Validating Scripted Web-Pages

Dr R G Stone

Department of Computer Science
Loughborough University
Leicestershire, LE11 3TU, England

Abstract

The validation of XML documents against a DTD is well understood and tools exist to accomplish this task. But the problem considered here is the validation of a generator of XML documents. The desired outcome is to establish for a particular generator that it is incapable of producing invalid output. Many (X)HTML web pages are generated from a document containing embedded scripts written in languages such as PHP. Existing tools can validate any particular instance of the XHTML generated from the document. However, there is no tool for validating the document itself, guaranteeing that all instances that might be generated are valid.

A prototype validating tool for scripted-documents has been developed which uses a notation developed to capture the generalised output from the document and a systematically augmented DTD.

Keywords: VALIDATION, XHTML, WML, PHP, DTD.

1 Introduction

The validation of a static web-page against a DTD can be achieved by certain browsers (e.g. Internet Explorer[1]), by web-based services (such as that offered by W3C[2], WDG[3]) and by commercial products (such as the CSE HTML Validator[4]).

However, millions of dynamic web documents exist, scripted using languages like PHP[5], which are capable of generating different XML pages each time they are browsed but there is no method by which the source document itself can be validated.

1 Email: R.G.Stone@lboro.ac.uk
The problem of validating generators of web pages has been tackled by various researchers by constructing controlled environments where invalid output is not possible. This has been done by controlled macro substitution in JWIG [6] for example or by the design and use of a special purpose language as in XDuce[7] and CDuce[8]. The languages XDuce and CDuce both have roots in ML. They both feature an XML tree data structure as a built-in primitive data type with (very) strict compile-time type-checking which can be derived from a DTD or Schema. A preliminary release of an extension of OCaml[9] with CDuce types has recently been released.

XQuery[10] as implemented by Galaxy[11] has a strong compile-time type-check that ensures that a query will not raise any type errors. The Haskell community have evolved WASH[12] and HaXML[13]. A version of C# called Cω (C-omega)[14] has been released by Microsoft which contains a data type extension for XML.

These systems can solve the validation problem neatly for those able and willing to adopt a new strategy, possibly requiring a move to a new programming language. But they do not offer an immediate solution of the legacy problem for users continuing to use older scripting languages which is addressed in this paper. The separate issue of whether any of these new systems are particularly suitable as the implementation language for the system proposed in this paper is discussed in a later section.

This paper addresses the legacy problem of the validation of documents scripted in languages with no built-in validation features or checks. For presentation the examples used will be of PHP generating WML but the techniques used apply equally well to other scripting languages and other XML compliant languages, notably XHTML.

2 Embedded Scripting

A web-page containing server-side scripting must have the script executed before being passed to the browser. There are several server-side scripting languages (PHP[5], ASP[15], Perl[16], etc.). At its simplest, a server-side scripting language generates its output by echo or print commands. The scripted elements are often embedded among the marked-up text so the code to generate a minimal WML page using PHP could look like this

```php
<?php
    echo "<card>";
    echo "</card>";
?>
```
In this and subsequent examples, the required `<?xml ...>` header and the `<!DOCTYPE wml ...>` header lines are omitted for brevity. Also note that PHP code is written inside ‘brackets’ which can be written

```php
<?php ...
```

The common abbreviation

```text
<? ... ?>
```

is not XML compliant.

## 3 Validation against a DTD

The context of this paper is where a script is used to deliver a page that is valid XML according to a Document Type Definition (DTD)[17]. A DTD describes the tags that can be used, their attributes and the content that the tags enclose. As an example, a simplified extract of the WML DTD[18] can be shown as

```xml
<!ELEMENT wml ( card+ )>
<!ELEMENT card ( p* )>
<!ELEMENT p ( #PCDATA )*>
```

This DTD notation can be read as follows. For a document to be a valid WML document there must be a single `wml` element which must contain at least one (`+`) `card` element. Each `card` element may contain zero or more (`*`) paragraph elements (`p`). Finally each paragraph element may contain an arbitrary amount of ‘Parsed Character Data’ (meaning anything that is not a tagged element). The part of the DTD which defines attribute structure is not shown. The output of the script in the previous section would be

```xml
<wml>
  <card>
  </card>
</wml>
```

and this would therefore be acceptable and be taken to be exercising the right to have no paragraph elements (`p`).

## 4 Informal Validation of Scripted Web-Pages

Here is an example of a PHP script which contains a structured statement (a loop)

```php
<wml>
  <card>
  ?
```
while($i<$limit){
    echo "<p>";
    ...
    echo "</p>";
    $i++;
}

<?
</card>
</wml>

We might argue informally that, whatever the value of $limit, the result of this script is valid WML because the while-loop, when executed, will always generate paragraph tags (<p>, </p>) in pairs and that the <card> tag accepts any number of such pairs (including none).

A more formal way of approaching this is to capture the output of the script using the star notation borrowed from regular expressions

<wml> <card> ( <p> ... </p> )* </card> </wml>

This may be read as describing the output as a wml tag containing a card tag which in turn contains zero or more paragraph tags. It is this output expression which is ‘checked’ against the WML DTD. The wml element contains exactly one card element (1 or more is allowed) and the card element contains zero or more paragraph elements (zero or more allowed). The idea of using regular expression notation to capture the generalised output from a script is developed further in what follows. However the notation is converted into XML style so that the result can still be validated by a DTD obtained by augmenting the original with extra rules. Hence

<wml> <card> ( <p>...</p> )* </card> </wml>

will become

<wml> <card> <p_list0> <p>...</p> </p_list0> </card> </wml>

Other invented tags like <p_list0> will eventually be needed and they will be referred to as meta-tags.

5 Generalised Output and Augmenting the DTD

A system is envisaged in which the scripted web-page is passed through a processor to obtain the generalised output expression and the generalised output expression is then validated against a DTD which has been obtained by augmenting the original DTD with rules involving the meta-tags. The various repetition and selection control structures in the scripting language will re-
quire appropriate meta-tags to describe their contribution to the generalised output expression. These are summarised in Table 1. The correspondence with the regular expression operators used in the DTD which is shown in the same table will provide the insight into how the DTD should be augmented to accept the meta-tags.

Continuing the example in the previous section, if a scripted while loop has produced

```html
<p_list0> <p>...</p> </p_list0>
```

the DTD will need to be augmented to accept this as a replacement for

```html
<p>...</p> <p>...</p> ... <p>...</p>
```

For this example it would be sufficient to replace all occurrences of `p*` in the DTD with `(p*|p_list0)` and to add the definition

```html
<!ELEMENT p_list0 ( p )>
```

<table>
<thead>
<tr>
<th>Concept</th>
<th>RegExp</th>
<th>Program Control</th>
<th>Example Code</th>
<th>Meta-tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,2,...</td>
<td>*</td>
<td>while loop</td>
<td>while(...)</td>
<td>&lt;t_list0&gt;</td>
</tr>
<tr>
<td>1,2,3,...</td>
<td>+</td>
<td>repeat loop</td>
<td>do...while()</td>
<td>&lt;t_list1&gt;</td>
</tr>
<tr>
<td>option</td>
<td>?</td>
<td>short conditional</td>
<td>if(...)</td>
<td>&lt;t_option&gt;</td>
</tr>
<tr>
<td>choice</td>
<td></td>
<td>long conditional</td>
<td>if()...else...</td>
<td>&lt;t_choices&gt;</td>
</tr>
</tbody>
</table>

Table 1

A table of correspondences between regular expression operators, program control structures and meta-tags

However only the simplest case has been considered so far where a sequence of literal paragraph elements has been created entirely by a simple while loop. In the more general case a script may be written to generate a sequence of paragraph elements using any mixture of literal tags, loops and conditionals. The following example is more realistic as it creates a sequence of paragraph elements via a sequence involving literals, a loop and a conditional:

```html
<wml>
<card>
<?
    echo "<p>...</p>";
    while(...){
        echo "<p>...</p>";
    }
</card>
</wml>
```
if(...)echo "<p>...</p>";
?>
</card>
</wml>

In this case the generalised output expression will look like

<wml>
<card>
  <p>...</p>
  <p_list0>
    <p>...</p>
  </p_list0>
  <p_option>
    <p>...</p>
  </p_option>
</card>
</wml>

To express this generality the entity $p0$ is introduced so that $p*$ in the DTD is replaced by $(%p0;)*$ with the definition

```xml
<!ENTITY % p0 (p|p_list0|p_list1|p_option|p_choices) >
```

Under this definition $(%p0;)*$ means a sequence of zero or more elements each of which contributes zero or more paragraph elements.

This rule must be repeated for all tags ($t$), so that wherever $t*$ occurs in the DTD it is to be replaced by $%t.star$; under the definitions

```xml
<!ENTITY % t.star (%t0;)* >
<!ENTITY % t0 (t|t_list0|t_list1|t_option|t_choices) >
```

Note that in $(...|t|...)*$, where the * applies to various alternatives including $t$, the $t$ should also be replaced by the entity $%t0$.

6 The Augmented DTD

All of the changes to the DTD so far have been motivated by the occurrence of ‘zero or more’ tagged elements, including meta-tagged elements, in the output expression which are validated by substituting occurrences of $t*$ in the DTD. Obviously it now remains to look at what other parts of the DTD might need augmenting. Repeat loops with their signature output of ‘one or more’ can be captured by the meta-tag $t_list1$ and would be expected to cause substitutions for $t+$ within the DTD. Short conditionals (no else part) with their signature ‘optional’ output can be captured by the meta-tag $t_option$ and would be expected to cause substitutions for $t?$ within the DTD. Long
conditionals (with an else part) have a signature ‘alternative’ output and can be captured by the meta-tags \texttt{t\_choices} and \texttt{t\_choice} like this
\begin{verbatim}
  <t\_choices><t\_choice>...this...</t\_choice>
  <t\_choice>...or this...</t\_choice></t\_choices>
\end{verbatim}
A long conditional would be expected to cause substitutions for any unadorned instances of \( t \) (that is an occurrence of \( t \) in the absence of any of the operators \(*\)', '+', '?') because alternative choices for a single tag \( t \) are being offered.

The substitution for \( t+ \) in the DTD is more complicated than for \( t* \) because it is necessary to ensure that at least one element with tag \( t \) is present. Before considering the substitution in detail, compare the following four entity definitions:

(i) Zero or more occurrences of elements \( t \), \( t_0 \) (presented earlier)
\begin{verbatim}
<!ENTITY % t0 ( t|t\_list0|t\_list1|t\_option|t\_choices )>
\end{verbatim}

(ii) One or more occurrences of elements \( t \), \( t1 \)
\begin{verbatim}
<!ENTITY % t1 ( t|t\_choices|t\_list1 )>
\end{verbatim}

(iii) Zero or one occurrences of element \( t \), \( t_01 \)
\begin{verbatim}
<!ENTITY % t01 ( t|t\_option|t\_choices )>
\end{verbatim}

(iv) Exactly one element \( t \), \( t11 \)
\begin{verbatim}
<!ENTITY % t11 ( t|t\_choices )>
\end{verbatim}

It is now possible to replace \( t+ \) by the entity \texttt{t\_plus} under the definition
\begin{verbatim}
<!ENTITY % t\_plus ( (t\_option|t\_list0)*, %t1; , %t\_star; ) >
\end{verbatim}
This can be read as defining \texttt{t\_plus} to be zero or more elements that cannot be relied upon to contain a \( t \) tag, followed by an element which definitely contains at least one \( t \) tag, followed by zero or more elements which will contribute zero or more \( t \) tags.

The substitution for \( t? \) in the DTD is the entity \texttt{t01} with the definition already given. The substitution for \( t \) is the entity \texttt{t11} with the definition already given.

The substitutions to be made to the DTD are summarised in Table 2. To support these substitutions there are the new entities \texttt{t\_star}, \texttt{t\_plus}, \texttt{t0}, \texttt{t1}, \texttt{t01} and \texttt{t11} to be added as defined above and finally the new element rules describing the derived tags \texttt{t\_list0}, \texttt{t\_list1}, \texttt{t\_option}, \texttt{t\_choices} and \texttt{t\_choice} for each tag \( t \).

\begin{verbatim}
<!ELEMENT t\_list0 %t\_star; >
<!ELEMENT t\_list1 %t\_plus; >
<!ELEMENT t\_option %t01; >
\end{verbatim}
<table>
<thead>
<tr>
<th>DTD phrase</th>
<th>replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t^* )</td>
<td>( %t.star; )</td>
</tr>
<tr>
<td>((...</td>
<td>t</td>
</tr>
<tr>
<td>( t+ )</td>
<td>( %t.plus; )</td>
</tr>
<tr>
<td>((...</td>
<td>t</td>
</tr>
<tr>
<td>( t? )</td>
<td>( %t01; )</td>
</tr>
<tr>
<td>( t )</td>
<td>( %t11; )</td>
</tr>
</tbody>
</table>

Table 2
An table of replacements to be made in the DTD

```xml
<!ELEMENT t_choices (t_choice,t_choice) >
<!ELEMENT t_choice %t11; >
```

Note that the augmentation rules do not alter the meaning of the DTD when no meta-tags are present. For example if \( t^* \) is replaced by \( t0^* \) and \( t0 \) is defined to be \( (t|t_list0|t_list1|t_option|t_choices) \) then, in the situation where no meta-tags \( (t_list0, t_list1, t_option, t_choices) \) are present, the substitution degenerates back to \( t^* \).

In the prototype the process of augmenting the DTD is handled by a prolog program which reads the original DTD, generates the extra ELEMENT definitions and ENTITY definitions and outputs the augmented DTD. This is made easier in SWI-prolog[21] by using a pre-written module[22] to read the DTD.

### 7 The Script Processor

Earlier it was stated that the script validation system was constructed of two parts. The first part has to process the script, introduce the meta-tags and generate the generalised output expression. The second part validates the output expression against an augmented DTD. In the prototype the first part, the script processor, has itself been split into two stages. The script processor first generates an output expression using general meta-tags like \( list0, list1, option \) and \( choices \). A second stage inspects the output of the first and inserts the correct tags to change these to specific meta-tags like \( p_list0, card_option \).

In the current implementation the first stage of the script processor is written in C using LEX[19] and YACC[20] to parse the script and this stage
produces an output expression containing general meta-tags. For example

\[
\text{<wml> <card> list0 p>...\</p> \</list0> \</card> \</wml>
\]

The second stage is written in prolog and produces specific meta-tags, for example

\[
\text{<wml> <card> p_list0 p>...\</p> \</p_list0> \</card> \</wml>
\]

8 Current implementation

The current implementation for PHP scripts producing WML and XHTML works perfectly well on a large class of scripts. However, if it fails to validate a script, it is not necessarily the case that the script is capable of emitting invalid output. The weak point is the first stage where the meta-tags are inserted. The problem lies with assuming that a control structure in the script language will generate a complete tagged structure capable of being described by the meta-tags. This does not always happen. An example to illustrate this would be

\[
\text{echo "<p>";} \\
\text{echo "0";} \\
\text{while(...){} \\
\text{echo "</p>";} \\
\text{echo "<p>";} \\
\text{echo "1";} \\
\text{echo "</p>";}
\]

For any particular execution this script will result in a sequence like

\[
<p> 0 </p> <p> 1 </p> <p> 1 </p> <p> 1 ... </p>
\]

which is valid. However it will be given the following meta-tags

\[
<p> 0 list0 \</p> <p> 1 \</p
\]

This expression, in which the tags are not properly nested, fails the second stage of the process (replacing general meta-tags with specific meta-tags) because the input stage assumes that the input is well-formed XML.

Work has begun to introduce an extra middle stage into the processor which uses rules along the lines of

\[
ab(cab)*c \Rightarrow abc(abc)* \Rightarrow (abc)+
\]

so that the example above can be manipulated to

\[
<p> 0 \</p> list0 <p> 1 \</p
\]
The problem with this is that the starting expression is not valid XML precisely because the tags are not properly nested, so that the expression cannot be read and manipulated as an XML document. This means that the manipulation has to be done by treating the expression merely as a linear mixture of starting tags, ending tags and non tag elements. This makes the processing harder but not intractable.

A more serious problem exists with the current code which replaces general meta-tags with specific meta-tags. At present, if the processor meets a opening `<list0>` tag it checks all the top-level tags up to the closing `</list0>` tag expecting them all to be of the same type (say) so that the general tag `<list0>` can be changed to `<t_list0>`. This will not always be the case as in the following example

```php
echo "<p>";
while(...){
    echo "<ul>...</ul>";
    echo "<br />";
}
echo "</p>";
```

The processor is presented with

```xml
<list0><ul>...</ul><br /></list0>
```
and cannot find a tag name t to change `<list0>` to `<t_list0>`. There are potential solutions to this. One is that with reference to the DTD it may be possible to change the scope of the `<list0>` tags thus:

```xml
<list0><ul>...</ul></list0> <list0><br /></list0>
```

Although this changes the meaning of the expression, if the DTD contains a rule along the lines of

```xml
<!ELEMENT p (...|ul|...|br|...)* >
```
the change will not alter the validity of the expression and so the validity check on the new expression will obtain the desired result. In practice it has been possible in many cases like this for the programmer to circumvent the issue by adding an enclosing `<span>` or `<div>` tag within the loop.

A further problem lies with the simplicity of the first stage of the processor. Because it is largely syntactic in nature it does not, and cannot, actually execute the script language. This means that if the script generates any tags by any other method than printing literals (for example by constructing them by string concatenation or obtaining them as part of a database lookup) then these tags will not be represented in the generalised output and consequently
these tags will not be validated.

9 Alternative Implementation Strategies

This paper discusses scripted documents that are intended to produce valid XML according to some DTD as output. In the introduction, new systems for handling XML documents via programming languages were briefly discussed. It is possible to consider these as possible implementation languages for the ideas introduced in this paper. These new systems are committed to XML documents to such an extent that any data document will also be required to be at least a well-formed XML document. The PHP language allows scripting to be embedded in <?...?> or <?php...?> but only the second kind is XML compliant. So at the very least a preprocessor will be required to rewrite the enclosing tags of the source document into XML format. After this has been done the input of the PHP scripted document can be achieved in a single step (for example using CDuce a script document script.php would be read in by supplying the name of the file as a parameter to the built-in function load_xml). However the text of the PHP commands would be held as PCDATA content of the php tag and thus would not have any structure. It would still be necessary to write a PHP parser to inspect the detail of the script. Furthermore, because of the way that the PHP scripts are written, the opening part of a structured statement ( e.g. while(...){} ) could easily be in a different tag to the closing part ( } ) as in the example below. Thus it is not simply a matter of parsing the individual php tags individually.

```xml
<wml><card>
<p>...</p>
<?php while(...){
    <p>...</p>
<?php }
</wml>
```

The new languages have powerful and elegant ways to transform an input XML tree into an output XML tree which would be very useful later in the process. However the strong compile-time type-checking causes problems at the point where the process is required to create tags on-the-fly. Recall that there is a requirement to transform <list0><T>stuff</T></list0> to <T_list0><T>stuff</T></T_list0> for any tag T. This construction of a tag name out of pieces ( T extended by _list0 ) is exactly the kind of thing that the type checking systems are designed to prevent. It would be possible to design workarounds either handling each possible tag separately (p to p_list0, h1 to h1_list0, etc) or representing the new tag T_list0 as a pair (T,list0). The
first possibility would drastically expand the amount of code needed and the second possibility would require a final transform via a printing routine to convert the pairs e.g. (p,list0) to a string p_list0. But they remain workarounds and are defeating the purpose of the design of the language.

The conclusion then is that the newer languages can (and do) handle the ordinary transformation of XML trees very elegantly and in a type safe way. However it seems that this meta level of programming (the construction of meta tags from components of the input data) cannot at present be handled in a type safe way.

Another general issue is whether the technique is applicable to validation by Schema which represent a newer standard than DTDs. An obvious criticism of the design of DTDs is that there was no attempt to define their notation so that a DTD was itself an XML document. In broad terms Schema are DTDs in XML format with better facilities for typing the leaves of an XML tree. Thus there is no reason in principle why the technique proposed should not be applicable to validation by Schema.

10 Summary

The concept of validating a scripted web-page rather than its output is thought to be novel and potentially very useful, at least for the large number of legacy sites which use this technology. A method has been found to validate such scripts which depends on processing the script to provide a generalised output expression and then validating this against an augmented DTD. The method has been prototyped for PHP scripts generating WML and XHTML. The method is readily applicable to any other combination of procedural scripting language and XML-based output.

Although the method can validate a large class of scripts it has its limitations. The processor which produces the generalised output expression has to be able to recognise where the script is generating tags. The current prototype requires these to be literal text within an echo/print command and not ‘hidden’ by string manipulation operators or resulting from database lookup. The current prototype also requires control statements within the script to generate well-formed XML, although there are plans to extend the processor to accommodate non well-formed output in situations where special rules can be applied which are derived from regular expression equivalences.
References


