Static validation of dynamically generated HTML documents based on abstract parsing and semantic processing

Hyunha Kim[1,⋆], Kyung-Goo Doh[1,⋆], and David A. Schmidt[2,⋆⋆]

[1] Hanyang University, Ansan, South Korea
hhkim@plasse.hanyang.ac.kr, doh@hanyang.ac.kr
[2] Kansas State University, Manhattan, Kansas, USA
das@ksu.edu

Abstract. Abstract parsing is a static-analysis technique for a program that, given a reference LR(k) context-free grammar, statically checks whether or not every dynamically generated string output by the program conforms to the grammar. The technique operates by applying an LR(k) parser for the reference language to data-flow equations extracted from the program, immediately parsing all the possible string outputs to validate their syntactic well-formedness.
In this paper, we extend abstract parsing to do semantic-attribute processing and apply this extension to statically verify that HTML documents generated by JSP or PHP are always valid according to the HTML DTD. This application is necessary because the HTML DTD cannot be fully described as an LR(k) grammar. We completely define the HTML 4.01 Transitional DTD in an attributed LALR(1) grammar, carry out experiments for selected real-world JSP and PHP applications, and expose numerous HTML validation errors in the applications. In the process, we experimentally show that semantic properties defined by attribute grammars can also be verified using our technique.

Keywords: static analysis, string analysis, abstract parsing, HTML validation

1 Introduction

Most HTML documents viewed from the web are dynamically generated by scripts that mix dynamic input with static structure. As a result, many dynamically generated documents are grammatically malformed, and some even contain user-supplied attacks that exploit the malformedness [18, 19]. HTML validation tools are provided at the W3C site, but the tools are impractical or impossible

⋆ This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2012-0000469).
⋆⋆ Research partially supported by National Science Foundation Project, NSF CNS-1219746.

to use with scripts that dynamically generate HTML. Therefore, our goal is to validate, in advance of execution, the syntactic *and* semantic properties of the HTML documents generated dynamically by an application.

Since HTML-document structure is context-sensitive, we wish to employ parsing theory and semantic-analysis techniques from compiling theory to do validation. *Abstract parsing* does this [7, 8]: It extracts from a script a set of flow equations that overapproximate the documents (strings) that the script might generate, and it solves the equations in the domain of *LR-parse stacks*, which encodes the documents' context-free structure.

In this paper, we explain how we employ abstract parsing to validate JSP and PHP scripts. When our implementation is applied to a standard suite of JSP and PHP programs, we found it to be sound, precise (it yields very few false positives — false indications of errors), and reasonably efficient.

1.1 Motivating examples

We show two HTML-generated PHP scripts, the first generating syntactically invalid HTML, the second generating semantically invalid HTML:

Validating syntactic structure The following code shows a portion of a PHP program that generates one of two different HTML pages depending on the value of a conditional expression `isset($_POST["mode"])` determined at run-time.

```
<body>
  <table>
    <tr><th>
    ...
<?php
  if (isset($_POST["mode"])) {
    echo "<tr>";
    $result = DB_query(...);
    while($fruit = DB_fetch_array($result) {
       echo "<td>" . $fruit . "</td>";
    }
    echo "</table>";
  }
?>
  ...
```

If the conditional evaluates to true, the program always generates a syntactically valid page. `<table>` is required to be paired with `</table>`, which is the case in the generated page. It is acceptable that `<tr>` has no matching `</tr>`, because the HTML definition allows that `</tr>` be omitted. However, if the conditional evaluates to false, `</table>` is missing in the generated page, making the page syntactically invalid.

Validating semantic properties The following PHP program fetched from `DiscountCategories.php` in WEBERP always generates a form element if the conditional in the first line is true. In the form, a table is built using data retrieved from a database and contains a `submit` button.

```
if (isset($_POST['selectchoice'])) {
  echo '<form id="update" method="post" action=" ... "';
  echo '<div>'; echo '<input type="hidden" name="FormID" value="...">'';
  $sql = "SELECT DISTINCT discountcategory FROM stockmaster WHERE discountcategory <>''";
  $result = DB_query($sql, $db);
  if (DB_num_rows($result) > 0) {
    echo '<table class="selection"><tr><td>';
    echo '<select name="DiscCat" onchange="ReloadForm(update.select)">';
    while ($myrow = DB_fetch_array($result)){
        echo '... <option selected="selected" value="..."> ...';
    }
    echo '</select></td>';
    echo '<td><input type="submit" name="select" value="'._('Select').'" /></td>
       </tr></table><br />';
  }
  echo '</div></form>';
}
```

However, when nothing is retrieved from the database, the second conditional is false, no `submit` button is generated, and the result is a useless form that never transmits data. (When there is nothing to submit, no form should be generated.)

Our abstract-parsing technique will analyze and detect both forms of errors — both the syntax *and semantics* of the dynamically generated documents can be predicted prior to run-time.

1.2 Contributions

The contributions of this paper are

- We extend abstract-parsing with an implementation of semantic-attribute processing, which makes it amenable to a wide range of static-analysis problems on document-generating scripts.
- We define a complete LALR(1) attribute grammar for the HTML 4.01 DTD Transitional definition, a nontrivial task.
- We statically validate JSP and PHP programs that dynamically generate HTML documents, by submitting the HTML attributed grammar to the abstract parser equipped with semantic processing. The implementation statically validates all the features that W3C HTML Validator does dynamically, as well as semantic properties. Our earlier work shown in [7] was only able to validate a subset of HTML, essentially XHTML, the part definable in LALR(1) grammar.

The paper's next section summarizes abstract-parsing methodology (c.f. [7]), and Section 3 explains semantic processing, extending earlier work [8]. Section 4 explains the difficulties and our achievement of defining precisely an attributed grammar for the HTML DTD. Sections 5 and 6 present our work and our results of validating syntactic and semantic properties of scripts that dynamically generate HTML. Section 7 examines related research in the field, and Section 8 concludes.

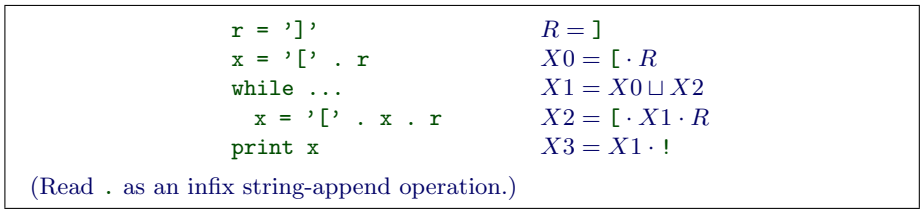


Fig. 1: Sample program and its data-flow equations

2 Fundamentals of abstract parsing

This section is a summary of [7], improved to support modular definitions. We present abstract parsing with an example: Say that a script must generate an output string that conforms to this grammar,

$$S \to \texttt{[]} \mid \texttt{[}S\texttt{]} \mid S\,S$$

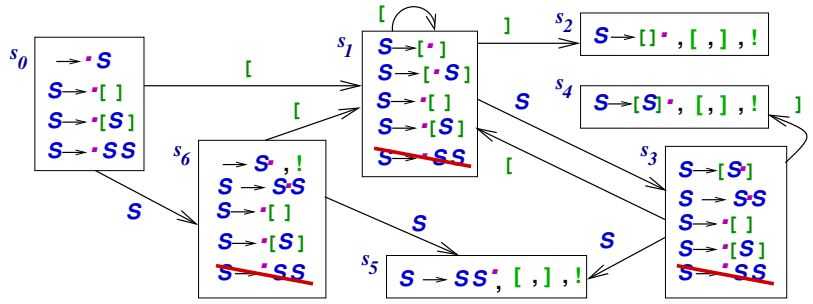
where S is the only nonterminal. (HTML, XML, and SQL are such bracket languages.) The grammar can be difficult to enforce even for simple programs, like the one in Figure 1, left column. Say this program must print only well-formed S-phrases.

Figure 1's right column shows the data-flow equations extracted from the program. Previous approaches have used type checking [3, 6, 17], regular expressions [4, 5, 12, 13], and language inclusion [14, 16, 15, 17], but all of these fail at some point to track precisely the context-free structure implicit in the string-valued document. For example, a standard regular-expression analysis solves the flow equations in the domain of regular expressions, determining that $X3$'s values conform to the regular expression, $\texttt{[}^* \cdot \texttt{[} \cdot \texttt{]} \cdot \texttt{]}^*$, which does not validate the demand. (It is possible to "jazz up" such an analysis [16, 15], but at some point, context-free structure is lost.)

We validate the desired property by solving the flow equations in Figure 1 in the domain of *LR-parse stacks* — $X3$'s meaning is the *set of parse stacks* of the strings that might be denoted by `x`. Our technique simultaneously unfolds and LR-parses the strings defined by $X3$, computing parse stacks that express structure in both the flow equations and the reference grammar. (Of course, a script might generate infinitely many different strings, and therefore the analysis might compute an infinite set of parse stacks. We finitely approximate an infinite set of parse stacks by exploiting a key feature of LR-parse theory, described in Section 2.2.)

First, let's understand the parser for the example grammar: Figure 2 gives the LALR(1)-parse-controller and parse of the string, `[[][]]`. The controller's transitions are coded as shift/reduce rewriting rules, which parse the string. The current state, $[s_i]$, of the parse appears at the top of the parse stack, $s_0 :: s_1 :: \cdots :: [s_i]$. Input symbols, i, are supplied to parse state s in the format, $[i \hookrightarrow s]$. The parser starts from the stack, $[s_0]$ and consumes the input string symbol by symbol, generating the parse in the Figure.

Input symbols label the transitions and ! denotes end of input:



Shift transitions:
$[\texttt{[} \hookrightarrow s_0] \Rightarrow s_0 :: [s_1]$
$[\texttt{[} \hookrightarrow s_1] \Rightarrow s_1 :: [s_1]$
$[\texttt{]} \hookrightarrow s_1] \Rightarrow s_1 :: [s_2]$
$[\texttt{[} \hookrightarrow s_3] \Rightarrow s_3 :: [s_1]$
$[\texttt{]} \hookrightarrow s_3] \Rightarrow s_3 :: [s_4]$
$[\texttt{[} \hookrightarrow s_6] \Rightarrow s_6 :: [s_1]$

Reduce transitions:
$s_i :: s_j :: [\ell \hookrightarrow s_2] \Rightarrow [\ell \hookrightarrow S \hookrightarrow s_i]$
$s_i :: s_j :: s_k :: [\ell \hookrightarrow s_4] \Rightarrow [\ell \hookrightarrow S \hookrightarrow s_i]$
$s_i :: s_j :: [\ell \hookrightarrow s_5] \Rightarrow [\ell \hookrightarrow S \hookrightarrow s_i]$
$s_i :: [\texttt{!} \hookrightarrow s_6] \Rightarrow s_i :: \textit{finished}$
where $\ell \in \{\texttt{[}, \texttt{]}, \texttt{!}\}$ and
$[\ell \hookrightarrow S \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_6]$
$[\ell \hookrightarrow S \hookrightarrow s_1] \Rightarrow s_1 :: [\ell \hookrightarrow s_3]$
$[\ell \hookrightarrow S \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_5]$
$[\ell \hookrightarrow S \hookrightarrow s_6] \Rightarrow s_6 :: [\ell \hookrightarrow s_5]$

parse stack	input sequence
$[s_0]$	`[[][]]!`
$[\texttt{[} \hookrightarrow s_0]$	`[][]]!` (ready for shift transition)
$s_0 :: [s_1]$	`[][]]!`
$s_0 :: [\texttt{[} \hookrightarrow s_1]$	`][]]!`
$s_0 :: s_1 :: [s_1]$	`][]]!`
$s_0 :: s_1 :: [\texttt{]} \hookrightarrow s_1]$	`[]]!`
$s_0 :: s_1 :: s_1 :: [s_2]$	`[]]!`
$s_0 :: s_1 :: s_1 :: [\texttt{[} \hookrightarrow s_2]$	`]]!` (ready for reduce transition)
$s_0 :: [\texttt{[} \hookrightarrow S \hookrightarrow s_1]$	`]]!` (reduce $S \to$ `[]`)
$s_0 :: s_1 :: [\texttt{[} \hookrightarrow s_3]$	`]]!`
$s_0 :: s_1 :: s_3 :: [s_1]$	`]]!`
$s_0 :: s_1 :: s_3 :: [\texttt{]} \hookrightarrow s_1]$	`]!`
$s_0 :: s_1 :: s_3 :: s_1 :: [s_2]$	`]!`
$s_0 :: s_1 :: s_3 :: s_1 :: [\texttt{]} \hookrightarrow s_2]$	`!`
$s_0 :: s_1 :: [\texttt{]} \hookrightarrow S \hookrightarrow s_3]$	`!` (reduce $S \to$ `[]`)
$s_0 :: s_1 :: s_3 :: [\texttt{]} \hookrightarrow s_5]$	`!`
$s_0 :: [\texttt{]} \hookrightarrow S \hookrightarrow s_1]$	`!` (reduce $S \to SS$)
$s_0 :: s_1 :: [\texttt{]} \hookrightarrow s_3]$	`!`
$s_0 :: s_1 :: s_3 :: [s_4]$	`!`
$s_0 :: s_1 :: s_3 :: [\texttt{!} \hookrightarrow s_4]$	
$[\texttt{!} \hookrightarrow S \hookrightarrow s_0]$	(reduce $S \to [S]$)
$s_0 :: [\texttt{!} \hookrightarrow s_6]$	
$s_0 :: \textit{finished}$	

Fig. 2: Disambiguated LALR(1) parser for $S \to$ `[ ]` $\mid$ `[`S`]` $\mid S\,S$, where $S\,S$ is made left associative

Our abstract parsing technique will apply the shift/reduce transition rules to the flow equations in the right column of Figure 1. To validate that the program prints only S-structured phrases at $X3$, we must evaluate the start stack, $[s_0]$, and (the string(s) denoted by) $X3$. We portray this as the function call, $X3[s_0]$ — *we treat the program-flow equations in Figure 1 as functions defined in combinator notation*, where we apply a function to the state used to parse it.

Starting from $X3[s_0]$, we use the flow equation, $X3 = X1$, to generate this calculation:

$$\begin{aligned} X3[s_0] &= (X1 \cdot \texttt{!})[s_0] \\ &= X1[s_0] \oplus \texttt{!} \end{aligned}$$

The first line says that the value of $X1 \cdot \texttt{!}$ must be parsed starting from $[s_0]$. The second line says that the string value of $X1$ is parsed first and the resulting parse stack, say, $s_0 :: s_i :: \cdots :: [s_j]$, is then used to parse $\texttt{!}$. (This will be $s_0 :: s_i :: \cdots :: \texttt{!}[s_j]$. The $\oplus$ operator is defined precisely below; for now, read $E_1(s) \oplus E_2$ as "$E_1(s)$ generates a parse stack whose top state is passed as the argument to E_2, which extends the stack.")

The call, $X1[s_0]$, generates this equation:

$$X1[s_0] = X0[s_0] \cup X2[s_0]$$

That is, the union of the parses of strings at $X0$ and $X2$ from s_0 must be computed. (*Important:* this computes a set of parse stacks, which must be finitely approximated in the implementation.) We consider first $X0[s_0]$:

$$\begin{aligned} X0[s_0] &= (\texttt{[} \cdot R)[s_0] = \texttt{[}[s_0] \oplus R = [\texttt{[} \hookrightarrow s_0] \oplus R \\ &= (s_0 :: [s_1]) \oplus R = s_0 :: (R[s_1]) \\ \text{and} \quad & R[s_1] = \texttt{]}[s_1] = [\texttt{]} \hookrightarrow s_1] = \{s_1 :: [s_2]\} \\ \text{so,} \quad & X0[s_0] = s_0 :: (R[s_1]) = \{s_0 :: s_1 :: [s_2]\} \end{aligned}$$

That is, the parse of $\texttt{[} \cdot R$ from $[s_0]$ generates the stack, $s_0 :: s_1 :: [s_2]$, which is one transition step from reducing $S \to \texttt{[]}$ (which occurs when the next input symbol is encountered and is verified in S's *follow set*).

The $\oplus$ is a "continuation operator": For parse stack, st, and combinator expression, E, define $st \oplus E = tail(st) :: E[head(st)]$. That is, stack st's top state feeds to E. (More generally, for a set of stacks, S, define $S \oplus E = \{tail(st) :: E[head(st)] \mid st \in S\}$.)

Next,

$$\begin{aligned} X2[s_0] &= (\texttt{[} \cdot X1 \cdot R)[s_0] = [\texttt{[} \hookrightarrow s_0] \oplus (X1 \cdot R) \\ &= (s_0 :: [s_1]) \oplus (X1 \cdot R) = s_0 :: (X1 \cdot R)[s_1] \\ &= s_0 :: (X1[s_1] \oplus R) \end{aligned}$$

The call to parse $X1$'s string from $[s_1]$ generates $X1[s_1] = X0[s_1] \cup X2[s_1]$ which in turn generates calls to $X0[s_1]$ and $X2[s_1]$. Here is the list of residual equations generated from the initial call, $X3[s_0]$:

$$\begin{array}{ll} X3[s_0] = X1[s_0] \oplus \texttt{!} & X2[s_0] = s_0 :: (X1[s_1] \oplus R) \\ X1[s_0] = X0[s_0] \cup X2[s_0] & X1[s_1] = X0[s_1] \cup X2[s_1] \\ X0[s_0] = \{s_0 :: s_1 :: [s_2]\} & X0[s_1] = \{s_1 :: s_1 :: [s_2]\} \\ R[s_1] \;\; = \{s_1 :: [s_2]\} & X2[s_1] = s_1 :: (X1[s_1] \oplus R) \end{array}$$

These equations will be solved by a least-fixed point calculation in the domain of sets of parse stacks. (That is, the meaning of each $X_i[s_j]$ computes to a set of stacks.)

More equations can and will be generated, in demand-driven fashion, during the fixed-point calculation. To show how this proceeds, we will solve the mutually recursive equations for $X1[s_0]$, $X2[s_0]$, $X1[s_1]$, and $X2[s_2]$, in stages:

$$X1_0[s_0] = X2_0[s_0] = X1_0[s_1] = X2_0[s_1] = \emptyset$$

$$\begin{aligned}
X1_1[s_0] &= \{s_0 :: s_1 :: [s_2]\} \\
X2_1[s_0] &= \emptyset \\
X1_1[s_1] &= \{s_1 :: s_1 :: [s_2]\} \\
X2_1[s_1] &= s_1 :: (X1_1[s_1] \oplus R) \\
&= s_1 :: s_1 :: s_1 :: R[s_2] \\
&\quad \text{where } R[s_2] = \texttt{[]} \hookrightarrow s_2] \\
&= s_1 :: s_1 :: s_1 :: \texttt{[]} \hookrightarrow s_2] \\
&= s_1[\texttt{[]} \hookrightarrow S \hookrightarrow s_1] \\
&= s_1 :: s_1 :: \texttt{[]} \hookrightarrow s_3] \\
&= \{s_1 :: s_1 :: s_3 :: [s_4]\}
\end{aligned}$$

$$\begin{aligned}
X2_2[s_0] &= s_0 :: (X1_1[s_1] \oplus R) \\
&= s_0 :: s_1 :: s_1 :: R[s_2] \\
&= \{s_0 :: s_1 :: s_3 :: [s_4]\} \\
X1_2[s_1] &= \{s_1 :: s_1 :: [s_2],\ s_1 :: s_1 :: s_3 :: [s_4]\} \\
X2_2[s_1] &= s_1 :: (X1_2[s_1] \oplus R) \\
&= \{s_1 :: s_1 :: s_1 :: R[s_2], \\
&\quad\ s_1 :: s_1 :: s_1 :: s_3 :: R[s_4]\} \\
&= \{s_1 :: s_1 :: s_3 :: [s_4]\} \\
X1_3[s_0] &= \{s_0 :: s_1 :: [s_2],\ s_0 :: s_1 :: s_3 :: [s_4]\} \\
X2_3[s_0] &= s_0 :: (X1_2[s_1] \oplus R) \\
&= \{s_0 :: s_1 :: s_3 :: [s_4]\}
\end{aligned}$$

At this point, the equations converge. Note that

$$X1[s_0] = \{s_0 :: s_1 :: [s_2],\ s_0 :: s_1 :: s_3 :: [s_4]\}$$

signifying that the parses of the value of `x` in the loop body come either from `[]` or from `[` $\cdot\, S \,\cdot$ `]`, where S represents a parse of some S-structured string. For this reason, we have

$$\begin{aligned}
X3[s_0] &= X1[s_0] \oplus \texttt{!} \\
&= \{s_0 :: s_1 :: \texttt{!}[s_2],\ s_0 :: s_1 :: s_3 :: \texttt{!}[s_4]\} \\
&= \{s_0 :: \mathit{finished}\}
\end{aligned}$$

This validates that the strings printed at the hot spot must be S-phrases. The algorithm that generates the residual equations and simultaneously solves them is a worklist algorithm like those used for demand-driven data-flow analysis [2, 9, 11].

2.1 Simplifying the calculation: Higher-order parse states

It is disappointing that the calculation of $X0[s_0]$ yields $\{s_0 :: s_1 :: [s_2]\}$ and not the nonterminal, S, since the assignment `x = '[' . r` assigns the string, `'[]'`, to `x`. The issue, of course, is that a lookahead symbol, ℓ, is required to validate the reduction of `'[]'` to S. This is formalized in the transitions stated in Figure 2:

$$\begin{aligned}
&s_i :: s_j :: [\ell \hookrightarrow s_2] \Rightarrow [\ell \hookrightarrow S \hookrightarrow s_i],\ \text{if } \ell \in \{\texttt{[}, \texttt{]}, \texttt{!}\} \\
&[\ell \hookrightarrow S \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_6]
\end{aligned}$$

If we make the current parse state "higher order" by parameterizing it on the lookahead symbol, we can simplify the situation — we use this variation of the above reduction transition:

$$s_i :: s_j :: [s_2] \Rightarrow [S_F \hookrightarrow s_i], \text{where } F = \{\texttt{[}, \texttt{]}, \texttt{!}\}$$

Conditional reduce transitions:

$s_i :: s_j :: [s_2] \Rightarrow [S_F \hookrightarrow s_i]$
$s_i :: s_j :: s_k :: [s_4] \Rightarrow [S_F \hookrightarrow s_i]$
$s_i :: s_j :: [s_5] \Rightarrow [S_F \hookrightarrow s_i]$
where $F = \{\texttt{[}, \texttt{]}, \texttt{!}\}$

Lookahead application transitions:

$[\ell \hookrightarrow S_F \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_6]$ if $\ell \in F$
$[\ell \hookrightarrow S_F \hookrightarrow s_1] \Rightarrow s_1 :: [\ell \hookrightarrow s_3]$ if $\ell \in F$
$[\ell \hookrightarrow S_F \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_5]$ if $\ell \in F$
$[\ell \hookrightarrow S_F \hookrightarrow s_6] \Rightarrow s_6 :: [\ell \hookrightarrow s_5]$ if $\ell \in F$

Reworked abstract parse of example program:

$$
\begin{aligned}
X0[s_0] &= \{s_0 :: s_1 :: [s_2]\} = \{[S_F \hookrightarrow s_0]\} \\
R[s_1] &= \{s_1 :: [s_2]\} \\
R[s_2] &= \{[\texttt{]} \hookrightarrow s_2]\} \\
R[s_4] &= \{[\texttt{]} \hookrightarrow s_4]\} \\
X1[s_0] &= X1[s_0] \oplus \texttt{!} = \{s_0 :: s_1 :: [s_2], s_0 :: s_1 :: s_3 :: [s_4]\} = \{[S_F \hookrightarrow s_0]\} \\
X2[s_0] &= s_0 :: (X1[s_1] \oplus R) = \{s_0 :: s_1 :: s_3 :: [s_4]\} = \{[S_F \hookrightarrow s_0]\} \\
X1[s_1] &= X0[s_1] \cup X2[s_1] = \{[S_F \hookrightarrow s_1]\} \\
X2[s_1] &= s_1 :: (X1[s_1] \oplus R) = \{[S_F \hookrightarrow s_1]\} \\
X3[s_0] &= X1[s_0] \oplus \texttt{!} = \texttt{!}[S_F \hookrightarrow s_0] = [\texttt{!} \hookrightarrow S_F \hookrightarrow s_0] = \{s_0 :: [\texttt{!} \hookrightarrow s_6]\} \\
&= \{s_0 :: \textit{finished}\}
\end{aligned}
$$

Fig. 3: Reformatted transition rules and reworked example

$[S_F \hookrightarrow s_i]$ is actually an abbreviation for $\lambda \ell \in F.\ [\ell \hookrightarrow S \hookrightarrow s_i]$.

The new rule reduces $s_0 :: s_1 :: [s_2]$ *before* the lookahead symbol arrives, conditionally on the value of the lookahead. The accompanying transition rule does application and validation:

$$[\ell \hookrightarrow S_F \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_6],\ \text{if } \ell \in F$$

Using the new rules, we calculate that

$$X0[s_0] = \{[S_F \hookrightarrow s_0]\}$$

That is, $X0$ generates an "S-typed" string and supplies it to s_0, conditional on the arrival of the lookahead symbol.

Figure 3 presents the higher-order variants of the reduction rules from Figure 2 and recalculates the abstract parse of the example program, producing more intuitive answers.

2.2 Finite convergence by stack folding

The previous example converged in finitely many calculation steps, but in general, an infinite set of parse stacks can be computed, e.g.,

```
x = '['                 X0 = [
while ...               X1 = X0 ⊔ X2
  x = x . '['           X2 = X1 · [
x = x . ']'             X3 = X1 · ] · !
```

At conclusion, `x` holds zero or more left brackets and an S-phrase. The analysis confirms this:

$$\begin{aligned}
X0[s_0] &= s_0 :: [s_1] \\
X1[s_0] &= \{s_0 :: s_1^i :: [s_1] \mid i \geq 0\} \\
X2[s_0] &= \{s_0 :: s_1^i :: [s_1] \mid i > 0\} \\
X3[s_0] &= \{(s_0 :: s_1^i :: [s_1]) \oplus \texttt{]} \oplus \texttt{!} \mid i \geq 0\} \\
&= \{s_0 :: s_1^i :: [\texttt{!} \hookrightarrow s_2] \mid i \geq 0\} \\
&= \{[\texttt{!} \hookrightarrow S \hookrightarrow s_0]\} \cup \{s_0 :: s_1^i :: [\texttt{!} \hookrightarrow S \hookrightarrow s_1] \mid i \geq 0\} \\
&= \{s_0 :: \mathit{finished}\} \cup \{s_0 :: s_1^j :: [\texttt{!} \hookrightarrow s_3] \mid j > 0\}
\end{aligned}$$

Since we want a finitely convergent analysis, we bound the infinite sets by "folding" their stacks so that no state repeats in a stack. Thus, the worklist algorithm calculates

$$\begin{aligned}
X0[s_0] &= s_0 :: [s_1] \\
X1[s_0] &= \{s_0 :: s_1^* :: [s_1]\} \\
X2[s_0] &= \{s_0 :: s_1^+ :: [s_1]\} \\
X3[s_0] &= \{s_0 :: \mathit{finished}, \quad s_0 :: s_1^* :: [\texttt{!} \hookrightarrow s_3]\}
\end{aligned}$$

Since the set of parse-state names is finite, folding produces a finite set of finite-sized stacks (that contain cycles). This works because each parse stack is a finite path through the LR-parser's finite-state controller, and folding a parse stack generates a (smallest) subgraph of the automaton that covers the path. Indeed, the subgraph can be represented by a regular expression, because it is a *viable prefix* [10] of the LR-parse.

Stack folding lets us generalize the abstract-parsing technique to arbitrary LALR(k) grammars with good success in practice.

3 Abstract semantic-processing

We now build on the proposal in [8] to implement a useful form of semantic processing. Since we can parse dynamically generated strings, we can predict their *semantics* as well by incorporating syntax-directed-translation (synthesized-attribute) techniques from compiling theory. For the bracket language,

$$S \to \texttt{[]} \mid \texttt{[}S\texttt{]} \mid S\,S$$

we might wish to track the *depth* at each point in a string as well as the *height* of each completed S-phrase. For example, for $(\alpha)\texttt{[[}(\beta)\texttt{]]}$, the depth at α is 0, the depth at β is 2, the height of `[]` is 1, and the height of the entire string is 2. The depth and height attributes typify the semantical information one must collect to validate HTML semantic properties, so we develop this example in detail.

Figure 4 gives a Madsen-Watt-style attribute grammar that defines depth and height, along with modified transition rules that compute the attributes, and also a calculation of the example string. All parse states are annotated

Semantic attributes : depth, d (inherited), annotates all parse states;
height, h (synthesized), annotates s_3, s_5, s_6.

attributes rules :

$$\rightarrow \downarrow 0\ S \uparrow d$$
$$\downarrow d\ S \uparrow 1 \rightarrow \texttt{[ ]}$$
$$\downarrow d\ S \uparrow h+1 \rightarrow \texttt{[} \downarrow d\ S \uparrow h\ \texttt{]}$$
$$\downarrow d\ S \uparrow max(h,h') \rightarrow \downarrow d\ S \uparrow h \quad \downarrow d\ S \uparrow h'$$

Attributed Shift transitions:

$[\texttt{[} \hookrightarrow s_0^d] \Rightarrow s_0^d :: [s_1^{d+1}]$
$[\texttt{[} \hookrightarrow s_1^d] \Rightarrow s_1^d :: [s_1^{d+1}]$
$[\texttt{]} \hookrightarrow s_1^d] \Rightarrow s_1^d :: [s_2^{d-1}]$
$[\texttt{[} \hookrightarrow s_3^d] \Rightarrow s_3^d :: [s_1^{d+1}]$
$[\texttt{]} \hookrightarrow s_3^d] \Rightarrow s_3^d :: [s_4^{d-1}]$
$[\texttt{[} \hookrightarrow s_6^d] \Rightarrow s_6^d :: [s_1^{d+1}]$

Attributed reduce transitions:

$s_i :: s_j :: [l \hookrightarrow s_2] \Rightarrow [l \hookrightarrow S^1 \hookrightarrow s_i]$
$s_i :: s_j :: s_k^h :: [l \hookrightarrow s_4] \Rightarrow [l \hookrightarrow S^{h+1} \hookrightarrow s_i]$
$s_i :: s_j^h :: [l \hookrightarrow s_5^{h'}] \Rightarrow [l \hookrightarrow S^{max(h,h')} \hookrightarrow s_i]$
$s_i :: [\texttt{!} \hookrightarrow s_6^h] \Rightarrow s_i^h :: finished$
where $l \in \{\texttt{[}, \texttt{]}, \texttt{!}\}$ and
$[l \hookrightarrow S^h \hookrightarrow s_0^d] \Rightarrow s_0^d :: [l \hookrightarrow s_6^{d,h}]$
$[l \hookrightarrow S^h \hookrightarrow s_1^d] \Rightarrow s_1^d :: [l \hookrightarrow s_3^{d,h}]$
$[l \hookrightarrow S^h \hookrightarrow s_3^d] \Rightarrow s_3^d :: [l \hookrightarrow s_5^{d,h}]$
$[l \hookrightarrow S^h \hookrightarrow s_6^d] \Rightarrow s_6^d :: [l \hookrightarrow s_5^{d,h}]$
(Read ! as symbol of end of string)

parse stack (top lies at right)	input sequence (front lies at left)
$[s_0^0]$	[[][]!
$[\texttt{[} \hookrightarrow s_0^0]$	[]]! (ready for shift transition)
$s_0^0 :: [s_1^1]$	[]]!
$s_0^0 :: [\texttt{[} \hookrightarrow s_1^1]$	]]!
$s_0^0 :: s_1^1 :: [s_1^2]$	]]!
$s_0^0 :: s_1^1 :: [\texttt{]} \hookrightarrow s_1^2]$	]!
$s_0^0 :: s_1^1 :: s_1^2 :: [s_2^1]$	]!
$s_0^0 :: s_1^1 :: s_1^2 :: [\texttt{[} \hookrightarrow s_2^1]$	]! (ready for reduce transition)
$s_0^0 :: [\texttt{[} \hookrightarrow S^1 \hookrightarrow s_1^1]$	]! (reduce $S \rightarrow$ [])
$s_0^0 :: s_1^1 :: [\texttt{[} \hookrightarrow s_3^{1,1}]$	]!
$s_0^0 :: s_1^1 :: s_3^{1,1} :: [s_1^2]$	]!
$s_0^0 :: s_1^1 :: s_3^{1,1} :: [\texttt{]} \hookrightarrow s_1^2]$	!
$s_0^0 :: s_1^1 :: s_3^{1,1} :: s_1^2 :: [s_2^1]$	! (reduce $S \rightarrow$ [])
$s_0^0 :: s_1^1 :: [\texttt{]} \hookrightarrow S^1 \hookrightarrow s_3^{1,1}]$	!
$s_0^0 :: s_1^1 :: s_3^{1,1} :: [\texttt{]} \hookrightarrow s_5^{1,1}]$	! (reduce $S \rightarrow SS$)
$s_0^0 :: [\texttt{]} \hookrightarrow S^1 \hookrightarrow s_1^1]$	!
$s_0^0 :: s_1^1 :: [\texttt{]} \hookrightarrow s_3^{1,1}]$	!
$s_0^0 :: s_1^1 :: s_3^{1,1} :: [s_4^1]$	
$s_0^0 :: s_1^1 :: s_3^{1,1} :: [\texttt{!} \hookrightarrow s_4^1]$	
$[\texttt{!} \hookrightarrow S^2 \hookrightarrow s_0^0]$	(reduce $S \rightarrow$ [S])
$s_0^0 :: [\texttt{!} \hookrightarrow s_6^{0,2}]$	
$s_0^{0,2} :: finished$	

Fig. 4: $S \rightarrow \texttt{[]} \mid \texttt{[}S\texttt{]} \mid S\,S$, annotated with depth and height *concrete* semantic attributes

with a depth attribute, d, since all parse points within the string possess depth. Nonterminals, S, are annotated with a height attribute, h, since a well-formed S-phrase has height: the height attributes are attributes of parse states, s_3, s_5, s_6, since these states are reached by transitions labelled by S.

As noted earlier, an LR(1) state has form, $[\ell_1 \hookrightarrow \ell_0 \hookrightarrow s]$, where s is the parser state, ℓ_0 the input, and ℓ_1 the lookahead. When a reduce transition occurs, the corresponding semantic rule is computed. For the example bracket language, `[[][]]`, the computed result is $height = 2$, as expected.

Of course, precision of semantic attributes can be affected by stack folding, but as demonstrated in the following sections, loss of precision has not been a significant problem in practice.

4 Attributed LR(1) grammar for the HTML DTD

The W3C recommends every HTML document be validated according to the DTD (Document Type Definition). But the commonly used standard, HTML 4.01 Transitional DTD [1], cannot be defined in LALR(1); indeed, some parts are not LR(k) and are even ambiguous. We now review trouble spots in the HTML DTD and explain how we handled them with a synthesized-attribute-based LALR(1)-grammar.

4.1 Unordered occurrences of elements

In a `HEAD` element, its contents, `TITLE`, `ISINDEX` and `BASE`, may appear in any order, with the restrictions that `TITLE` must appear once, and `ISINDEX` and `BASE` may appear once or none:

```
<!ELEMENT HEAD O O (%head.content;) +(%head.misc;) >
<!ENTITY % head.content "TITLE & ISINDEX? & BASE?">
<!ENTITY % head.misc  "SCRIPT|STYLE|META|LINK|OBJECT" -- repeatable head elements -->
<!ELEMENT TITLE - - (#PCDATA) -(%head.misc;) -- document title -->
<!ELEMENT ISINDEX - O EMPTY -- single line prompt -->
<!ELEMENT BASE - O EMPTY -- document base URI -->
```

The tag inclusion `+(%head.misc;)` indicates that elements in `head.misc` can appear in `HEAD`. However, the declarations of `TITLE`, `ISINDEX`, and `BASE` prevent elements in `head.misc` from propagating inside `head.content`. The `TITLE` element excludes `head.misc`, and the `ISINDEX` and `BASE` elements have their bodies empty. Due to the unorderedness of `head.content`, a LALR(1)-grammatical expansion would grow exponentially, so we utilized synthesized attributes instead: An attribute tag for each of three elements counts the occurrences of its element and checks if the number of occurrences falls within the boundaries. The synthesized-attribute LALR(1) grammar is defined in Figure 5.

4.2 Tag inclusion and exclusion

Tag inclusion, `+(A)`, which is an SGML feature, signifies that element A may appear anywhere within its defining element. There are only two occurrences of tag inclusion in HTML DTD, one of which is the following:

production		semantic rules
head	→ $\mathsf{head}_\circ{}^?$	
	contents	{ $(t, b, i) = \mathit{contents}.\mathit{count}$;
		check $t == 1 \wedge 0 \leq b \leq 1 \wedge 0 \leq i \leq 1$; }
	$\mathsf{head}_\bullet{}^?$	
contents	→ $\mathit{contents}_1$	
	content	{ $\mathit{contents}.\mathit{count} = \mathit{contents}_1.\mathit{count} +\!+\!+ \mathit{content}.\mathit{count}$ }
contents	→ *content*	{ $\mathit{contents}.\mathit{count} = \mathit{content}.\mathit{count}$ }
content	→ *title*	{ $\mathit{content}.\mathit{count} = (1,0,0)$ }
content	→ *base*	{ $\mathit{content}.\mathit{count} = (0,1,0)$ }
content	→ *isindex*	{ $\mathit{content}.\mathit{count} = (0,0,1)$ }
content	→ *misc*	{ $\mathit{content}.\mathit{count} = (0,0,0)$ }

where the attribute *count* is *(t,b,i)*:
$\mathsf{head}_\circ$ is start tag of `HEAD` element — t is the number of `TITLE` elements
$\mathsf{head}_\bullet$ is end tag of `HEAD` element — b is the number of `BASE` elements
i is the number of `ISINDEX` elements
and $(t_1, b_1, i_1) +\!+\!+ (t_2, b_2, i_2) = (min(2, t_1 + t_2), min(2, b_1 + b_2), min(2, i_1 + i_2))$

Fig. 5: Attribute grammar for head elements

```
<!ELEMENT BODY O O (%flow;)* +(INS|DEL) >
<!ELEMENT (INS|DEL) - - (%flow;)* -- inserted text, deleted text -->
```

That is, `INS` and `DEL` elements may appear anywhere in `BODY` element. This is not directly definable in LALR(1), so we manually expanded the grammar by adding production rules for `INS` and `DEL` to every nested element.

The tag exclusion, `-(A)`, which is another SGML feature, signifies that the element `A` cannot appear in the defining element. For example, consider the following declaration of anchor element `A`:

```
<!ELEMENT A - - (%inline;)* -(A)>
```

`-(A)` indicates that the element `A` cannot be nested. A simple-minded construction of LALR(1) grammar for tag exclusion results in an exponentially large number of productions, and thus we chose to use synthesized attributes in Figure 6.

4.3 Validation of attributes in an HTML element

Attributes[1] in each HTML element have to be validated according to the `ATTLIST` declaration, where for each attribute, defined are its name, its type, and whether it is required or implied. We employed synthesized-attribute semantic processing to validate attributes in an HTML element. A global environment for attributes are constructed from element declarations. We get the necessary information about the attributes from the global environment as follows:

– $\mathsf{defined}(\mathsf{n}_\circ)$: the set of all attribute names in $\mathsf{n}_\circ$

[1] The reader should be careful not to confuse HTML "attributes" with the synthesized attributes used by the abstract parser.

production			semantic rules
a	$\rightarrow$	$\mathsf{a}_\circ$	
		$inlines$	{ check $\mathsf{name}(\mathsf{a}_\circ) \notin inlines.names$ }
		$\mathsf{a}_\bullet$	{ $a.names = \{ \mathsf{name}(\mathsf{a}_\circ) \} \cup inlines.names$ }
$inlines$	$\rightarrow$	$inlines_1$	
		$inline$	{ $inlines.names = inlines_1.names \cup inline.names$ }
$inlines$	$\rightarrow$	$inline$	{ $inlines.names = inline.names$ }
n	$\rightarrow$	$\mathsf{n}_\circ$	
		$some$	{ $n.names = \{ \mathsf{name}(\mathsf{n}_\circ) \} \cup some.names$ }
		$\mathsf{n}_\bullet{}^?$	

$\mathsf{name}(\mathsf{a}_\circ)$: element name of $\mathsf{a}_\circ = \mathsf{a}$

Fig. 6: Attribute grammar for Tag exclusion

- well-typed(a,$\mathsf{n}_\circ$) : the value of attribute a in $\mathsf{n}_\circ$ is well-typed
- required($\mathsf{n}_\circ$) : the set of all required attribute names in $\mathsf{n}_\circ$, where $\forall \mathsf{n}_\circ$. required($\mathsf{n}_\circ$) $\subseteq$ defined($\mathsf{n}_\circ$)

For example, consider the following attribute definition of `PARAM`:

```
<!ELEMENT PARAM - O EMPTY>
<!ATTLIST PARAM
  id          ID              #IMPLIED
  name        CDATA           #REQUIRED
  value       CDATA           #IMPLIED
  valuetype   (DATA|REF|OBJECT) DATA
  type        %ContentType;   #IMPLIED>
```

- defined($\mathsf{param}_\circ$) = {`id`,`name`,`value`,`valuetype`,`type`}
- well-typed(`valuetype`,$\mathsf{param}_\circ$) = *true*
  if the value of `valuetype` in $\mathsf{param}_\circ$ is among {`DATA`,`REF`,`OBJECT`}
- required($\mathsf{param}_\circ$) = {`name`}

Semantic rules for validating attributes in an element are defined as follows:

production	semantic rules
$n \rightarrow \mathsf{n}_\circ$	{ check $\forall a \in \mathsf{parsed}(\mathsf{n}_\circ).\ a \in \mathsf{defined}(\mathsf{n}_\circ)$;
	check $\forall a \in \mathsf{parsed}(\mathsf{n}_\circ).$ well-typed(a,$\mathsf{n}_\circ$);
	check $\forall a \in \mathsf{required}(\mathsf{n}_\circ).\ a \in \mathsf{parsed}(\mathsf{n}_\circ)$; }
	where $\mathsf{parsed}(\mathsf{n}_\circ)$ is the set of parsed attribute names in $\mathsf{n}_\circ$

Each semantic rule for the production $n \rightarrow \mathsf{n}_\circ$ asserts the following for attributes in $\mathsf{n}_\circ$:

- every parsed attribute name is declared
- every parsed attribute value is well-typed
- every required attribute is present

5 Experiments: Static HTML validation

Applying abstract parsing enhanced with attribute grammars, we implemented a static validator for JSP and PHP scripts. The architecture of our platform is

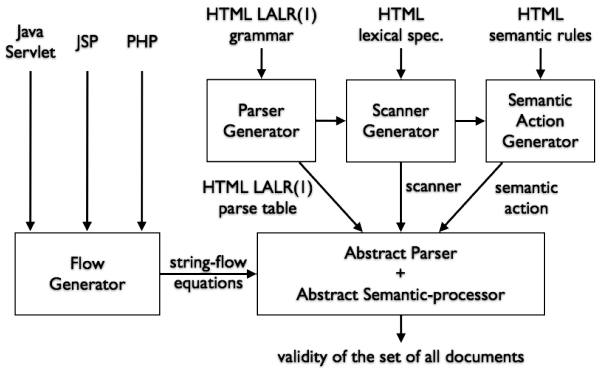


Fig. 7: Architecture of static HTML validator

shown in Figure 7. Java Servlets, JSP pages and PHP scripts are converted to sets of flow equations, and the LALR(1) grammar for HTML 4.01 Transitional DTD is given to `ocamlyacc`, generating its parsing table. A lexical specification is given to `ocamllex`, generating a scanner. Semantic rules are given to the semantic-action generator, emitting semantic actions. All of these are forwarded to the static validator, which is a generic abstract parser equipped with a semantic-attribute processor. The static validator analyzes all the documents generated by the input program. When a validation error occurs, the original position of the source that causes the error is returned. The entire implementation is written in Objective Caml.

We experimented with our static validator on a suite of JSP programs (the same one as Møller and Schwarz [15]) and PHP programs. The experiment was done with Mac OS X 10.8.2 Mountain Lion with Intel Core 2 Duo processor (2.56GHz) and 8GB memory. The results are summarized in Table 1.

The execution time measured is the total time used to validate each set of valid programs after all errors found are fixed manually. The average running time for each program is approximately one second except one case (WebChess, in PHP) averaged close to 5 seconds. Building the LALR(1) parse table takes only a few seconds and is not counted in the analysis time.

The details of detected errors are summarized in Table 2. The errors are classified into three groups: *tag matching, misplaced element*, and *attributes*. There are some false positives (only in PHP programs) that are all due to the lack of path-sensitive analysis. The number of false positives is shown inside parentheses.

5.1 Tag matching

Missing matched tag errors are abundant. Examples are `<strong>` with no matching `</strong>` (unmatched start/end tag) and `<h3>` followed by `</h2>` (mismatched tags). More serious errors are "improperly nested tags" as follows:

	program	Files	SLOC	Errors	FP	Time
JSP	Pebble	117	41,893	36	0	118.0s
	Bookstore1	6	919	8	0	6.2s
	Bookstore2	7	532	5	0	7.0s
	Bookstore3	11	753	5	0	11.0s
	Bookstore4	6	279	3	0	6.0s
	Bookstore5	7	249	6	0	7.1s
	Bookstore6	8	1,960	1	0	7.9s
	JSP Chat	14	920	21	0	16.1s
	JPivot	7	635	0	0	7.0s
	JSTL Book	53	1,457	18	0	52.7s
PHP	Schoolmate	65	6,470	149	0	62s
	FaqForge	19	940	68	0	10s
	WebChess	24	2,906	11	2	106s
	HGB	20	645	92	0	27s
	WEBERP	572	183,511	600	54	590s

Table 1: Summary of experimental results

categories	errors	JSP						PHP					
		A	B	C	D	E	total	F	G	H	I	J	total
tag matching	unmatched start/end tag	3	16	2	2	0	23	4	57	2(2)	14	145(16)	268(18)
	mismatched tags	0	0	1	0	0	1	0	0	0	0	1	3
	improperly nested tags	0	9	6	2	0	17	0	0	0	0	4	38
misplaced element	no `TITLE` in `HEAD`	0	0	0	0	0	0	0	0	0	2	38	40
	misplaced `</head>`	0	0	0	0	0	0	0	0	0	1	2	3
	misplaced `<body>`	0	0	0	0	0	0	1	0	0	0	4	5
	misplaced `</body>`	0	0	0	0	0	0	1	0	0	0	2	3
	misplaced `</html>`	0	0	0	0	0	0	0	0	0	0	2	2
	illegal appearance of *blocks* in `P`	16	0	0	0	0	16	0	0	3	0	5	40
	illegal appearance of *blocks* in `FONT`	0	0	0	0	0	0	0	0	0	3	0	3
	illegal appearance of `NOFRAMES`	0	0	1	0	0	1	0	0	0	0	0	2
	illegal appearance of `LINK`	0	0	0	0	0	0	0	0	2	0	0	2
	illegal appearance of `META`	0	0	0	0	0	0	0	0	0	0	7	7
	improperly closed `FORM`	0	0	0	0	0	0	1	3	0	5	4	13
	illegal appearance of `HTML` before `HTML`	0	0	0	0	0	0	0	0	0	0	1	1
	illegal appearance of `TABLE` before `BODY`	0	0	0	0	0	0	0	1	0	0	0	1
	no `TR` in `TABLE`	0	1	1	0	0	2	0	0	0	0	0	4
	missing `TR` outside `TD` or `TH`	2	0	0	0	0	2	0	0	0	3	13(6)	20(6)
	missing `TD` or `TH` in `TR`	4	0	1	0	0	5	0	1	0	0	13(3)	24(3)
	missing both `TR` and `TD`(or `TH`)	0	0	0	0	0	0	1	1	0	6	41	49
	improperly missing `<tbody>`	4	1	0	0	0	5	0	0	0	0	0	10
	no `OPTION`s in `SELECT`	0	0	0	0	0	0	31	0	1	0	282(29)	314(29)
	nonstandard element name	0	0	0	2	0	2	20	0	1	0	0	25
attributes	missing required attributes	5	1	6	9	0	21	24	3	1	6	14	90
	using undefined attributes	1	0	3	3	0	7	15	1	0	26	2	58
misc.	lexical errors	1	0	0	0	0	1	0	2	0	0	10	14
total		36	28	21	18	0	103	98	69	10(2)	66	590(54)	1039(56)

A = Pebble, B = Bookstore, C = JSP Chat, D = JSTP Book, E = JPivot,
F = Schoolmate, G = FaqForge, H = WebChess, I = HGB, J = WEBERP.

$m(n)$ means that n of m errors are false positives

blocks = block-level elements

Table 2: Classification of errors in JSP and PHP programs

– ...
– <tr><form><td> ... </td></form><tr>

These should have been generated respectively as follows:

– ...
– <tr><td><form> ... </form></td><tr>

5.2 Misplaced element

Block-level elements, such as TABLE, FORM, DIV, and UL, in P element are detected as errors, e.g., <table> ... </table> .

Errors related to TABLE elements are also found. According to TABLE DTD, a TABLE element should contain at least one TBODY or TR element and a TR element should contain at least one TD or TH element. The followings are the detected example patterns that violate the DTD:

– <table> text : both TR and TD are missing
– <table><td> text : TR is missing
– <table><tr> text : TD is missing

The correct pattern should have been: <table><tr><td> text. The requirement is: If TABLE element contains either THEAD or TFOOT, TBODY element cannot be omitted. Errors in Pebble 2.6.2 have the following common pattern:

– <table><thead> ... </thead> <tr> ... </table>

which should have been written as follows:

– <table><thead> ... </thead> <tbody> <tr> ... </table>

According to the HTML DTD, a SELECT element must contain at least one OPTION. The following program excerpted from WEBERP violates this when the loop is not executed:

```
$result = execute_query("SELECT ... ");
echo "<select>";
while($result) {
  ... echo "<option> ...";
}
echo "</select>"
```

Some programs carefully avoid this by filtering out empty data as follows:

```
if (DB_num_rows($result) == 0) then {
  ...
} else {
  echo "<select>";
  while($result) {
    ... echo "<option> ...";
  }
  echo "</select>"
}
```

Our tool falsely decides this situation is an error due to its ignorance of conditional expressions.

5.3 Attributes in HTML elements

Our tool detected multiple misuses of attributes in HTML elements. The `TD` element has an undefined attribute `background` as follows:

– `<td class='b' width=10 background='./images/left.gif'>`

The `TEXTAREA` has no required attributes `rows` and `cols`:

– `<textarea name='task'>`

6 Static validation of semantic properties

Additional semantic requirements, beyond those described in the DTD, are abundant in the HTML specification and listed in natural language. We carefully chose several critical semantic properties and specified them in an attributed LALR(1) grammar and then supplied the grammar to our static analyzer based on attributed-abstract parsing.

The semantic errors found are classified in Table 3. False positives here are also due to the lack of path-sensitive analysis. We examine the table in detail in the following subsections.

errors	WebChess	HGB	WEBERP	total
non-unique `id` attribute	0	8	1(1)	9(1)
unmatched `id` and `name` in a single element	0	0	4	4
`href` or `hrefs` refer undefined identifier	0	0	0	0
unsubmittable `FORM` field	10(7)	4(4)	26	40(11)

Table 3: Classification of semantic errors in PHP programs

6.1 Properties of element identifiers

According to the HTML 4.01 Specification, element identifiers must have the following properties:

– the value of `id` attribute must be unique in a document
– the values of `id` and `name` must be the same when both appear in an element's start tag
– the values of `href` and `hrefs` attributes should refer to defined identifiers in the same document

production			semantic rules

element	→	element$_\circ$	{ if element$_\circ$(id) is given then
			check element$_\circ$(id) $\notin$ *element.idset*(id);
			if element$_\circ$(name) is given then
			check element$_\circ$(id) == element$_\circ$(name);
			element.idset = *element.idset* $\cup$ {element$_\circ$(id)};
			if element$_\circ$(href) is given then
			element.hrefset = *element.hrefset* $\cup$ {element$_\circ$(href)};
			if element$_\circ$(hrefs) is given then
			element.hrefset = *element.hrefset* $\cup$ element$_\circ$(hrefs); }

		contents element$_\bullet^?$	
document	→	*html*	{ *html.idset* = $\emptyset$; *html.hrefset* = $\emptyset$; }
			{ check $\forall$id $\in$ *html.hrefset*, id $\in$ *html.idset* }

Fig. 8: attribute grammar for checking properties of element indentifiers

Figure 8 shows an attribute grammar for checking the above properties.

Errors found in HGB are all from `header.php` originate from eight redundant uses of the same value, `tl`, as follows:

```
// hgb/header.php
<?php if($block === false){ ?>
<div align=center>
<a id=tl href="./admin.php">Admin HOME</a> || <a id=tl href="filter.php">Spam Filter</a> ||
<a id=tl href="ipblock.php">IP Blocker</a> || <a id=tl href="passwrd.php">Change Password</a> ||
<a id=tl href="about.php?out=signout">Sign out</a><br>
<a id=tl href="url.php">Properties</a> || <a id=tl href="about.php">About</a> ||
<a id=tl href="readme.php">Read me</a> ||  <a id=tl target="_blank"  ...
```

An error found in `AccountGroups.php` of WEBERP is a false positive: Two conditional branches share the same value `AccountGroups` of `id`, but only one branch of the two will be executed, i.e., if one is executed, the other isn't. Since our analyzer does not take into account the meaning of conditional, it announces an error.

```
// WEBERP/AccountGroups.php
 ...
} elseif (isset($_GET['delete'])) {
  ...
  if ($myrow['groups']>0) {
    echo '... <br /><form method="post" id="AccountGroups"
      action="' . htmlspecialchars($_SERVER['PHP_SELF'], ENT_QUOTES, 'UTF-8') . '"> ...';
  }
}
...
if (!isset($_GET['delete'])) {
  echo '<form method="post" id="AccountGroups"
      action="' . htmlspecialchars($_SERVER['PHP_SELF'], ENT_QUOTES, 'UTF-8') . '">';
```

6.2 Submission of `FORM` fields

A `FORM` field only transfers its data when one of the following conditions is true:

– it contains at least one `INPUT` element whose `type` is `submit` or `image`,

– it contains one and only `INPUT` element whose `type` is `text`,
– it contains a `BUTTON` element whose `name` is `submit`.

Another way of transferring data is to use the `submit()` function of JavaScript. An attribute grammar for validating `FORM` data submission is defined as follows:

production		semantic rules
form	→ form$_\circ$ *contents* form$_\bullet$	{ *form*.*submittable* = false; *form*.*textcount* = 0; } { check *contents*.*submittable* ∨ *contents*.*textcount* == 1 }
input	→ input$_\circ$	{ if input$_\circ$(disabled) ≠ `true` ∧ input$_\circ$(type) ∈ { `submit`, `image` } then *input*.*submittable* =true; if input$_\circ$(disabled) ≠ `true` ∧ input$_\circ$(type) = `text` then *input*.*textcount* = $min(2, \mathit{input}.\mathit{textcount} + 1)$; }
button	→ button$_\circ$	{ if button$_\circ$(disabled) ≠ `true` ∧ button$_\circ$(type) = `submit` then *button*.*submittable* =true; }
	...	

submittable is a synthesized attribute becoming true when one of the first and third conditions above is true. *textcount* is also a synthesized attribute counting the number of text elements. Note that the domain of these attribute values are finite. The value of *textcount* is one of 0, 1, and 2.

Eleven errors are classified as false positives because all use JavaScript function `submit()` to submit `FORM` field data and JavaScript code itself is not analyzed by the tool. For instance,

```
print("<script language='JavaScript'>
  function schoolInfo() {
    document.admin.page2.value=1;
    document.admin.submit();
  } ... </script>");
...
  print("...
   <form name='admin' action='./index.php' method='POST'>
     <a class='menu' href='javascript: schoolInfo();' ... >School</a>
     ...
     <input type='hidden' name='page2' value='$page2'>
     <input type='hidden' name='logout'>
     <input type='hidden' name='page' value='$page'>
   </form> ...");
```

Three hidden input fields are submitted by function `schoolInfo()`, the first link of `A` in `FORM`. However, if JavaScript is unsupported or disabled in a web browser, the submission would not be working, hence they might well be classified as true positives. An additional analysis of JavaScript would remedy this problem.

7 Related research

Because of the popularity of HTML-document generators there exist a variety of approaches for static validation.

Minamide's initial efforts used data-flow equations to approximate the documents generated from PHP programs and then treats the equations as a grammar, matching it against an HTML/XHTML grammar [13]. However, since the

language inclusion problem for context-free grammar is undecidable, nesting depth of elements must be bounded, making the approach miss errors.

Later, Minamide's and Møller's groups independently developed sound methods of validating dynamically generated XML documents based on balanced grammars [12, 14], but their methods are difficult to generalize to HTML features such as tag omission and inclusion/exclusion. Some improvement has been made by Nishiyama and Minamide, who translate a subclass of the SGML DTD (including HTML) into a regular hedge grammar, avoiding undecidability [16]. However, this method does not support start tag omission and tag inclusion, and the translation to support exclusion causes exponential blowup of the grammar.

Recently, Møller and Schwarz developed an HTML validation algorithm [15] that is a generalization of the core algorithm for SGML parsing to work on context-free-grammar representation of documents. The approach is stated sound, precise, and efficient, and handles tag omissions and inclusions/exclusions; it is comparable to our work, limited to the extent of JSP validation. The comparison of JSP experimental results of ours and theirs (what is in the paper) reveals that ours finds more errors in J2EE Bookstore. We also located errors (in 4 pages from Bookstore 2) that are not mentioned in their paper, as follows:

- bookcashier.jsp : unmatched `</strong>` at line 32
- bookcatalog.jsp :
  - improperly nested tag `<p><strong>` at line 62 and `</p></strong>` at line 66
  - unmatched `<strong>` at 117 line
- booldetails.jsp : improperly nested tag `<p><strong>` at line 60 and `</p></strong>` at line 73
- bookshowcart.jsp : unmatched `</td>` at line 143

Extending the SGML parsing algorithm to handle semantic-attribute processing remains to be seen. Interestingly, our tool found no errors in JPivot, whereas Møller and Schwarz's tool found errors in 2 pages out of 3. The possible explanation might be that our tool skipped one JSP page that generates documents through XSL transformation, which our JSP-to-Java translator has yet to handle.

8 Conclusion

We have demonstrated the utility of marrying parsing, semantic processing, and data-flow analysis in the form of attributed abstract parsing, which can predict, parse, and semantically process with surprising accuracy the documents dynamically generated by scripts. The application domain described here, JSP and PHP scripts that generate HTML documents that conform with the HTML 4.01 Transitional DTD, demonstrates the feasibility of the approach.

Acknowledgements We thank anonymous referees for valuable suggestions and comments.

References

[1] HTML 4.01 Transitional DTD W3C Recommendation 24 December 1999 `http://www.w3.org/TR/html4/loose.dtd`

[2] G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proc. International Conference on Software Maintenance*, Oxford, 1999.

[3] C. Brabrand, A. Møller, and M.I. Schwartzbach. The `<bigwig>` project. *ACM Transaction on Internet Technology*, 2, 2002.

[4] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *Proc. Asian Symposium on Programming Languages and Systems*, pages 374-388. Springer LNCS 4279, 2006.

[5] A.S. Christensen, A. Møller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proc. PLAN-X-02*, 2002.

[6] A.S. Christensen, A. Møller, and M.I. Schwartzbach. Extending Java for high-level web service construction. *ACM TOPLAS*, 25, 2003.

[7] K.-G. Doh, H. Kim, and D.A. Schmidt. Abstract parsing: static analysis of dynamically generated string output using LR-parsing technology. In *Proc. 16th International Static Analysis Symposium*, pages 256-272. Springer LNCS 5673, 2009.

[8] K.-G. Doh, H. Kim, and D.A. Schmidt. Abstract LR-parsing. In *Proc. Formal Modeling: Actors; Open Systems, Biological Systems*, pages 90-109. Springer LNCS 7000, 2011.

[9] E. Duesterwald, R. Gupta, and M.L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM TOPLAS*, 19:992-1030, 1997.

[10] J. Hopcroft and J. Ullman. Formal Languages and their Relation to Automata. Addison Wesley, 1969

[11] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1995.

[12] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium*, pages 336-352. Springer LNCS 4134, 2006.

[13] Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. 14th International Conference on World Wide Web*, pages 432-441, 2005.

[14] Y. Minamide and A. Tozawa. XML validation for context-free grammars. In *Proc. Asian Symposium on Programming Languages and Systems*, pages 357-373. Springer LNCS 4279, 2006.

[15] A. Møller and M. Schwarz. HTML validation of context-free languages. 14th International Conference on Foundations of Software Science and Computation Structures, pages 426-440. Springer LNCS 6604, 2011.

[16] T. Nishiyama and Y. Minamide. A translation from the HTML DTD into a regular hedge grammar. In *Proc. 13th International Conference on Implementation and Application of Automata*, pages 122-131. Springer LNCS 5148, 2008.

[17] P. Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 59-70, 2005.

[18] G. Wassermann and Z. Su. The essence of command injection attacks in web applications. In *Proc. 33rd ACM Symp. POPL*, pages 372-382, 2006.

[19] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM PLDI*, pages 32-41, 2007.