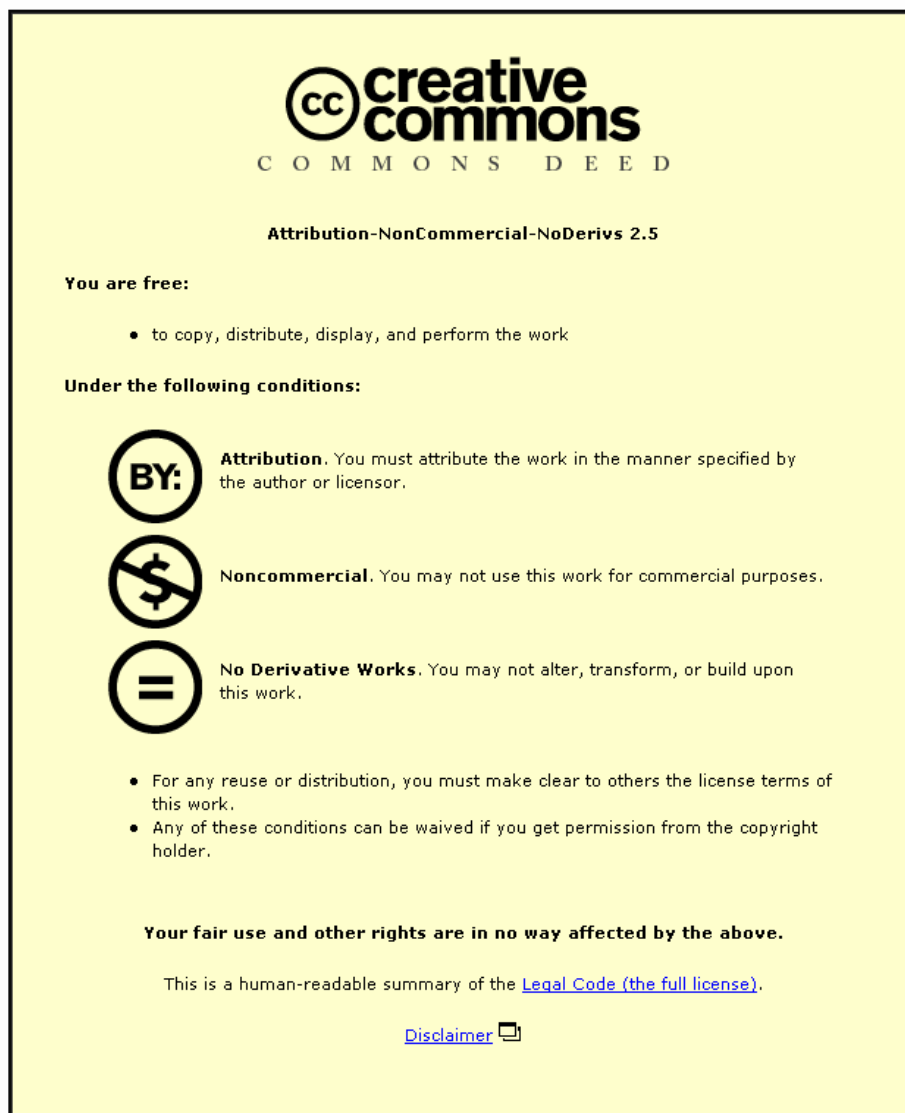




This item was submitted to Loughborough's Institutional Repository by the author and is made available under the following Creative Commons Licence conditions.



The image shows a yellow rectangular box containing the Creative Commons Attribution-NonCommercial-NoDerivs 2.5 license summary. At the top is the Creative Commons logo (CC) and the text 'creative commons' in a bold, lowercase font, with 'COMMONS DEED' in a smaller, spaced-out font below it. Underneath is the license title 'Attribution-NonCommercial-NoDerivs 2.5'. The text 'You are free:' is followed by a bullet point: 'to copy, distribute, display, and perform the work'. Below this is the heading 'Under the following conditions:'. Three icons are listed vertically: a 'BY' icon, a crossed-out dollar sign icon, and an equals sign icon. Each icon is followed by a bold heading and a description: 'Attribution. You must attribute the work in the manner specified by the author or licensor.', 'Noncommercial. You may not use this work for commercial purposes.', and 'No Derivative Works. You may not alter, transform, or build upon this work.'. A final bullet point states: 'For any reuse or distribution, you must make clear to others the license terms of this work.' and 'Any of these conditions can be waived if you get permission from the copyright holder.'. At the bottom, it says 'Your fair use and other rights are in no way affected by the above.' and 'This is a human-readable summary of the [Legal Code \(the full license\)](#)'. A 'Disclaimer' link with a document icon is at the very bottom.

CC creative commons
COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

BY: **Attribution.** You must attribute the work in the manner specified by the author or licensor.


Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the [Legal Code \(the full license\)](#).

[Disclaimer](#) 

For the full text of this licence, please go to: <http://creativecommons.org/licenses/by-nc-nd/2.5/>

Validation of Dynamic Web Pages generated by an Embedded Scripting Language

Dr Roger G. Stone

Department of Computer Science

Loughborough University

Loughborough

LE11 3TU

email: R.G.Stone@lboro.ac.uk

tel: 01509 222686

fax: 01509 211586

Validation of Dynamic Web Pages generated by an Embedded Scripting Language

SUMMARY

This paper attempts to provide insight as to how to guarantee a statement like: My PHP script produces WML. To expand a little, the emphasis is to ensure that a script *always* produces a *valid* WML page. The context is where pages in a web-site are being created by an embedded scripting language (like PHP, ASP, Perl) and also that the resulting pages are to conform to a strict tagged mark-up scheme like WML or XHTML. Although there are validators for static pages there is nothing available to check that a page containing embedded scripting will (always) generate valid documents. What is required is a validator for dynamic web pages.

KEYWORDS

Validation, Dynamic Web-Pages, WML, XHTML, PHP, DTD

INTRODUCTION

Ever since the world-wide web came into existence a large proportion of HTML web-sites have contained pages with malformed HTML. However because the web-browsers have been lenient to a fault in rendering poorly structured documents, it has not been of great concern to most web users. In more recent times, when a large proportion of pages are being dynamically generated by various server-side scripting languages, it is still not essential to ensure that scripts produce properly structured HTML. However with the advent of XML, and the derived subset WML, the situation has changed. The browsers on WAP-enabled mobile phones are much less tolerant. The pages of a WAP site must be correctly structured. In particular they must be well-formed XML and in addition, to be valid, they must conform to the WML Document Type Definition (DTD). If a WML site is made up entirely of static WML pages, then each of these pages can be checked for conformance using standard validators [e.g. 1,2,3,4]. However if a page is dynamically generated then the question arises as to how the script responsible can be checked to ensure that it always delivers valid WML.

ENSURING VALID SCRIPT OUTPUT

The problem of guaranteeing valid XML output from programs has been tackled head on in several projects which offer completely new languages. Thus if a programmer is able and willing to learn and switch to an entirely different language then there is a solution. New languages like Jwig [5], CDuce [6], XDuce [7] either do not allow the programmer to generate invalid output because of the way the output is constructed via templates or the programs created have the ability to be analysed at compile time to guarantee that only valid documents can be constructed at run-time. Jwig is a Java-based development system incorporating the bigwig language and thus provides safe dynamic document construction. CDuce & XDuce are programming languages specifically aimed at manipulating XML documents and feature datatypes suited to that need including regular expression types. Although these languages provide an attractive solution for the future, there is a legacy problem with the vast number of existing scripted web-pages and for those who are unwilling or unable to switch to the newer languages.

Exhaustive testing can only prove that a program generates correct output for the most trivial of programs. For any non-trivial program the only proof possible is that the program output matches the program specification in some way. In the context of this paper the specification of the output is that it is valid WML (or XHTML). This is a syntactic specification. So the problem is one of demonstrating that a particular program (a script written in PHP, ASP, Perl or similar) produces output that is correctly structured according to a particular syntax.

Now proofs of program correctness are notoriously difficult to construct and ideally we are looking for a simple tool that can routinely be used by ordinary scripters to ensure that their scripts produce valid output. A solution is proposed which has two stages. Firstly a notation is developed in which every possible output from a script can be captured. Secondly a method is developed to check the notated output from the script, to decide whether all actual output would be syntactically correct. In this paper the two stages are developed for WML or XHTML as the mark-up language and PHP as the scripting language, although the method generalises to any XML conformant mark-up language and any procedural scripting language.

SCRIPTED WEB-PAGES

A web-page intended to be rendered by a browser is authored as a text file and should conform to some tagged mark-up language like HTML. In its simplest form it is referred to as a static web-page. However, if it contains any scripting elements to be executed by the web-server before delivery to the browser, then it is called a dynamic page. It is the validation of dynamic pages that is the subject of this paper.

There are several server-side scripting languages (PHP[8], ASP[9], Perl[10], 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 short WML page using PHP could look like this

```
<wml>
<?php
    echo "<card>";
    echo "</card>";
?>
</wml>
```

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 ... ?>
```

and which can, in certain circumstances, be abbreviated to

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

XML, DTDS AND WML

The context of this paper is where a script is used to deliver a page that has to conform to a strict tagged mark-up language. A WAP site[11] based on WML pages where at least some of the pages contain server-side scripting is an example. WML pages are XML pages which in addition conform to a Document Type Definition (DTD). An XML page is termed well-formed if it satisfies simple rules like an end tag for every start tag and strict nesting of tags. An XML page is termed valid if it conforms to a DTD. 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[12] can be shown as

```
<!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.

VALIDATING SCRIPTED WEB-PAGES

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

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

```
    }  
  ?>  
  </card>  
</wml>
```

We might argue informally that, whatever the value of \$limit, the result of this script is good 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). Another way of describing this is to realise that we have captured the output of the script using notation borrowed from regular expressions

```
<wml> <card> ( <p> not_a_tag* </p>)* </card> </wml>
```

Furthermore we have mentally 'checked' this 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).

VALIDATION BY TESTING

It has already been mentioned that the notion of proof by exhaustive testing is infeasible. However we briefly consider the generation of carefully chosen test sets. Since the notation of DTD's is fairly restricted, we could perhaps generate a test set of sample output - perhaps three samples in this case, the first with no paragraph tags, the second with exactly one and the third with two paragraph tags. This test set could be validated using conventional validators. If all the samples passed the validation test, some confidence would be gained that the script would always produce valid output. But this method is unworkable for more complicated scripts where the number of samples of output to be tested would expand factorially and still there would be no absolute proof. So we will return to the idea of capturing the output of a script using regular expression notation and working directly with that. For the trial implementation described in this paper validation is obtained by formally extending the DTD to allow it to recognise the regular expression description of the generalised output from the script.

AUGMENTING THE DTD

The point of having a rule like

```
<!ELEMENT card ( p* )>
```

in the DTD is to accept a sequence of any number of valid paragraph elements as valid content for a card. If a script contains a loop which, on each iteration, generates a paragraph element, we wish to capture the output of the script more like

```
(p_elt)*
```

than as

```
p_elt p_elt p_elt ...
```

As the environment is a tagged mark-up language, rather than use the '*' notation, it seems better to use a tag notation. So the meta-tag **<p_list>** is invented with definition

```
<!ELEMENT p_list (p)>
```

and will be used when the script emits a collection of zero or more paragraph elements via a while loop. It is anticipated that the output from any script can be captured via such notation. If this is done then the DTD which validates the output will need to be augmented with extra rules to accept the meta element **p_list** in places where (p*) is indicated. We might augment the DTD rules for **card** as follows

```
<!ELEMENT card ( p | p_list )* >
```


This effect of this last rule is to allow within a **card** structure, any number (*) of components, each being an individual paragraph element (<p>...</p>) or a paragraph list (<p_list><p>...</p></p_list>) which represents the output of a loop-structure in a script.

By continuing in this way the original DTD can be augmented so that it is directly capable of accepting the 'regular expression' notation version of the output from the example script which was originally written as

```
<wml> <card> ( <p> not_a_tag* </p>)* </card> </wml>
```

but would now be rewritten as

```
<wml><card><p_list><p>PCDATA</p></p_list></card></wml>
```

Now that the basic principle has been explained by example it may be useful to check the consequences of what has been proposed. A script is to be processed to produce an expression which represents all possible outputs from the script. Following this the expression is to be checked by a validator using a DTD augmented by rules involving additional meta-tags like <p_list>.

This apparently two-stage process is actually accomplished in three stages. It is convenient in the first stage of processing the PHP to introduce less specific meta-tags e.g. <LIST> rather than <p_list>. A middle stage is responsible for deducing the appropriate specific meta-tags like <p_list> which are then validated by the third stage.

NOTATING ALL POSSIBLE OUTPUT FROM A PHP SCRIPT

We need to be able to process a PHP script to obtain a meta-tagged expression representing its generalised output. It is required to build something which is more than a parser but less than a full-blown interpreter for PHP. Primarily it should be able to recognise *echo* commands and deduce the resultant output and also recognise structures like while-loops and 'encode' any output from them within meta-tags like <LIST>. Notice

that the relationship that has been exploited so far is that between the while-loop in the script and the meaning of the "*" in the DTD. So do other similar relationships exist?

The notation of a DTD is essentially to define the content of elements via

zero or more of	a*
at least one of	a+
option	a?
choice	a b
sequence	a,b

So far we have only introduced the <LIST> meta-tag for the "*" notation and linked it to the while-loop. The full set of meta-tags linked to program structures are shown below where <t> stands for an arbitrary tag:

(i) a* - <LIST> - linked with the while loop:

```
while(...) echo "<t>...</t>";
```

becomes

```
<LIST><t>...</t></LIST>
```

and eventually

```
<t_list><t>...</t></t_list>
```

(ii) a+ - <LIST1> - associated with the repeat loop:

```
do echo "<t>...</t>"; while(...);
```

becomes

<LIST1><t>...</t></LIST1>

(iii) a? - <OPTION> - associated with the short conditional

if (...) echo "<t>...</t>";

becomes

<OPTION><t>...</t></OPTION>

(iv) a|b - <CHOICES> - associated with the long conditional

if (...) echo "<t>1</t>"; else echo "<t>2</t>";

becomes

<CHOICES><CHOICE><t>1</t></CHOICE>

<CHOICE><t>2</t></CHOICE></CHOICES>

(v) a,b - no meta-tag is needed for sequence

Based on the relationship

for(E1;E2:E3)S = E1;while(E2){S;E3}

for loops can be treated as while loops in this context.

Now that we have extended the collection of meta-tags and looked at how they might be used, we can complete our extension of the definition of **card** which was originally

```
<!ELEMENT card (p*)>
```

and which was provisionally extended to

```
<!ELEMENT card ( p | p_list )* >
```

From a scripting point of view, the zero or more **p** elements that will form the content of a **card** can be produced by any combination of loops and conditionals, i.e. by any combination of meta-elements which deliver none, one or more paragraph elements. Using the complete set of meta-tags it can be seen that the rule for **p*** should be further extended to allow **list1**, choices and option components. So the full definition expands to

```
<!ELEMENT card ( p | p_option | p_choices | p_list | p_list1 )* >
```

By contrast when we are dealing with a "one or more" rule, for example

```
<!ELEMENT wml (card+) >
```

it would be extended in a more limited way to

```
<ELEMENT wml ( card | card_choices | card_list1 )+ >
```

It is not possible to include **card_option** or **card_list** structures as alternatives as these allow empty possibilities and would break the force of the "+".

THE MIDDLE STAGE

There are two considerations which suggest a middle stage in the process. The first, already presented, is the post processing of the meta-tagged output to make such changes as `<LIST>` to `<p_list>`. The second motivation comes from considering the DTD extension for rules involving repetition of "one or more" (+).

We would want sequences containing a definite card element like

```
<card>...</card>  
<? while(...) echo "<card>...</card>"; ?>
```

which on a first pass would be converted to

```
<card>...</card>  
<LIST><card>...</card></LIST>
```

and then to

```
<card>...</card>  
<card_list><card>...</card></card_list>
```

to be acceptable as `card+` i.e. `<card_list1>`. This suggests that a middle stage in the process before the extended DTD validation should include some algebraic simplification towards a canonical form. This process would be defined by simplification rules like

$$t_1 \langle t_list \rangle t_2 \langle /t_list \rangle \Rightarrow \langle t_list \rangle t_1 t_2 \langle /t_list \rangle$$

The right hand side of this rule is to be read as "there is a sequence of one or more `t` structures with `t1` and `t2` being representative of the elements involved. By regarding the component elements as being of five types (`t`,

`t_list`, `t_list1`, `t_option`, `t_choices`) it is clear that there are 25 cases of this type to consider where adjacent elements of the same type `t` are aggregated into a list or list1 structure. These rules are all added to the simplifier. There are also simplification rules for nested meta-tags for example

```
<t_list><t_list1>...</t_list1></t_list> => <t_list>...</t_list>
```

Now this coalescing of structures based on the same tag would not be safe if the DTD involved contained a rule like

```
<!ELEMENT exactly4p (p,p,p,p) >
```

If the simplification process found consecutive paragraph tags it would change them to `<p_list1>...</p_list1>` and the DTD would not be satisfied. However inspection shows that the WML and XHTML DTDs do not define sequences with adjacent tags the same.

There are two consequences of this simplification process for the extended rules for the DTD. The first is that, because of aggregation, definitions like `<!ELEMENT p_list (p)>` must minimally be upgraded to `<!ELEMENT p_list (p*)>` [but see also the appendix]. Secondly, changes made to occurrences of `t*` in the original DTD can now be simplified because all sequences of tags of the same kind will be reduced to either `<t_list>` or `<t_list1>`. The original DTD had

```
<!ELEMENT card (p*)>
```

which was extended to

```
<!ELEMENT card ( p | p_option | p_choices | p_list | p_list1 )* >
```

We can now omit the final "*" and write

```
<!ELEMENT card ( p | p_option | p_choices | p_list | p_list1 ) >
```

meaning that any sequence of zero or more paragraph structures will be represented either as an actual paragraph tag, an optional paragraph tag, choices of paragraph tag, a list of zero or more paragraph tags or a list of one or more paragraph tags. In fact the substitution chosen in practice uses the entity convention

```
<!ENTITY % p.star "( p | p_option | p_choices | p_list | p_list1 )" >
```

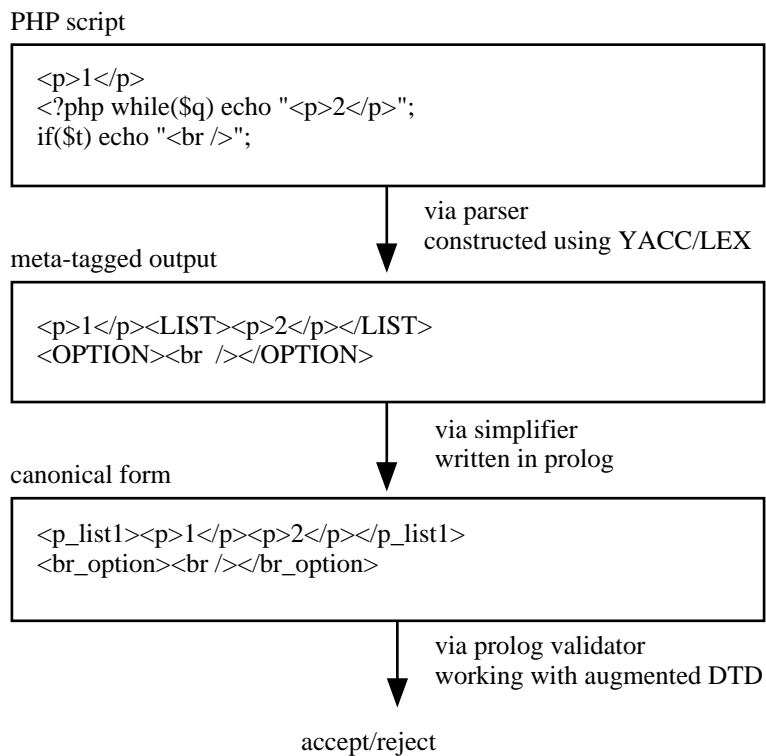
```
<!ELEMENT card ( %p.star; ) >
```

Similar considerations apply to occurrences of t^+ , $t?$ and $t_1|t_2$ in the original DTD and the substitutions are presented in full in the appendix.

THE PROTOTYPE IMPLEMENTATION

An initial trial was carried out using PHP as the scripting language and WML and (strict) XHTML as alternative mark-up languages.

The 3 Stage Process



The parser for PHP was written using LEX & YACC [13]. The DTDs for WML and XHTML are publicly available [12,14]. The DTDs were extended by hand using the entity notation as described above. Prolog was used as the main implementation language to prototype the middle and final stages. The meta tagged generalised output expression is imported into prolog, simplified and then the result validated against the (augmented) DTD. The resulting three stage process is shown in the diagram.

It is possible to create a recogniser for XML files conforming to a specific DTD using the prolog DCG system [15]. However by using SWI-Prolog [16] advantage can be taken of the prewritten SGML/XML parser package [17] which can both import an XML document into prolog and validate it against a chosen DTD. The XML import works by translating a tag structure like `<tag attributes>content</tag>` into a prolog term with the general structure element(`tag`, [`attributes`], [`content`]). The package is able to accept a DTD in its standard textual presentation [12,14].

The simplifier needs to implement the 25 adjacency rules mentioned earlier. It is implemented in prolog and so the rule given earlier as

$t_1 \langle t_list \rangle t_2 \langle /t_list \rangle \Rightarrow \langle t_list1 \rangle t_1 t_2 \langle /t_list1 \rangle$

becomes

$element(tag, [attrs], [\dots t1, element(t_list, [], [t2]) \dots])$

\Rightarrow

$element(tag [attrs], [\dots element(t_list1, [], [t1, t2]) \dots])$

These replacements are made during a tree walk of the content of the root tag.

When building the parser, no official syntax for PHP was located (but see [18]). However it was easy to create a syntax for most of the language including:

- the full range of operators in expressions
- the control statements
- function declarations and function calls
- arrays

One special feature of the compiling actions is that it is necessary to inspect structured statements to find out if they contribute any output. Only if the statements contribute output are the meta-tags are required. To see this consider the following extract

```
while($i<10){
    echo "<p>...</p>"; $i++;
}
while($j<10){
    $v[$j]=0; $j++;
}
```

Now the first loop (based on the counter \$i) makes a contribution to the WML output and should be represented in the output expression by

```
<LIST><p>...</p></LIST>
```

The second loop (based on the counter \$j) does not make a contribution to the WML output and has no representation in the output expression.

The most serious omission in the trial implementation is the lack of proper interpretation of the string in the echo statement. The correct interpretation of the PHP statement

```
echo "the value of variable x is $x";
```

requires variable interpolation, that is the value of the variable is to be inserted in place of its name (\$x). To keep the trial implementation simple, the variable interpolation has not yet been attempted. This means that the validator will only give the correct results if either

- the PHP script does not use variable interpolation or
- the only values that would be interpolated are PCDATA

Functions which contribute to the output by side effects (echo or inline tagged content) can be used. However functions which contribute to the output by return value are not handled correctly.

LIMITATIONS OF THE VALIDATION METHOD

One consequence of this syntactic approach to validation is that the script must work within structures rather than across them. The specific restriction is that each control structure used in the script must deliver either a single complete tagged element or a sequence of complete tagged elements all of the same type.

Two examples of scripting style which ultimately deliver valid output, but which are unacceptable to our validator are given. The first develops a list of tags but the loop involved generates an end tag followed by a start tag.

```
echo "<p>";
echo "0";
while(...){
    echo "</p>";
    echo "<p>";
    echo "1";
}
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 <LIST> </p> <p> 1 </LIST> </p>
```

Unfortunately the trial implementation uses a prewritten XML to prolog importer. Since this structure is not valid XML it will fail the import. So although there is a potential for writing rules to 'simplify' this kind of expression along the lines of

```
ab(cab)*c => abc(abc)* => (abc)+
```

they cannot be used in the prototype.

It would be possible in many cases like this to re-write the script with minimal effort to achieve the same effect. It is arguable that requiring structures in the script to generate whole tagged elements would produce an inherently cleaner and/or clearer version of the script anyway.

The second example is where a loop generates a list of tags which are complete in themselves but not enclosed.

```
echo "<p>";
while(...){
    echo "<strong>message</strong>";
    echo "<br />";
}
echo "</p>";
```

The simplifier is presented with

```
<LIST><strong>message</strong><br /></LIST>
```

and cannot find a tag name **t** to change **<LIST>** to **<t_list>**. In practice it has been possible in many cases like this to circumvent the issue by using an enclosing **** or **<div>** tag within the loop.

CONCLUSION & FURTHER WORK

Initial trials have proved encouraging. The department has an intranet with interlinked sites for students and staff. The dynamic pages are powered by PHP and MySQL. Typical scripts have been checked by the system as described above. The WML scripts were easy to check and showed up minor problems only. There are many quite involved HTML scripts. Representative examples of the more involved scripts have been checked. Since they were originally written as HTML and not XHTML, the first thing that needed to be done was to make them XML compatible. This means having closing tags for every opening tag and requires **
** to be changed to **<br**

/> and similar changes to <hr>, <img...> <link...>, etc and making sure that e.g. <p> tags have their closing </p>. The system showed up instances where the PHP structures were not delivering single whole tagged structures. In all the examples tested so far it has been possible (and simple) to rewrite the PHP accordingly and so have the system accept the updated script.

The examples tested include scripts which access a database to provide data to deliver to the WML page. Another typical script that has been tested is the kind which can deliver either a <form> or a reply to the form depending on whether a Submit button has been pressed.

```
<html>
<? if(isset($Submit)){
    ... //construct <html> reply page based on data from <form>
}else{
    ... //draw <form> on <html> page to elicit data from user
}?>
</html>
```

This style of scripting is quite common and the validator checks both types of <html> page in one operation.

In looking for public domain scripts to test, the popular phpMyAdmin application was selected. PhpMyAdmin is a system written in PHP for administering MySQL databases via a web interface. The PHP source is freely downloadable [19]. Two 800 line scripts were tested, one (db_details_structure.php) which displays a list of tables in a selected database and the other (tbl_properties_structure.php) which displays the properties of a selected table.

Testing these gave error messages from our validator which on investigation revealed instances of where a control structure did not deliver a complete tagged structure, e.g.

```
if($alternate)
```

```
        echo "<table><tr>...header style 1...</tr>";
    else
        echo "<table><tr>...header style 2...</tr>";
```

This was actually placed inside a function so the structure was

```
output_header($alternate);
...output table data...
echo "</table>";
```

By placing the building of the table data into a function it would be simple enough to change the structure to

```
if($alternate){
    echo "<table><tr>...header style 1...</tr>";
    output_table_data();
    echo "</table>";
}else{
    echo "<table><tr>...header style 2...</tr>";
    output_table_data();
    echo "</table>";
}
```

Thus a rewrite is possible and necessary in order to have this script validated by our system. It is a matter of judgement whether the benefits of the validation balance the programming style enforced on the script programmer.

While obvious limitations apply to the validator described in this paper it nevertheless is applicable to a significant proportion of existing scripts which form the stated target. It seems that many scripts used to generate WML and (x)HTML use the scripting language in a simple way. In this case 'simple' means that the

structure of the intended output is modelled in the structure of the script. This in turn means that, after parsing the script, a very simple strategy can be used to generate the expression which captures the generalised output of the script. This leaves the engine required to perform the generation of the expression closer to a parser than an interpreter.

As various limitations have been found with the prototype implementation it is reasonable to ask whether any alternative implementation strategies are possible. Consider the middle 'simplification' process. Chuang [20] has used ML to validate XML using WML as an example, by exploiting the parametric module facility. Hosoya and Pierce [21] report on their use of ML (via CAML) as a vehicle for regular expression pattern matching for XML. They create CAML functions directly from a DTD which perform a 'type check' on XML data that it is valid against the DTD. These methods still have the restriction that their input must be XML and so do not eliminate the current major obstacle with the middle process.

The limitations of the parsing strategy used to create tagged output from PHP scripts can only be completely removed by building or having access to a full PHP interpreter. Nevertheless even with these simple tools a validator has been built which is useful because it can fully validate a range of typical scripts. It is believed that this is the first attempt to build a validation tool for a script *per se*.

Although a script has been written to generate the rules needed to augment the DTD a tool to automate the substitutions within the original DTD is required. It is believed that some limited variable interpolation could be added fairly easily which would usefully extend the range of scripts which can be validated.

The validation technique that has been described (however implemented) can readily be applied to other scripting languages and any other target mark-up language that is specified as a sublanguage of XML via a DTD.

APPENDIX - CHANGES TO DTD

For each tag (**t**) mentioned in the DTD five new (derived) tags must be defined. They are **t_option**, **t_choice**, **t_choices**, **t_star**, **t_plus**.

(i) **t_option** : the optional **t** in **t_option** can be represented by structures that yield 0 or 1 units of **t**

```
<!ELEMENT t_option ( t | t_option | t_choices )>
```

(ii) **t_choice**: a **t_choice** structure can be represented by structures that yield exactly 1 unit of **t**

```
<!ELEMENT t_choice ( t | t_choices )+ >
```

(iii) **t_choices**: each choice of a **t_choices** structure can be represented by **t_choice**

```
<!ELEMENT t_choices ( t_choice , t_choice ) >
```

(iv) **t_list**: the elements of the possibly empty list **t_list** can be represented by structures that yield 0, 1 or more units of **t**

```
<!ELEMENT t_list ( t | t_option | t_choices | t_list | t_list1 )*>
```

(v) **t_list1**: the elements of the non-empty list **t_list1** can be represented by structures that yield 1 or more units of **t**

```
<!ELEMENT t_list1 ( t | t_choices | t_list1 )+>
```

Four entities are also defined for each tag (**t**), being **%t**;, **%t.opt**;, **%t.star**; and **%t.plus**;

The following replacements are made within the DTD

(...t...)	(... %t; ...)
(...t?...)	(... %t.opt; ...)
t* or (... t ...)*	%t.star; or (... %t.star; ...)*
t+ or (... t ...)+	%t.plus; or (... %t.plus; ...)+

The four entities are defined as follows:

(i) %t; - a single tag t can be represented by structures that yield exactly 1 unit of t

```
<!ENTITY % t "( t | t_choices )">
```

(ii) %t.opt; - an optional tag t? can be represented by structures that yield 0 or 1 units of t

```
<!ENTITY % t.opt "( t | t_option | t_choices )">
```

(iii) %t.star; - a possibly empty list t* can be represented by structures that yield 0 or more units of t

```
<!ENTITY % t.star "( t | t_option | t_choices | t_list | t_list1 )">
```

(iii) %t.plus; - a non empty list t+ can be represented by structures that yield 1 or more units of t

```
<!ENTITY % t.plus "( t | t_choices | t_list1 )">
```

Note: These entities can also be used to simplify the definitions of the 5 derived tags given earlier in the appendix.

REFERENCES/LINKS

1. W3C HTML Validator (free web service), <http://validator.w3.org/>

2. WDG HTML Validator (own free web service and links to other validators), <http://www.htmlhelp.com/>
3. CSE HTML Validator (commercial product), <http://www.htmlvalidator.com>
4. WML Validator, http://www.w3schools.com/wap/wml_validate.asp
5. JWIG, <http://www.brics.dk/JWIG>
6. CDuce, <http://www.cduce.org/>
7. XDuce, <http://xduce.sourceforge.net/>
8. Hypertext Preprocessor (PHP), <http://www.php.net/>
9. Microsoft Active Server Pages (ASP), <http://msdn.microsoft.com/asp/>
10. Practical Extraction & Report Language (PERL), <http://www.perl.com/>
11. Wireless Application Protocol (WAP), <http://www.w3schools.com/wap/>
12. Wireless Markup Language Document Type Definition (WML DTD),
http://www.wapforum.org/DTD/wml_1_1.dtd
13. YACC/LEX, Unix Programmers Manual (see also <http://dinosaur.compilertools.net/>)
14. XHTML Document Type Definition (Strict) (XHTML DTD), <http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd>

15. A Logic Programming Approach to Supporting the Entries of XML Documents in an Object Database, Ching-long Yey, PADL 2000: 278-292

16. SWI-Prolog, <http://www.swi-prolog.org/>

17. SWI-Prolog SGML/XML parser, <http://www.swi-prolog.org/packages/sgml2pl.html>

18. PHP syntax, <http://www.mare.ee/indrek/sablecc/php4.sablecc3.txt>

19. phpMyAdmin - MySQL database administration tool, <http://www.phpmyadmin.net/>

20. Generic Validation of Structural Content with Parametric Models, Chuang, T-R, Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP'01), ACM SIGPLAN Notices 36(10), Oct 2001

(http://portal.acm.org/ft_gateway.cfm?id=507649&type=pdf&coll=Portal&dl=GUIDE)

21. Regular Expression Pattern Matching for XML, Hosoya, H. & Pierce, B.C., Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp 67 - 80, 2001

(<http://portal.acm.org/citation.cfm?id=360209&dl=ACM&coll=portal>)