
                #etag(verse))>
```

says that they must start with a start tag of a verse, then contain interleaved `<s>` elements and verse start or end tags, and end with the end tag of a verse. Because we know (from the chapter-based hierarchy) that verses can't nest or overlap each other, these constraints guarantee that each `<para>` contains a whole number of verses.

But the declaration for the `<para>` element also adds a constraint about the position of the verse tags: the first thing in the `<para>` element must be the start tag for a `<verse>` element. The following paragraph wouldn't be valid because the first verse start tag appears within the `<s>` element.

```
<para|<s|<verse no="24"|When the waters had increased over the
earth for a hundred and fifty days, |verse>|chapter><chapter
no="8"|<verse no="1"|God thought of Noah and all the wild animals
and the cattle with him in the ark, and he made a wind pass over
the earth, and the waters began to subside.|verse>|s>...|para>
```

A page-based grammar could similarly constrain the position of page breaks by viewing headings and titles as normal elements:

```
<!-- Page grammar -->
<!GRAMMAR book [
  <!ELEMENT book (page+)>
  <!ELEMENT page (#PCDATA | title | heading)*>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
]>
```

to give:

```
<book>
  <page no="1">
    <title>Genesis<title>
    ...
    <heading>The flood and the tower of Babel<heading>
    ...
    God wiped out every living thing that existed on earth, man
    and beast, reptile
  </page>
  <page no="74">
    and bird; they were all wiped out over the whole earth, and
    only Noah and his company in the ark survived.

    When the waters had increased over the earth for a hundred
    and fifty days, God thought of Noah and all the wild animals
    and the cattle with him in the ark, and he made a wind pass
    over the earth, and the waters began to subside.
    ...
  </page>
</book>
```

Finally, to handle the self-overlapping index references, rabbit/duck grammars introduce a new kind of group called an overlap group (`#overlap()`) and an overlap token (`~`*`name`*`~`) which can only appear within an overlap group and matches an instance of an element that overlaps itself. Here, to constrain where index references can appear we could use a grammar that treats paragraphs and headings as normal elements that contain overlap groups:

```
<!-- Index grammar -->
<!GRAMMAR book [
  <!ELEMENT book (heading, s+)+>
  <!ELEMENT heading (#overlap( (#PCDATA | #tag(index))*,
                               ~index~,
                               (#PCDATA | #tag(index))* )>
  <!ELEMENT s (#overlap( (#PCDATA | #tag(index))*,
                         ~index~,
                         (#PCDATA | #tag(index))* )>
  <!ELEMENT index (#PCDATA | #tag(index))*>
  <!ATTLIST index
    ref  NMTOKEN  #REQUIRED>
  <!ELEMENT title IGNORE>
]>
```

To validate these structures, each portion of a document that matches an overlap group is read multiple times. Each time, a different element matching the overlap token is interpreted as a normal element; all other instances of that element are treated as milestones. For example, the sentence

```
<s|...God wiped out every living thing that existed on earth,
<index~1 ref="i0037"|man and <index~2 ref="i0038"|beast|index~1>,
reptile and bird|index~2>; they were all wiped out over the whole
earth, and only Noah and his company in the ark survived.|s>
```

contains two index references. The first view of the sentence would yield:

```
<s>...God wiped out every living thing that existed on earth,
<index ref="i0037">man and <index ref="i0038" sID="e1" />beast
</index>, reptile and bird<index eID="e1" />; they were all wiped
out over the whole earth, and only Noah and his company in the
ark survived.</s>
```

The second view would yield:

```
<s>...God wiped out every living thing that existed on earth,
<index ref="i0037" sID="e1" />man and <index ref="i0038">beast
<index eID="e1" />, reptile and bird</index>; they were all wiped
out over the whole earth, and only Noah and his company in the
ark survived.</s>
```

Rabbit/duck grammars have a lot going for them. They provide a much more flexible method of validating overlapping markup than CONCUR, and they can handle self-overlap. However, it can be hard to predict the impact of interacting grammars on the actual structure of a document, and the limits that tag-based particles place on documents can seem quite arbitrary. In addition, the method of expressing self-overlap is quite verbose even in simple cases.

XCONCUR

XCONCUR [HIL05, SCH06] uses a combination of a similar mechanism to CONCUR and a constraint-based language similar to Schematron in order to validate multi-hierarchy documents.

As in CONCUR, each element in the document must be assigned to an "annotation layer", for which a grammar is supplied. To be valid overall, the document must be valid on each annotation layer.

The constraint-based language is of most interest here, because it differs from the previous approaches. XCONCUR provides three particles:

- `start(`*`layer`*`, `*`name`*`)` indicates the start tag of an element called *name* in the layer *layer*
- `end(`*`layer`*`, `*`name`*`)` indicates the end tag of an element called *name* in the layer *layer*
- `element(`*`layer`*`, `*`name`*`)` indicates the element called *name* in the layer *layer*

and six operators:

- `tag1 << tag2` means that `tag1` precedes `tag2`
- `tag1 == tag2` means that `tag1` and `tag2` appear at the same position within the text of the document
- `tag1 ][ el2` means that `tag1` is not within `el2`
- `tag1 >> tag2` means that `tag1` follows `tag2`
- `tag1 [] el2` means that `tag1` is within `el2`
- `tag1 <= tag2` means that `tag1` either appears before or at the same place as `tag2` within the text of the document

These operators are sufficient to express relationships between ranges. For example, to say that each paragraph must start a new verse, you could use:

```
stag(book.s, para) == stag(book.c, verse)
```

although presumably this constraint would also mean that all verses must coincide with the start of a paragraph, which is not desired.

XCONCUR usefully distinguishes the method of defining annotation layers from the method used to validate them, but articulating the cross-layer constraints separately is awkward.

Creole

The validation technologies that we've looked at so far are based on the concept of separating a single document into separate documents, each of which is validated against a separate DTD. The underlying assumption is that a single document has been marked up separately by distinct languages and the results merged. This can be compared to code-switching in natural language:

> **Code-switching** is a term in linguistics referring to alternation between two or more languages, dialects, or language registers in a single conversation, stretch of discourse, or utterance between people who have more than one language in common. (Wikipedia)

The biggest problem with this approach is that it can be hard to get an overview about what the intersection of the separate grammars actually means for a single document.

Creole takes a different tack. It views documents in which elements overlap as conforming to a unified markup language in its own right that allows overlap. This is similar to a natural language creole:

> A **creole language**, or simply a **creole**, is a well-defined and stable language that originated from a non-trivial combination of two or more languages, typically with many distinctive features that are not inherited from either parent. (Wikipedia)

Creole stands for Composable Regular Expressions for Overlapping Languages etc. It takes Relax NG as its basis, rather than DTDs, which gives it more power and flexibility than CONCUR and rabbit/duck grammars. It's also based on the LMNL (Layered Markup and Annotation Language) data model, which views elements as named and annotated ranges of text within a character stream (see www.lmnlwiki.org for details).

In this section, we'll look at Creole's syntax and in the next section an algorithm that can be used to validate documents with it. For simplicity, this summary will omit new patterns that match the LMNL-specific constructs of annotations and atoms.

Everything that's allowed in Relax NG is allowed in Creole, and every Relax NG schema is a valid Creole schema. (If you're unfamiliar with Relax NG, I recommend the tutorial at http://www.oasis-open.org/committees/relax-ng/tutorial.html.)

The first pattern that Creole adds to Relax NG is the `<range>` pattern. The `<range>` pattern is identical in syntax to the `<element>` pattern, and matches a named range within a document, which can overlap other ranges. For example:

```
<range name="page">
  <attribute name="no" />
  <text />
</range>
```

matches a range named `page` with an attribute called `no`. Attribute patterns in Creole are treated as a shorthand for an `<annotation>` pattern, since LMNL has structured and annotatable *annotations* rather than attributes. The pattern can be used to match annotations that have neither structure nor annotations.

Element patterns are still allowed within Creole, but they are treated as a syntactic shorthand for a `<partition>` pattern wrapped around a `<range>` pattern. The new `<partition>` pattern says that this part of the document can't be overlapped by anything else. We can define the title of the book as an element, with:

```
<element name="title"><text /></element>
```

which is shorthand for:

```
<partition>
  <range name="title"><text /></range>
<partition>
```

and nothing will be able to overlap it.

In sequences (a `<group>` pattern), ranges behave like elements: the ranges must appear in that order and cannot overlap. However, if you use `<interleave>` then ranges may overlap. For example,

```
<interleave>
  <text />
  <range name="strong"><text /></range>
  <range name="em"><text /></range>
</interleave>
```

allows the `<strong>` and `<em>` elements to overlap each other, or appear inside each other. To explain this, it helps to imagine each range pattern expanded to a start tag, the content allowed in the range, and an end tag. Although not legal Creole syntax, expanding the above pattern would give

```
<interleave>
  <text />
  <group>
    <startTag name="strong" />
    <text />
    <endTag name="strong" />
  </group>
  <group>
    <startTag name="em" />
    <text />
    <endTag name="em" />
  </group>
</interleave>
```

Partitions act as a strongly bound group which cannot be interrupted once it is started. It's as if the ranges within a partition aren't expanded as described above.

The third new pattern is `<concur>`. The `<concur>` pattern is like `<group>` and `<interleave>`, in that it takes two or more sub-patterns. Both patterns must be matched, but while any content (text) must be matched in *both* patterns, markup can be matched by either or both. Put another way, `<concur>` describes one or more distinct grammars much as CONCUR does, but applies them to a subtree rather than the entire document.

To see `<concur>` in action, let's look at the content of the `<para>` range in the Biblical example, as defined by Creole:

```
<range name="para">
  <concur>
    <oneOrMore>
      <ref name="verse" />
    </oneOrMore>
    <oneOrMore>
      <ref name="s" />
    </oneOrMore>
  </concur>
</range>
```

The `<para>` range can be described as containing one or more `<verse>` ranges. It can also be described as containing one or more `<s>` ranges. These two content models run concurrently; all the textual content of the `<para>` range must appear both within a `<verse>` and within a `<s>`.

It's worth noting that, unlike the equivalent rabbit/duck grammar, in which the first thing in the `<para>` element had to be start tag of a `<verse>` element, this definition of the `<para>` range allows the tags for the `<verse>` and `<s>` ranges to appear in any order.

The `<book>` element in our example has similar concurrent content. It can be described as having one or more `<page>` ranges. It can also be described as having a `<title>` followed by (concurrently) one or more `<chapter>` ranges and one or more `<section>` ranges. The pattern for the `<book>` element looks like:

```
<element name="book">
  <concur>
    <oneOrMore>
      <ref name="page" />
    </oneOrMore>
    <group>
      <ref name="title" />
      <concur>
        <oneOrMore>
          <ref name="chapter" />
        </oneOrMore>
        <oneOrMore>
          <ref name="section" />
        </oneOrMore>
      </concur>
    </group>
  </concur>
</element>
```

What about the `<heading>` element within the `<section>`? Its text isn't inside a `<verse>`, so would normally be invalid. However, we can define the `<heading>` as a partition — it can't overlap anything else — and this means that it's ignored by any concurrent patterns. The definition of `<heading>` is:

```
<element name="heading">
  <ref name="indexedText" />
</element>
```

which expands to

```
<partition>
  <range name="heading">
    <ref name="indexedText" />
  </range>
</partition>
```

The final patterns to introduce are the `<concurOneOrMore>` and `<concurZeroOrMore>` patterns, which support self-overlap. These patterns are similar to `<oneOrMore>` and `<zeroOrMore>` except that rather than the sub-pattern matching several times in sequence, the sub-pattern matches several times concurrently. In the Biblical example, the `<index>` elements overlap with themselves. The pattern for the `<s>` range is:

```
<range name="s">
  <concurOneOrMore>
    <mixed>
      <zeroOrMore>
        <ref name="index" />
      </zeroOrMore>
    </mixed>
  </concurOneOrMore>
</range>
```

The `<s>` range has mixed content. At its simplest, it can contain text with zero or more `<index>` ranges appearing in sequence. But that same content model can appear multiple times concurrently, with `<index>` ranges overlapping other `<index>` ranges.

A compact version of the complete Creole grammar for the Biblical example we've been looking at is

```
start = book
book = element book { page ~
                      ( title, ( chapter+ ~ section+ ) ) }
page = range page { attribute no { text }, text }
title = element title { text }
chapter = range chapter { attribute no { text }, verse+ }
verse = range verse { attribute no { text }, text }
section = range section { heading, para+ }
heading = element heading { indexedText }
para = range para { verse+ ~ s+ }
s = range s { indexedText }
indexedText = concurOneOrMore { mixed { index* } }
index = range index { attribute ref { text }, text }
```

The ~ is used to create `<concur>` pattern, and `concurOneOrMore{...}` creates a `<concurOneOrMore>` pattern. The full syntax version is in the Appendix.

Validating with Creole

Creole can be implemented using a similar algorithm as that used for Relax NG [CLA02], based on Brzozowski derivatives. Brzozowski derivatives have also been used successfully with XML Schema validation [SPER05] and rabbit/duck grammars [SPER06].

Algorithms based on Brzozowski derivatives start with a pattern and a sequence of tokens that might or might not match the pattern. Using string regular expressions, say we had the pattern

```
a(b|c)d+
```

and the tokens (characters)

```
acd
```

We look at the first token and construct a new pattern, based on the old one, that we can use to test the rest of the tokens. This new pattern is known as the *derivative*. In this case, the derivative from the first character, `'a'`, is the regular expression

```
(b|c)d+
```

Then we take the second token and create a new derivative from that. In this example, the new derivative after the character `'c'` is

```
d+
```

We keep doing this until we get to the end of the sequence of tokens. If the final regular expression is empty or is nullable (which means it can match an empty sequence of tokens), then the string matches the regular expression. If not, it doesn't. For example, if we started with the string

```
bd
```

then the first derivative would be a regular expression that cannot match any string.

This algorithm for Creole works in largely the same way as that for string regular expressions: the tokens are tags and text events rather than characters, and the patterns are different, but the principle of constructing derivatives still applies. It builds on the algorithm developed by James Clark for Relax NG [CLA02].

In this section, we'll go through the basic algorithm for Creole, but omit those parts that deal with annotations (attributes), atoms and datatypes. The full algorithm is available at http://www.lmnlwiki.org/index.php/Algorithm_for_Creole_Validation. I'll use the Haskell-based syntax used for the Relax NG algorithm as it's slightly more approachable than mathematical notation.

Basics

We need basic datatypes for URIs, local names and IDs:

```
type Uri = String
type LocalName = String
type Id = String
```

Qualified names are represented by the `QName` datatype, which consists of a URI and a local name:

```
data QName = QName Uri LocalName
```

We need to define the different kinds of name classes that are allowed (these are the same as those allowed in Relax NG):

```
data NameClass = AnyName
               | AnyNameExcept NameClass
               | Name Uri LocalName
               | NsName Uri
               | NsNameExcept Uri NameClass
               | NameClassChoice NameClass NameClass
```

We also need to define the different kinds of patterns that are supported by Creole. This is an different set from those allowed in Relax NG: ignoring those to do with annotations/attributes, atoms and datatypes, `Concur`, `Partition`, `ConcurOneOrMore`, `Range`, `EndRange` and `All` are new, and `Element` is no longer present.

```
data Pattern = Empty
               | NotAllowed
               | Text
               | Choice Pattern Pattern
               | Interleave Pattern Pattern
               | Group Pattern Pattern
               | Concur Pattern Pattern
               | Partition Pattern
               | OneOrMore Pattern
               | ConcurOneOrMore Pattern
               | Range NameClass Pattern
               | EndRange QName Id
               | After Pattern Pattern
               | All Pattern Pattern
```

The patterns `EndRange`, `After` and `All` are constructs that are used in the algorithm but don't correspond to patterns you can write in a Creole grammar.

- The `EndRange` pattern matches the end tag of a range with the given qualified name and ID. The ID is used for matching self-overlapping ranges.
- The `After` pattern is used when creating partitions; in the algorithm for Relax NG it guarantees that interleaved elements can't appear within each other. It's a bit like `Group`, in that the first sub-pattern of `After` must be completed before the second can be matched, but it's handled differently when patterns are composed, so that the `After` pattern "bubbles up" through groups.
- The `All` pattern means that both sub-patterns must be matched by the events1.

For validating against Creole, a document is considered to be a sequence of events; the only ones we're going to look at here are start tag events, end tag events and text events. Note that, unlike in normal XML, start and end tags can have IDs, which allows us to match the tags of self-overlapping ranges.

```
data Event = StartTagEvent QName Id
             | EndTagEvent QName Id
             | TextEvent String Context
```

Utilities

The following utility functions are used in the algorithm.

The most important utility function is `nullable`, which tests whether a given pattern can match an empty sequence of events. Nullable is defined as follows for the various kinds of patterns:

```
nullable:: Pattern -> Bool
nullable Empty = True
nullable NotAllowed = False
nullable Text = True
nullable (Choice p1 p2) = nullable p1 || nullable p2
nullable (Interleave p1 p2) = nullable p1 && nullable p2
nullable (Group p1 p2) = nullable p1 && nullable p2
nullable (Concur p1 p2) = nullable p1 && nullable p2
nullable (Partition p) = nullable p
nullable (OneOrMore p) = nullable p
nullable (ConcurOneOrMore p) = nullable p
nullable (Range _ _) = False
nullable (EndRange _ _) = False
nullable (After _ _) = False
nullable (All p1 p2) = nullable p1 && nullable p2
```

The second utility function is allowsText, which returns true if the pattern can match text. This is important because whitespace-only text events are ignored if text isn't allowed by a pattern.

```
allowsText:: Pattern -> Bool
allowsText (Choice p1 p2) = allowsText p1 || allowsText p2
allowsText (Group p1 p2) =
  if nullable p1 then (allowsText p1 || allowsText p2)
                 else allowsText p1
allowsText (Interleave p1 p2) =
  allowsText p1 || allowsText p2
allowsText (Concur p1 p2) = allowsText p1 && allowsText p2
allowsText (Partition p) = allowsText p
allowsText (OneOrMore p) = allowsText p
allowsText (ConcurOneOrMore p) = allowsText p
allowsText (After p1 p2) =
  if nullable p1 then (allowsText p1 || allowsText p2)
                 else allowsText p1
allowsText (All p1 p2) = allowsText p1 && allowsText p2
allowsText Text = True
allowsText _ = False
```

Finally, like Relax NG, Creole needs a method of testing whether a given qualified name matches a given name class:

```
contains :: NameClass -> QName -> Bool
contains AnyName _ = True
contains (AnyNameExcept nc) n = not (contains nc n)
contains (NsName ns1) (QName ns2 _) = (ns1 == ns2)
contains (NsNameExcept ns1 nc) (QName ns2 ln) =
  ns1 == ns2 && not (contains nc (QName ns2 ln))
contains (Name ns1 ln1) (QName ns2 ln2) =
  (ns1 == ns2) && (ln1 == ln2)
contains (NameClassChoice nc1 nc2) n =
  (contains nc1 n) || (contains nc2 n)
```

Constructors

When we create a derivative, we often need to create a new pattern. These constructors take into account special handling of NotAllowed, Empty and After patterns.

```
choice :: Pattern -> Pattern -> Pattern
choice p NotAllowed = p
choice NotAllowed p = p
choice Empty Empty = Empty
choice p1 p2 = Choice p1 p2
group :: Pattern -> Pattern -> Pattern
group p NotAllowed = NotAllowed
group NotAllowed p = NotAllowed
group p Empty = p
group Empty p = p
group (After p1 p2) p3 = after p1 (group p2 p3)
group p1 (After p2 p3) = after p2 (group p1 p3)
group p1 p2 = Group p1 p2
interleave :: Pattern -> Pattern -> Pattern
interleave p NotAllowed = NotAllowed
interleave NotAllowed p = NotAllowed
interleave p Empty = p
interleave Empty p = p
interleave (After p1 p2) p3 = after p1 (interleave p2 p3)
interleave p1 (After p2 p3) = after p2 (interleave p1 p3)
interleave p1 p2 = Interleave p1 p2
concur :: Pattern -> Pattern -> Pattern
concur p NotAllowed = NotAllowed
concur NotAllowed p = NotAllowed
concur p Text = p
concur Text p = p
concur (After p1 p2) (After p3 p4) =
  after (all p1 p3) (concur p2 p4)
concur (After p1 p2) p3 = after p1 (concur p2 p3)
concur p1 (After p2 p3) = after p2 (concur p1 p3)
concur p1 p2 = Concur p1 p2
partition :: Pattern -> Pattern
partition NotAllowed = NotAllowed
partition Empty = Empty
partition p = Partition p
oneOrMore :: Pattern -> Pattern
oneOrMore NotAllowed = NotAllowed
oneOrMore Empty = Empty
oneOrMore p = OneOrMore p
concurOneOrMore :: Pattern -> Pattern
concurOneOrMore NotAllowed = NotAllowed
concurOneOrMore Empty = Empty
concurOneOrMore p = ConcurOneOrMore p
after :: Pattern -> Pattern -> Pattern
after p NotAllowed = NotAllowed
after NotAllowed p = NotAllowed
after Empty p = p
after (After p1 p2) p3 = after p1 (after p2 p3)
after p1 p2 = After p1 p2
all :: Pattern -> Pattern -> Pattern
all p NotAllowed = NotAllowed
all NotAllowed p = NotAllowed
all p Empty = if nullable p then Empty else NotAllowed
all Empty p = if nullable p then Empty else NotAllowed
all (After p1 p2) (After p3 p4) =
  after (all p1 p3) (all p2 p4)
all p1 p2 = All p1 p2
```

Derivatives

Finally, we can look at the calculation of derivatives. A document is a sequence of events. The derivative for a sequence of events against a pattern is the derivative of the remaining events against the derivative of the first event.

```
eventsDeriv :: Pattern -> [Event] -> Pattern
eventsDeriv p [] = p
eventsDeriv p (h:t) = eventsDeriv (eventDeriv p h) t
```

The derivative for an event depends on the kind of event, and we use different functions for each kind. Whitespace-only text nodes can be ignored if the pattern doesn't allow text.

```
eventDeriv :: Pattern -> Event -> Pattern
eventDeriv p (TextEvent s cx) =
  if (whitespace s && not allowsText p)
  then p
  else (textDeriv cx p s)
eventDeriv p (StartTagEvent qn id) = startTagDeriv p qn id
eventDeriv p (EndTagEvent qn id) = endTagDeriv p qn id
```

Text Derivatives

`textDeriv` computes the derivative of a pattern with respect to a text event.

```
textDeriv :: Context -> Pattern -> String -> Pattern
```

For `Choice`, `Group`, `Interleave` and the other standard Relax NG patterns, the derivative is just the same as in Relax NG:

```
textDeriv cx (Choice p1 p2) s =
  choice (textDeriv cx p1 s) (textDeriv cx p2 s)
textDeriv cx (Interleave p1 p2) s =
  choice (interleave (textDeriv cx p1 s) p2)
         (interleave p1 (textDeriv cx p2 s))
textDeriv cx (Group p1 p2) s =
  let p = group (textDeriv cx p1 s) p2
  in if nullable p1 then choice p (textDeriv cx p2 s)
                    else p
textDeriv cx (After p1 p2) s =
  after (textDeriv cx p1 s) p2
textDeriv cx (OneOrMore p) s =
  group (textDeriv cx p s) (choice (OneOrMore p) Empty)
textDeriv cx Text _ = Text
```

For `Concur`, text is only allowed if it is allowed by both of the sub-patterns: we create a new `Concur` whose sub-patterns are the derivatives of the original sub-patterns.

```
textDeriv cx (Concur p1 p2) s =
  concur (textDeriv cx p1 s)
         (textDeriv cx p2 s)
```

For `ConcurOneOrMore`, we partially expand the `ConcurOneOrMore` into a `Concur`. This mirrors the derivative for `OneOrMore`, except that a new `Concur` pattern is constructed rather than a `Group`, and the second sub-pattern is a choice between a `ConcurOneOrMore` and `Text`.

```
textDeriv cx (ConcurOneOrMore p) s =
  concur (textDeriv cx p s)
         (choice (ConcurOneOrMore p) Text)
```

For `Partition`, we create an `After` pattern that contains the derivative.

```
textDeriv cx (Partition p) s =
  after (textDeriv cx p s) Empty
```

No other patterns can match a text event; the default is specified as

```
textDeriv _ _ _ = NotAllowed
```

Start-tag Derivatives

Start tags are handled in a very generic way by all the patterns, except the `Range` pattern, whose derivative is a group of the content pattern for the range followed by an `EndRange` pattern for the range. Note that the `EndRange` pattern is created with the same qualified name and ID as the matched range.

```
startTagDeriv :: Pattern -> QName -> Id -> Pattern
startTagDeriv (Range nc p) qn id =
  if contains nc qn then group p (EndRange qn id)
                    else NotAllowed
startTagDeriv (Choice p1 p2) qn id =
  choice (startTagDeriv p1 qn id)
         (startTagDeriv p2 qn id)
startTagDeriv (Group p1 p2) qn id =
  let d = group (startTagDeriv p1 qn id) p2
  in if nullable p1 then choice d (startTagDeriv p2 qn id)
                    else d
startTagDeriv (Interleave p1 p2) qn id =
  choice (interleave (startTagDeriv p1 qn id) p2)
         (interleave p1 (startTagDeriv p2 qn id))
startTagDeriv (Concur p1 p2) qn id =
  let d1 = startTagDeriv p1 qn id
      d2 = startTagDeriv p2 qn id
  in choice (choice (concur d1 p2) (concur p1 d2))
            (concur d1 d2)
startTagDeriv (Partition p) qn id =
  after (startTagDeriv p qn id) Empty
startTagDeriv (OneOrMore p) qn id =
  group (startTagDeriv p qn id)
        (choice (OneOrMore p) Empty)
startTagDeriv (ConcurOneOrMore p) qn id =
  concur (startTagDeriv p qn id)
         (choice (ConcurOneOrMore p) anyContent)
startTagDeriv (After p1 p2) qn id =
  after (startTagDeriv p1 qn id) p2
startTagDeriv _ _ _ = NotAllowed
```

End Tags

End tags are matched by `EndRange` patterns. An id is used to support self-overlap: when an `EndTagEvent` matches an `EndRange` pattern, the names have to match and so do the ids.

```
endTagDeriv :: Pattern -> QName -> Id -> Pattern
endTagDeriv (EndRange (QName ns1 ln1) id1)
            (QName ns2 ln2) id2 =
 if id1 == id2 ||
    (id1 == '' && id2 == '' && ns1 == ns2 && ln1 == ln2)
 then Empty
 else NotAllowed
endTagDeriv (Choice p1 p2) qn id =
  choice (endTagDeriv p1 qn id)
         (endTagDeriv p2 qn id)
endTagDeriv (Group p1 p2) qn id =
  let p = group (endTagDeriv p1 qn id) p2
  if nullable p1 then choice p (endTagDeriv p2 qn id)
                 else p
endTagDeriv (Interleave p1 p2) qn id =
  choice (interleave (endTagDeriv p1 qn id) p2)
         (interleave p1 (endTagDeriv p2 qn id))
endTagDeriv (Concur p1 p2) qn id =
  let d1 = endTagDeriv p1 qn id
      d2 = endTagDeriv p2 qn id
  in choice (choice (concur d1 p2) (concur p1 d2))
            (concur d1 d2)
endTagDeriv (Partition p) qn id =
  after (endTagDeriv p qn id) Empty
endTagDeriv (OneOrMore p) qn id =
  group (endTagDeriv p qn id)
        (choice (OneOrMore p) Empty)
endTagDeriv (ConcurOneOrMore p) qn id =
  concur (endTagDeriv p qn id)
         (choice (ConcurOneOrMore p) anyContent)
endTagDeriv (After p1 p2) qn id =
  after (endTagDeriv p1 qn id) p2
endTagDeriv _ _ _ = NotAllowed
```

Implementation

Implementations of this algorithm can be optimized in the same way as Relax NG implementations:

- storing the result of the utility functions `nullable` and `allowsText`
- interning patterns to make it easy to test if two patterns are the same
- eliminating redundant choices
- using memoization to record the result of derivatives so they don't have to be recalculated

These techniques have been used in a proof-of-concept XSLT 2.0 implementation of Creole.

Conclusion

We've started to recognise XML's limitations in the representation of data, and the role of alternatives such as JSON (www.json.org). In the same way, we should be exploring the real requirements of documents, which include overlapping structures.

In fact, there's been a persistent, but low-level, thread of research into overlapping markup from the earliest days of SGML, and the limitations of milestones and CONCUR have been well-known for many years. Various methods of representing and querying overlapping markup have been proposed (see [DER04] for an overview), but, as we've seen, none of the methods of validating overlap have been entirely satisfactory.

In [SPER06], Sperberg-McQueen offered some goals for methods of validating overlapping structures, before acknowledging that rabbit/duck grammars don't meet them all:

- The method should be powerful enough to do useful work.
- It need not, however, be universal or Turing-complete. In the interests of being able to reason about the mechanism, Turing completeness should be avoided if possible.
- It should provide a clear comprehensible method of expressing constraints on overlapping structures. This is not easy to judge at first exposure to any formalism; the ease with which an idea is first grasped is not the same as the clarity given by that idea when it is understood.
- It should be be built around simple core ideas; the algorithms involved should be simple; the notation should be simple. Again, this is not easy to judge without more experience.
- Ideally, it should be possible to validate both data streams and graph data structures representing the overlapping structures of interest. DTDs and most XML schema languages have this property: they can be used to check both an XML document in serialized form and a tree derived from such an XML document.

Creole meets almost all these goals. It is powerful enough to validate documents that involve many types of overlap, such as the Biblical example that we've looked at in this paper, but it's not Turing complete. It provides a clear, comprehensible method of expressing constraints. The core idea, algorithm and notation are as simple as those for Relax NG, and compare well to the CONCUR-based methods that we've looked at.

Indeed, Creole's patterns can even be useful in schemas for plain XML. Mixed content models can be expressed using patterns like:

```
inline = mixed { italic* & bold* & u* }
italic = range italic { text }
bold   = range bold   { text }
u      = range u      { text }
```

which allow each element to contain any of the others but not themselves (e.g. `<italic>` can contain `<bold>` and `<u>` but not `<italic>`), a constraint that can't be expressed using the more usual technique:

```
inline = mixed { (italic | bold | u)* }
italic = element italic { inline }
bold   = element bold   { inline }
u      = element u      { inline }
```

The Brzozowski-derivative-based algorithm for validating documents with Creole requires them to be represented as a sequence of events, which enables streaming implementations. Although designed to be used with LMNL, Creole is data-model agnostic, and can be used with documents represented using any of the techniques we've looked at — milestone elements, processing instructions, TexMecs, or any other syntax — if that syntax can be mapped onto start tags, end tags and text. However, like rabbit/duck grammars, Creole can't handle some of the more esoteric structures found in the overlapping markup literature, such as discontinuous and virtual elements [HUIT06].

Creole's XSLT 2.0 implementation has proved that the algorithm works, but it is too slow, because of the constraints of the language, to be a real-world validator. An implementation in a mainstream language is the next step.

It would be interesting to see how Creole could be applied to querying documents, in particular for extracting well-formed hierarchies on demand. But even as just a schema language, Creole stands alone in its approach, and approachability, for validating overlapping markup.

Bibliography

[BRUN06] E. Bruno & E. Murisasco. 2006. Describing and querying hierarchical XML structures defined over the same textual data. In *Proceedings of the 2006 ACM symposium on Document engineering*, Amsterdam, October 2006. [WWW] http://portal.acm.org/citation.cfm?id=1166199

[CLA02] J. Clark. 2002. An algorithm for RELAX NG validation. [WWW] http://www.thaiopensource.com/relaxng/derivative.html

[DER04] S. DeRose. 2004. Markup Overlap: A Review and a Horse. In *Proceedings of Extreme Markup Languages 2004*, Montreal, August 2004. [WWW] http://www.idealliance.org/papers/extreme/proceedings/html/2004/DeRose01/EML2004DeRose01.html

[DUR02] P. Durusau & M.B. O'Donnell. 2002. Coming Down from the Trees: Next step in the evolution of markup? In *Proceedings of Extreme Markup Languages 2002*, Montreal, August 2002. [WWW] http://www.idealliance.org/papers/extreme/proceedings/html/2002/Durusau01/EML2002Durusau01.html

[HIL05] M. Hilbert, O. Schonefeld, & A. Witt. 2005. Making Concur Work. In *Proceedings of Extreme Markup Languages 2005*, Montreal, August 2005. [WWW] http://www.idealliance.org/papers/extreme/Proceedings/html/2005/Witt01/EML2005Witt01.xml

[HUIT06] C. Huitfeldt & C.M. Sperberg-McQueen. 2006. Representation and processing of Goddag structures: implementation strategies and progress report. In *Proceedings of Extreme Markup Languages 2006*, Montreal, August 2006. [WWW] http://www.idealliance.org/papers/extreme/proceedings/html/2006/Huitfeldt01/EML2006Huitfeldt01.html

[SCH06] O. Schonefeld & A. Witt. 2006. Towards Validation of Concurrent Markup. In *Proceedings of Extreme Markup Languages 2006*, Montreal, August 2006. [WWW] http://www.idealliance.org/papers/extreme/Proceedings/html/2006/Schonefeld01/EML2006Schonefeld01.html

[SPER05] C.M. Sperberg-McQueen. 2005. Applications of Brzozowski derivatives to XML Schema processing. In *Proceedings of Extreme Markup Languages 2005*, Montreal, August 2005. [WWW] http://www.idealliance.org/papers/extreme/proceedings/html/2005/SperbergMcQueen01/EML2005SperbergMcQueen01.html

[SPER06] C.M. Sperberg-McQueen. 2006. Rabbit/duck grammars: a validation method for overlapping structures. In *Proceedings of Extreme Markup Languages 2006*, Montreal, August 2006. [WWW] http://www.idealliance.org/papers/extreme/proceedings/html/2006/SperbergMcQueen01/EML2006SperbergMcQueen01.html

[TENN02] J. Tennison & W. Piez. 2002. The Layered Markup and Annotation Language (LMNL). In *Proceedings of Extreme Markup Languages 2002*, Montreal, August 2002. [WWW] http://www.idealliance.org/papers/extreme/proceedings/html/2002/Tennison02/EML2002Tennison02.html

Appendix: Creole Grammar for Biblical Example

```
<grammar xmlns="http://lmnl.net/ns/creole">
<start>
  <ref name="book" />
</start>

<define name="book">
  <element name="book">
    <concur>
      <oneOrMore>
        <ref name="page" />
      </oneOrMore>
      <group>
        <ref name="title" />
        <concur>
          <oneOrMore>
            <ref name="chapter" />
          </oneOrMore>
          <oneOrMore>
            <ref name="section" />
          </oneOrMore>
        </concur>
      </group>
    </concur>
  </element>
</define>

<define name="page">
  <range name="page">
    <attribute name="no" />
    <text />
  </range>
</define>

<define name="title">
  <element name="title"><text /></element>
```

```
</define>

<define name="chapter">
  <range name="chapter">
    <attribute name="no" />
    <oneOrMore>
      <ref name="verse" />
    </oneOrMore>
  </range>
</define>

<define name="verse">
  <range name="verse">
    <attribute name="no" />
    <text />
  </range>
</define>

<define name="section">
  <range name="section">
    <ref name="heading" />
    <oneOrMore>
      <ref name="para" />
    </oneOrMore>
  </range>
</define>

<define name="heading">
  <element name="heading">
    <ref name="indexedText" />
  </element>
</define>

<define name="indexedText">
  <concurOneOrMore>
    <mixed>
      <zeroOrMore>
        <ref name="index" />
      </zeroOrMore>
    </mixed>
  </concurOneOrMore>
</define>

<define name="para">
  <range name="para">
    <concur>
      <oneOrMore>
        <ref name="verse" />
      </oneOrMore>
      <oneOrMore>
        <ref name="s" />
      </oneOrMore>
    </concur>
  </range>
</define>
```

```
<define name="s">
  <range name="s">
    <ref name="indexedText" />
  </range>
</define>

<define name="index">
  <range name="index">
    <attribute name="ref" />
    <text />
  </range>
</define>

</grammar>
```