HTML Validation of Context-Free Languages

Anders Møller* and Mathias Schwarz*

Aarhus University, Denmark
{amoeller,schwarz}@cs.au.dk

Abstract. We present an algorithm that generalizes HTML validation of individual documents to work on context-free sets of documents. Together with a program analysis that soundly approximates the output of Java Servlets and JSP web applications as context-free languages, we obtain a method for statically checking that such web applications never produce invalid HTML at runtime. Experiments with our prototype implementation demonstrate that the approach is useful: On 6 open source web applications consisting of a total of 104 pages, our tool finds 64 errors in less than a second per page, with 0 false positives. It produces detailed error messages that help the programmer locate the sources of the errors. After manually correcting the errors reported by the tool, the soundness of the analysis ensures that no more validity errors exist in the applications.

1 Introduction

An HTML document is *valid* if it syntactically conforms to a DTD for one of the versions of HTML. Since the HTML specifications only prescribe the meaning of valid documents, invalid HTML documents are often rendered differently, depending on which browser is used [1]. For this reason, careful HTML document authors validate their documents, for example using the validation tool provided by W3C[1]. An increasing number of HTML documents are, however, produced dynamically by programs running on web servers. It is well known that errors caught early in development are cheaper to fix. Our goal is to develop a program analysis that can check statically, that is, at the time programs are written, that they will never produce invalid HTML when running. We want this analysis to be *sound*, in the sense that whenever it claims that the given program has this property that is in fact the case, *precise* meaning that it does not overwhelm the user with spurious warnings about potential invalidity problems, and *efficient* such that it can analyze non-trivial applications with modest time and space resources. Furthermore, all warning messages being produced must be *useful* toward guiding the programmer to the source of the potential errors.

The task can be divided into two challenges: 1) Web applications typically generate HTML either by printing page fragments as strings to an output stream

* Supported by The Danish Research Council for Technology and Production, grant no. 274-07-0488.

[1] http://validator.w3.org

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 426–440, 2011.

(as in e.g. Java Servlets) or with template systems (as e.g. JSP, PHP, or ASP). In any case, the analysis front-end must extract a formal description of the set of possible outputs of the application, for example in the form of a context-free grammar. 2) The analysis back-end must analyze this formal description of the output to check that all strings that it represents are valid HTML. Several existing techniques follow this pattern, although considering XHTML instead of HTML [8, 6]. In practice, however, many web applications output HTML data, not XHTML data, and the existing techniques – with the exception of the work by Nishiyama and Minamide [11], which we discuss in Section 2 – do not work for HTML.

The key differences between HTML and XHTML are that the former allows certain tags to be omitted, for example the start tags `<html>` and `<tbody>` and the end tags `</html>` and `</p>`, and that it uses tag inclusions and exclusions, for example to forbid deep nesting of `a` elements. This extra flexibility of HTML is precisely what makes it popular, compared to its XML variant XHTML. On the other hand, this flexibility means that the process of checking *well-formedness*, i.e. that a document defines a proper tree structure, cannot be separated from the process of checking *validity*, i.e. that the tree structure satisfies the requirements of the DTD.

In this paper, we present an algorithm that, given as input a context-free grammar G and an SGML DTD D (one of the DTDs that exist for the different versions of HTML[2]), checks whether every string in the language of G is valid according to D, written $\mathcal{L}(G) \subseteq \mathcal{L}(D)$. The key idea in our approach is a generalization of a core algorithm for SGML parsing [4, 14] to work on context-free sets of documents rather than individual documents.

1.1 Outline of the Paper

The paper is organized as follows. We first give an overview of related approaches in Section 2. In Section 3 we then present a formal model of SGML/HTML validation that captures the essence of the features that distinguish it from XML/XHTML validation. Based on this model, in Section 4 we present our generalization for validating context-free sets of documents. We have implemented the algorithm together with an analysis front-end for Java Servlets and JSP, which constitute a widely used platform for server-based web application development. (Due to the limited space we focus on the back-end in this paper.) In Section 5, we report on experiments on a range of open source web applications. Our results show that the algorithm is fast and able to pinpoint programming errors. After manually correcting the errors based on the messages generated by the tool, the analysis is able to prove that the output will always be valid HTML when the applications are executed.

[2] The HTML 5 language currently under development will likely evoke renewed interest in HTML. Although it technically does not use SGML, its syntax closely resembles that of the earlier versions.

```
<%@ page import="java.util.*, org.example" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html><head><meta name="description" content="Joke Collection">
  <title>Jokes</title>
  <%! List<Joke> js = Jokes.get();%>
  <body><table>
    <tr><th>Question<th>Punch line</tr>
      <% if (js.size() > 0) {
        request.setParameter("Jokes", js); %>
        <c:forEach items="${Jokes}" var="joke">
          <tr><td><c:out value="${joke.question}"/>
            <td><c:out value="${joke.punchline}"/></tr>
        </c:forEach>
      <% } else {
        out.print("<td>No more jokes</tr>");
      } %>
  </table></body>
</html>
```

Fig. 1. A JSP page that uses the JSTL tag library and embedded Java code. The example takes advantage of SGML features such as tag omission and inclusions.

1.2 Example

Figure 1 shows an example of a JSP program that outputs a dynamically generated table from a list of data using a combination of many of the JSP and SGML features that appear in typical applications. The `meta` element is not part of the content model of `head`, but it is allowed by an SGML inclusion rule. The `body` element contains a table where both the start and the end tag of the `tbody` element are omitted, and a parser needs to insert those to validate a generated document. Similarly, all `td` and `th` end tags are omitted. The contents of the table are generated by a combination of tags from JSP Standard Tag Library, embedded Java code that prints to the output stream, and ordinary JSP template code.

The static analysis that we present is able to soundly check that the output from such code is always valid according to e.g. the HTML 4.01 Transitional specification.

2 Related Work

Previous work on reasoning about programs that dynamically generate semistructured data has focused on XML [9], not SGML, despite the fact that the SGML language HTML remains widely used. (Since XML languages are essentially the subclass of SGML languages that do not use the tag omission and exception features, our algorithm also works for XML.) Most closely related to our approach is the work by Minamide et al. [7, 8, 11] and Kirkegaard and Møller [6].

In [7] context-free grammars are derived from PHP programs. From such a grammar, sample documents are derived and processed by an ordinary HTML or XHTML validator. Unless the nesting depth of the elements in the generated documents is bounded, this approach is unsound as it may miss errors. Later, an alternative grammar analysis was suggested for soundly validating dynamically generated XML data [8]. That algorithm relies on the theory of balanced grammars over an alphabet of tag names, which does not easily generalize to handle the tag omission and inclusion/exclusion features that exist in HTML. The approach in [6] is comparable to [8], however considering the more fine-grained alphabet of individual Unicode characters instead of entire tag names and using XML graphs for representing sets of XML documents.

Yet another grammar analysis algorithm is presented by Nishiyama and Minamide [11]. They define a subclass of SGML DTDs that includes HTML and shows a translation into regular hedge grammars, such that the validation problem reduces to checking inclusion of a context-free language in a regular language. That approach has some limitations, however: 1) it does not support start tag omission, although that feature of SGML is used in HTML (e.g. `tbody` and `head`); 2) the exclusion feature is handled by a transformation of the DTD that may lead to an exponential blow-up prohibiting practical use; and 3) the inclusion feature is not supported. The alternative approach we suggest overcomes all these limitations.

The abstract parsing algorithm by Doh et al. [3] and the grammar-based analysis by Thiemann [12] are also based on the idea of generalizing existing parsing algorithms. The approach in [3] relies on abstract interpretation with a domain of LR(k) parse stacks constructed from an LR(k) grammar for XHTML, and [12] is based on Earley's parsing algorithm. By instead using SGML parsing as a starting point, we avoid the abstraction and we handle the special features of HTML: Given a context-free grammar describing the output of a program, our algorithm for checking that all derivable strings are valid HTML is both sound and complete.

3 Parsing HTML Documents

Although HTML is based on the SGML standard [4] it uses only a small subset of the features of the full standard. SGML languages are formally described using the DTD language (not to confuse with the DTD language for XML). Such a description provides a formal description for the parser on how a document is parsed from its textual form into a tree structure. Specifically, in SGML both start and end tags may be omitted if 1) allowed by the DTD, and 2) the omission does not result in ambiguities in the parsing of the document. The DTD description provides the content models, that is, the allowed children of each element, as deterministic regular expressions over sequences of elements. Furthermore special exceptions, called inclusions and exclusions, are possible for allowing additional element children or disallowing nesting of certain elements. An inclusion rule permits elements anywhere in the descendant tree even if not allowed by the content model expressions. Conversely, an exclusion rule prohibits elements, overriding the content model expressions and inclusions.

Consider a small example DTD:

```
<!ELEMENT inventory - - (item*) +(note)>
<!ELEMENT item - O (#PCDATA)>
<!ELEMENT note - O (#PCDATA)>
```

In each element declaration, `O` means “optional” and `-` means “required”, for the start tag and the end tag, respectively. This DTD declares an element `inventory` where the start and end tags are both required. (Following the usual SGML terminology, an *element* generally consists of a start tag and its matching end tag, although certain tags may be omitted in the textual representation of the documents.) The content model of `inventory` allows a sequence of `item` elements as children in the document tree. In addition, `note` is included such that `note` elements may be descendants of `inventory` elements even though they are not allowed directly in the content models of the descendants. The second line declares an element `item` that requires a start tag but allows omission of the end tag. The content model of `item` allows only text (PCDATA) and no child elements in the document tree. Finally, the element `note` is also declared with end tag omission and PCDATA content. An example of a valid document for this DTD is the following:

```
<inventory><item>gadget<item>widget</inventory>
```

The parser inserts the omitted end tags for `item` to obtain the following document, which is valid according to the DTD content models for `inventory` and `item`:

```
<inventory><item>gadget</item><item>widget</item></inventory>
```

Because of the inclusion of `note` elements in the declaration of `inventory`, the following document is also parsed as a valid instance:

```
<inventory><item>gadget<note>new</note><item>widget</inventory>
```

SGML is similar to XML but it has looser requirements on the syntax of the input documents. For the features used by HTML, the only relevant differences are that XML does not support tag omissions nor content model exceptions.

We consider only DTDs that are acyclic:

Definition 1. *An SGML DTD is* acyclic *if it satisfies the following requirement: For elements that allow end tag omissions there must be a bound on the possible depth of the direct nesting of those elements. That is, if we create a directed graph where the nodes correspond to the declared elements whose end tags may be omitted and there is an edge from a node A to a node B if the content model of A contains B, then there must be no cycles in this graph.*

This requirement also exists in Nishiyama and Minamide’s approach [11], and it is fulfilled by all versions of the HTML DTD. Contrary to their approach we do not impose any further restrictions and our algorithm thus works for all the HTML DTDs without any limitations or rewritings.

3.1 A Model of HTML Parsing

As our algorithm is a generalization of the traditional SGML parsing algorithm we first present a formal description of the essence of that algorithm. We base our description on the work by Warmer and van Egmond [14]. The algorithm provides the basis for explaining our main contribution in the next section.

We abstract away from SGML features such as text (i.e. PCDATA), comments, and attributes. These features are straightforward to add subsequently. Furthermore, a lexing phase allows us to consider strings over the alphabet of start and end tags, written <a> and </a>, respectively, for every element a declared in the DTD. (This lexing phase is far from trivial; our implementation is based on the technique used in [6], and we omit the details here due to the limited space.) More formally, we consider strings over the alphabet $\Sigma = \{\texttt{<}a\texttt{>} \mid a \in \mathcal{E}\} \cup \{\texttt{</}a\texttt{>} \mid a \in \mathcal{E}\}$ where $\mathcal{E}$ is the set of declared element names in the DTD. We assume that root $\in \mathcal{E}$ is a pseudo-element representing the root node of the document, with a content model that accepts a single element of any kind (or, one specific, such as `html` for HTML). The sets of included and excluded elements of an element $a \in \mathcal{E}$ are denoted I_a and E_a, respectively.

For simplicity, we represent all content models together as one finite-state automaton [5] defined as follows:

Definition 2. *A* content model automaton *for a DTD D is a tuple $(Q, \mathcal{E}, [q_a]_{a\in\mathcal{E}}, F, \delta)$ where Q is a set of states, its alphabet is $\mathcal{E}$ as defined above, $[q_a]_{a\in\mathcal{E}}$ is a family of initial states (one for each declared element), $F \subseteq Q$ is a set of accept states and $\delta : Q \times \Sigma \hookrightarrow Q$ is a partial transition function (with $\perp$ representing undefined).*

Following the requirement from the SGML standard that content models must be unambiguous, this content model automaton can be assumed to be deterministic by construction. Also, we assume that all states in the automaton can reach some accept state. Each state in the automaton uniquely corresponds to a position in a content model expression in D.

SGML documents are parsed in a single left-to-right scan with a look-ahead of 1. The state of the parser is represented by a *context stack*. The set of possible contexts is $\mathcal{H} = \mathcal{E} \times Q \times \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{E})$. ($\mathcal{P}(\mathcal{E})$ denotes the powerset of $\mathcal{E}$.) We refer to the context $c_n = (a, q, \iota, \eta)$ at the top of a stack $c_1 \cdots c_n \in \mathcal{H}^*$ as the *current context*, and a, q, ι, and η are then the *current element*, the *current state*, the *current inclusions*, and the *current exclusions*, respectively. An element b is *permitted* in the current context (a, q, ι, η) if $\delta(q, b) \neq \perp$. We refer to a tag a just below another tag b in the context stack as b's *parent*. We say that $\textsc{OmitStart}(a, q)$ holds if the start tag of a elements may be omitted according to D when the current state is q, and, similarly, $\textsc{OmitEnd}(a, q)$ holds if the end tag of a elements may be omitted in state q. (The precise rules defining $\textsc{OmitStart}$ and $\textsc{OmitEnd}$ from D are quite complicated; we refer to [4,14] for the details.) The current inclusions and exclusions reflect the sets of included and excluded elements, respectively. These two sets can in principle be determined

```
function $Parse_D(p \in \mathcal{H}^*,\ x \in \Sigma^*)$ :
if $|x| = 0$ then
    // reached end of input
    return $p$
else if $|p| = 0$ then
    // empty stack error
    return $\bigcirc$
let $p_1 \cdots p_{n-1} \cdot (a_n, s_n, \iota_n, \eta_n) = p$
let $x_1 \cdots x_m = x$
if $x_1 =$ <$a$> $\wedge\ a \notin \eta_n$ for some $a \in \mathcal{E}$ then
    // reading a non-excluded start tag
    if $\delta(s_n, a) \neq \bot$ then
        // the start tag is permitted by the content model, push onto stack and proceed
        return $Parse_D\big(p_1 \cdots p_{n-1} \cdot (a_n, \delta(s_n, a), \iota_n, \eta_n) \cdot (a, q_a, \iota_n \cup I_a, \eta_n \cup E_a),\ x_2 \cdots x_m\big)$
    else if $a \in \iota_n$ then
        // the start tag is permitted by inclusion, push onto stack and proceed
        return $Parse_D\big(p_1 \cdots p_n \cdot (a, q_a, \emptyset, \eta_n \cup E_a),\ x_2 \cdots x_m\big)$
else if $x_1 =$ </$a$> $\wedge\ a = a_n \wedge s_n \in F$ for some $a \in \mathcal{E}$ then
    // reading an end tag that is permitted, pop from stack and proceed
    return $Parse_D\big(p_1 \cdots p_{n-1},\ x_2 \cdots x_m\big)$
else if $\textsc{OmitEnd}(a_n, s_n)$ then
    // insert omitted end tag, then retry
    return $Parse_D(p,$ </$a_n$> $\cdot\, x)$
else if $\exists a' \in \mathcal{E} : \textsc{OmitStart}(a', s_n)$ then
    // insert omitted start tag, then retry
    return $Parse_D(p,$ <$a'$> $\cdot\, x)$
else
    // parse error
    return $\lightning$
```

Fig. 2. The $Parse_D$ function for checking validity of a given document

from the element names appearing in the context stack, but we maintain them in each context for reasons that will become clear in Section 4.

Informally, when encountering a start tag <a> that is permitted in the current context, its content automaton state is modified accordingly, and a new context is pushed onto the stack. When an end tag </a> is encountered, the current context is popped off the stack if it matches the element name a.

An end tag may be omitted only if it is followed by either the end tag of another open element or a start tag that is not allowed at this place. A start tag may be omitted only if omission does not cause an ambiguity during parsing. These conditions, which define OmitEnd and OmitStart, can be determined from the current state and either the next tag in the input or the current element on the stack, respectively, without considering the rest of the parse stack and input. Moreover, OmitStart has the property that no more than $|\mathcal{E}|$ omitted start tags can be inserted before the next tag from the input is consumed.

Our formalization of SGML parsing is expressed as the function $Parse_D$: $\mathcal{H}^* \times \Sigma^* \to \big(\mathcal{H}^* \cup \{\lightning, \bigcirc\}\big)$ shown in Figure 2. The result $\bigcirc$ arises if an end tag is encountered while the stack is empty, and $\lightning$ represents other kinds of parse errors. In this algorithm, OmitEnd and OmitStart allow us to abstract away from the precise rules for tag omission, to keep the presentation simple. The algorithm captures an essential property of SGML parsing: a substring $x \in \Sigma^*$ of a document is parsed relative to a parse stack $p \in \mathcal{H}^*$ as defined above, and it outputs a new parse stack or one of the error indicators ○ and ↯. We distinguish between the two kinds of errors for reasons that become clear in Section 4.

With this, we can define validity of a document relative to the DTD D:

Definition 3. *A string $x \in \Sigma^*$ is a* valid document *if*

$$\mathit{Parse}_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), x\big) = (\mathsf{root}, q, \emptyset, \emptyset)$$

for some $q \in F$.

The Parse_D function has some interesting properties that we shall need in Section 4:

Observation 4. *Notice that the Parse_D function either returns directly or via a tail call to itself. Let $(p^1, x^1), (p^2, x^2), \ldots$ be the sequence of parameters to Parse_D that appear if executing $\mathit{Parse}_D(p^1, x^1)$ for some $p^1 \in \mathcal{H}^*, x^1 \in \Sigma^*$. Now, because the DTD is acyclic, for all $i = 1, 2, \ldots$ we have $|x^{i+|\mathcal{E}|}| < |x^i|$, that is, after at most $|\mathcal{E}|$ recursive calls, one more input symbol is consumed. Moreover, in each step in the recursion sequence, the decisions made depend only on the current context and the next input symbol.*

4 Parsing Context-Free Sets of Documents

We now show that the parsing algorithm described in the previous section can be generalized to work for *sets* of documents, or more precisely, context-free languages over the alphabet Σ. The resulting algorithm determines whether or not all strings in a given language are valid according to a given DTD. The languages are represented as context-free grammars that are constructed by the analysis front-end from the programs being analyzed.

The definitions of context-free grammars and their languages are standard:

Definition 5. *A* context-free grammar *(CFG) is a tuple $G = (N, \Sigma, P, S)$ where N is the set of nonterminal symbols, Σ is the alphabet (of start and end tag symbols, as in Section 3.1), P is the set of productions of the form $A \rightarrow r$ where $A \in N$, $r \in (\Sigma \cup N)^*$, and S is the start nonterminal. The* language *of G is $\mathcal{L}(G) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$ where $\Rightarrow^*$ is the reflexive transitive closure of the derivation relation $\Rightarrow$ defined by $u_1 A u_2 \Rightarrow u_1 r u_2$ whenever $u_1, u_2 \in (\Sigma \cup N)^*$ and $A \rightarrow r \in P$.*

Definition 6. *A CFG G is* valid *if x is valid for every $x \in \mathcal{L}(G)$.*

To simplify the presentation we will assume that G is in Chomsky normal form, so that all productions are of the form $A \rightarrow s$ or $A \rightarrow A'A''$ where $s \in \Sigma$ and $A, A', A'' \in N$, and that there are no useless nonterminals. It is well-known how to transform an arbitrary CFG to this form [5]. We can disregard the empty string since that is never valid for any DTD, and the empty language is trivially valid.

The idea behind the generalization of the parse algorithm is to find out for every occurrence of an alphabet symbol s in the given CFG which context stacks may appear when encountering s during parsing of a string. The context stacks may of course be unbounded in general. However, because of Observation 4 *we only need to keep track of a bounded size top (i.e. a postfix) of each context stack*, and hence a bounded number of context stacks, at every point in the grammar.

4.1 Generating Constraints

To make the idea more concrete, we define a family of *context functions*, one for each nonterminal $A \in N$. Each is a partial function that takes as input a context stack and returns a set of context stacks:

$$\mathcal{C}_A : \mathcal{H}^* \hookrightarrow \mathcal{P}(\mathcal{H}^*)$$

Informally, the domain of $\mathcal{C}_A$ consists of the context stacks that appear during parsing when entering a substring derived from A, and the co-domain similarly consists of the context stacks that appear immediately after the substring has been parsed. Formally, assume $x \in \mathcal{L}(G)$ such that $S \Rightarrow^* u_1 A u_2 \Rightarrow^* u_1 y u_2 = x$, that is, the nonterminal A is used in the derivation of x, and y is the substring derived from A. The domain $dom(\mathcal{C}_A)$ then contains the context stack p that arises after parsing of u_1, that is, $p = Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), u_1\big) \in dom(\mathcal{C}_A)$. Similarly, $\mathcal{C}_A(p)$ contains the context stack that arises after parsing of $u_1 y$, that is, $Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), u_1 y\big) = Parse_D(p, y) \in \mathcal{C}_A(p)$ if $p \notin \{\lightning, \bigcirc\}$. As explained in detail below, we truncate the context stacks and only store the top of the stacks in these sets. To obtain an efficient algorithm, we truncate as much as possible and exploit the fact that $Parse_D$ returns $\bigcirc$ if a too short context stack is given.

The context functions are defined from the DTD as a solution to the set of constraints defined by the following three rules:

§1 Following Definition 3, parsing starts with the initial context stack at the start nonterminal S and must end in a valid final stack:

$$\mathcal{C}_S(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \subseteq \{(\mathsf{root}, q, \emptyset, \emptyset) \mid q \in F\}$$

§2 For every production of the form $A \to s$ in P where $s \in \Sigma$, the context function for A respects the $Parse_D$ function, which must not return $\lightning$ or $\bigcirc$:

$$\forall p \in dom(\mathcal{C}_A) : p' \notin \{\lightning, \bigcirc\} \ \wedge \ p' \in \mathcal{C}_A(p) \text{ where } p' = Parse_D(p, s)$$

§3 For every production of the form $A \to A'A''$ in P, the entry context stacks of A are also entry context stacks for A', the exit context stacks for A' are also entry context stacks for A'', and the exit context stacks for A'' are also exit context stacks for A. However, we allow the context stacks to be truncated when propagated from one nonterminal to the next:

$$\begin{aligned}\forall p \in dom(\mathcal{C}_A) : \exists p_1, p_2 : p = p_1 \cdot p_2 \ \wedge \ p_2 \in dom(\mathcal{C}_{A'}) \ \wedge \\ \forall p_2' \in \mathcal{C}_{A'}(p_2) : \exists t_1, t_2 : p_1 \cdot p_2' = t_1 \cdot t_2 \ \wedge \ t_2 \in dom(\mathcal{C}_{A''}) \ \wedge \\ \forall t_2' \in \mathcal{C}_{A''}(t_2) \Rightarrow t_1 \cdot t_2' \in \mathcal{C}_A(p)\end{aligned}$$

Note that rule §3 permits the context stacks to be truncated; on the other hand, rule §2 ensures that the stacks are not truncated too much since that would lead to the error value $\bigcirc$.

Theorem 7. *There exists a solution to the constraints defined by the rules above for a grammar G if and only if G is valid.*

Proof. See the technical report [10].

4.2 Solving Constraints

It is relatively simple to construct an algorithm that searches for a solution to the collection of constraints generated from a CFG by the rules defined in Section 4.1. Figure 3 shows the pseudo-code for such an algorithm, $\mathit{ParseCFG}_D$. We write $w\,\textsc{defs}\,A$ for $w \in P$, $A \in N$ if A appears on the left-hand side of w, and $w\,\textsc{uses}\,A$ if A appears on the right-hand side of w. The solution being constructed is represented by the family of context functions, denoted $[\mathcal{C}_A]_{A \in N}$.

The idea in the algorithm is to search for a solution by truncating the context stacks as much as possible, iteratively trying longer context stacks, until the special error value $\bigcirc$ no longer appears. The algorithm initializes $[\mathcal{C}_A]_{A \in N}$ on line 6 and iteratively on lines 9–58 extends these functions to build a solution. The worklist W (a queue, without duplicates) consists of productions that need to be processed because the domains of the context functions of their left-hand-side nonterminals have changed. The function Δ maintains for each nonterminal a set of context stacks that are known to lead to $\bigcirc$.

Each production in the worklist of the form $A \to s$ is parsed according to rule §2 on lines 14– 26, relative to each context stack p in $\mathit{dom}(\mathcal{C}_A)$. If this results in $\bigcirc$, the corresponding context stack is added to $\Delta(A)$, and all productions that use A are added to the worklist to make sure that the information that the context stack was too short is propagated back to those productions. If a parse error $\lightning$ occurs (line 20), the algorithm terminates with a failure. If the parsing is successful (line 23), the resulting context stack p' is added to $\mathcal{C}_A$.

For a production of two nonterminals, $A \to A'A''$, we proceed according to rule §3. For each context stack p in $\mathit{dom}(\mathcal{C}_A)$ on line 29 we pick the smallest possible postfix p_2 of p that is not in $\Delta(A')$ and propagate this to $\mathcal{C}_{A'}$. If no such postfix exists, we know that p is too short, so we update $\Delta(A)$ and W as before. Otherwise, we repeat the process (line 37) to propagate the resulting context stack through A'' and further to $\mathcal{C}_A$ (line 46).

Finally, on line 57 we check that rule §1 is satisfied.

Theorem 8. *The $\mathit{ParseCFG}_D$ algorithm always terminates, and it terminates successfully if and only if a solution exists to the constraints from Section 4.1 for the given CFG.*
(We leave a proof of this theorem as future work.)

Corollary 9. *Combining Theorem 7 and Theorem 8, we see that $\mathit{ParseCFG}_D$ always terminates, and it terminates successfully if and only if the given CFG is valid.*

4.3 Example

As an example of a normalized grammar, consider $G_{ul} = (N, \Sigma, P, S)$ where $N = \{A_1, A_2, A_3, A_4, A_5, A_6\}$, $\Sigma = \{$`<ul>`, `</ul>`, `<li>`, `</li>`$\}$, $S = A_1$, and P consists of the following productions:

```
function $ParseCFG_D(N, \Sigma, P, S)$ :
  declare $W \subseteq P,\ [\mathcal{C}_A]_{A \in N} : \mathcal{H}^* \hookrightarrow \mathcal{P}(\mathcal{H}^*),\ \Delta : N \to \mathcal{P}(\mathcal{H}^*)$
  // initialize worklist and context functions
  $W := [\, w \in P \mid w \text{ DEFS } S \,]$
  for all $A \in N,\ p \in \mathcal{H}^*$ do
    $\mathcal{C}_A(p) := \begin{cases} \emptyset & \text{if } A = S \ \wedge\ p = (\mathsf{root}, q_{\mathsf{root},\emptyset,\emptyset}) \\ \bot & \text{otherwise} \end{cases}$
    $\Delta(A) := \emptyset$
  // iterate until fixpoint
  while $W \neq \emptyset$ do
    remove the next production $A \to r$ from $W$
    for all $p \in dom(\mathcal{C}_A)$ do
      if $A \to r$ is of the form $A \to s$ where $s \in \Sigma$ then
        // rule §2
        let $p' = Parse_D(p, s)$
        if $p' = \bigcirc$ then
          // record that entry context stack p is too short for A
          $\Delta(A) := \Delta(A) \cup \{p\}$
          $\mathcal{C}_A(p) := \bot$
          for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
        else if $p' = \lightning$ then
          // fail right away
          fail
        else if $p' \notin \mathcal{C}_A(p)$ then
          // add new final context stack p' for A
          $\mathcal{C}_A(p) := \mathcal{C}_A(p) \cup \{p'\}$
          for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
      else if $A \to r$ is of the form $A \to A'A''$ where $A', A'' \in N$ then
        // rule §3
        let $p_2$ be the smallest string such that $p = p_1 \cdot p_2$ and $p_2 \notin \Delta(A')$
        if no such $p_2$ exists then
          // record that entry context stack p is too short for A
          $\Delta(A) := \Delta(A) \cup \{p\}$
          $\mathcal{C}_A(p) := \bot$
          for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
        else if $p_2 \in dom(\mathcal{C}_{A'})$ then
          for all $p'_2 \in \mathcal{C}_{A'}(p_2)$ do
            let $t_2$ be the smallest string such that $p_1 \cdot p'_2 = t_1 \cdot t_2$ and $t_2 \notin \Delta(A'')$
            if no such $t_2$ exists then
              // record that entry context stack p is too short for A
              $\Delta(A) := \Delta(A) \cup \{p\}$
              $\mathcal{C}_A(p) := \bot$
              for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
            else if $t_2 \in dom(\mathcal{C}_{A''})$ then
              if $\{t_1 \cdot t'_2 \mid t'_2 \in \mathcal{C}_{A''}(t_2)\} \not\subseteq \mathcal{C}_A(p)$ then
                // add new final context stacks for A
                $\mathcal{C}_A(p) := \mathcal{C}_A(p) \cup \{t_1 \cdot t'_2 \mid t'_2 \in \mathcal{C}_{A''}(t_2)\}$
                for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
            else
              // add new entry context stack t_2 for A''
              $\mathcal{C}_{A''}(t_2) := \emptyset$
              for all $w \in P$ where $w \text{ DEFS } A''$ add $w$ to $W$
        else
          // add new entry context stack p_2 for A'
          $\mathcal{C}_{A'}(p_2) := \emptyset$
          for all $w \in P$ where $w \text{ DEFS } A'$ add $w$ to $W$
    // rule §1
    if $\mathcal{C}_S(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \not\subseteq \{(\mathsf{root}, q, \emptyset, \emptyset) \mid q \in F\}$ then
      fail
  return $[\mathcal{C}_A]_{A \in N}$
```

Fig. 3. The $ParseCFG_D$ algorithm for solving the parse constraints for a given CFG

$$\begin{array}{ll} A_1 \rightarrow A_5\ A_2 & A_2 \rightarrow A_6\ A_3 \\ A_3 \rightarrow \texttt{</ul>} & A_4 \rightarrow \texttt{<li>} \\ A_5 \rightarrow \texttt{<ul>} & A_6 \rightarrow \texttt{<li>} \\ A_6 \rightarrow A_4\ A_1 & \end{array}$$

The language generated by G_{ul} consists of documents that have a `ul` root element containing a single `li` element that in turn contains zero or one `ul` element. The grammar can thus generate deeply nested `ul` and `li` elements, and truncation of context stacks is therefore crucial for the $\mathit{ParseCFG}_D$ algorithm to terminate. Notice that all `</li>` end tags are omitted in the documents.

We wish to ensure that the strings generated from G_{ul} are valid relative to the following DTD, which mimics a very small fraction of the HTML DTD for unordered lists:

```
<!ELEMENT ul - - (li*)>
<!ELEMENT li - O (ul*)>
```

For this combination of a CFG and a DTD, the $\mathit{ParseCFG}_D$ algorithm produces the following solution to the constraints:

	C
A_1	$(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \mapsto \{(\mathsf{root}, q, \emptyset, \emptyset)\}$ $(\texttt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{(\texttt{li}, q_{li}, \emptyset, \emptyset)\}$
A_2	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{\epsilon\}$
A_3	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\texttt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{\epsilon\}$
A_4	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\texttt{li}, q_{li}, \emptyset, \emptyset)\}$
A_5	$(\texttt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{(\texttt{li}, q_{li}, \emptyset, \emptyset) \cdot (\texttt{ul}, q_{ul}, \emptyset, \emptyset)\}$ $(\texttt{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \mapsto \{(\texttt{root}, q, \emptyset, \emptyset) \cdot (\texttt{ul}, q_{ul}, \emptyset, \emptyset)\}$
A_6	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\texttt{li}, q_{li}, \emptyset, \emptyset)\}$

Although the context stacks may grow arbitrarily when parsing individual documents with Parse_D, the truncation trick ensures that $\mathit{ParseCFG}_D$ terminates and succeeds in capturing the relevant top-most parts of the context stacks.

5 Experimental Results

We have implemented the algorithm from Section 4.2 in Java, together with an analysis front-end for constructing CFGs that soundly approximate the output of web applications written with Java Servlets and JSP. The front-end follows the structure described in [6], extended with specialized support for JSP, and builds on Soot [13] and the Java String Analyzer [2]. (We omit a more detailed explanation of this front-end, due to the limited space.)

The purpose of the prototype implementation is to obtain preliminary answers to the following research questions:

- What is the typical analysis time for a Servlet/JSP page, and how is the analysis time affected by the absence or presence of validity errors?
- What is the precision of the analysis in terms of false positives?
- Are the warnings produced by the tool useful to locate the sources of the errors?

We have run the analysis on six open source programs found on the web. The programs range from simple one man projects, such as the JSP Chat application (JSP Chat[3]), the official J2EE tutorial Servlet and JSP examples (J2EE Bookstore 1 and 2[4]) to the widely used blogging framework Pebble[5], which included dozens of pages and features. We have also included the largest example from a book on JSTL (JSTL Book ex.[6]) and an application named JPivot[7]. The tests have been performed on a 2.4 GHz Core i5 laptop with 4GB RAM running OS X. As DTD, we use HTML 4.01 Transitional.

Figure 4 summarizes the results. For each program, it shows the number of JSP pages, the time it takes to run the whole analysis on all pages (excluding the time used by Soot), the time spent in the CFG parser algorithm, the number of warnings from the analyzer, and the number of false positives determined by manual inspection of the analyzed source code.

The tool currently has two limitations, which we expect to remedy with a modest additional implementation effort. First, validation of attributes is currently not supported. Second, the implementation can track a validity error to the place in the generated Java code where the invalid element is generated, but not all the way back to the JSP source in the case of JSP pages.

In some cases when an unknown value is inserted into the output without escaping special XML characters (for example, by using the `out` tag from JSTL), the front-end is unable to reason about the language of that value. This may for instance happen when the value is read from disk or input at runtime. The analysis will in such cases issue an additional warning, which is not included in the count in Figure 4, and treat the unknown value as a special alphabet symbol and continue analyzing the grammar. In practice, there are typically a few such symbols per page. While they may be indications of cross site scripting vulnerabilities, there may also be invariants in the program ensuring that there is no problem at runtime.

The typical analysis time for a single JSP page is around 200-600 ms. As can be seen from the table, only a small fraction of the time is spent on parsing the CFG. The worklist algorithm typically requires between 1 and 100 iterations for each JSP page, which means that each nonterminal is visited between 1 and 10 times.

Validity errors were found in all the applications. The following is an example of a warning generated by the tool on the JSP Chat application:

[3] `http://www.web-tech-india.com/software/jsp_chat.php`

[4] `http://download.oracle.com/javaee/5/tutorial/doc/bnaey.html`

[5] `http://pebble.sourceforge.net/`

[6] `http://www.manning.com/bayern/`

[7] `http://jpivot.sourceforge.net/`

Program	Pages	Time	CFG Parser time	Warnings	False positives
Pebble[3]	61	24.0 s	369 ms	32	0
J2EE Bookstore 1[4]	5	6.7 s	93 ms	5	0
J2EE Bookstore 2[4]	7	9.0 s	<1 ms	7	0
JPivot[5]	3	2.8 s	8 ms	2	0
JSP Chat[6]	14	6.8 s	100 ms	12	0
JSTL Book ex.[7]	14	4.9 s	24 ms	6	0

Fig. 4. Analysis times and results for various open source web applications written in Java Servlets and JSP

```
ERROR: Invalid string printed in
  dk.brics.servletvalidator.jsp.generated.editInfo_jsp on line 94:
Start tag INPUT not allowed in TBODY
Parse context is [root HTML BODY DIV CENTER FORM TABLE TBODY]
```

This warning indicates that the programmer forgot both a `tr` start tag and a `td` start tag in which the `input` element would be allowed, causing the `input` tag to appear directly inside the `tbody` element. This may very well lead to browsers rendering the page differently.

The reason that all JSP pages of the J2EE Bookstore applications are invalid it that there is an unmatched `</center>` tag and a nonstandard `<comment>` tag in a header used by all pages. After removing these two tags, only one page of this application is (correctly) rejected by the analysis. While Pebble seems to be programmed with the goal of only outputting valid HTML, the general problem in this web application is that the `table`, `ul`, and `tr` elements require non-empty contents, which is not always respected by Pebble. Furthermore, several more serious errors, such as forgotten `td` tags, exist in the application. The JSP Chat application is written in JSP but makes heavy use of embedded Java code. The tool is able to analyze it precisely enough to find several errors that are mostly due to unobvious (but feasible) flow in the program.

Based on the warnings generated by the tool, we managed to manually correct all the errors within a few hours without any prior knowledge of the applications. After running the analysis again, no more warnings were produced. This second round of analysis took essentially the same time as before the errors were corrected. Since the analysis is sound, we can trust that the applications after the corrections cannot output invalid HTML.

6 Conclusion

We have presented an algorithm for validating context-free sets of documents relative to an HTML DTD. The key idea – to generalize a parsing algorithm for SGML to work on grammars instead of concrete documents – has lead to an approach that smoothly handles the intricate features of HTML, in particular tag omissions and exceptions. Preliminary experiments with our prototype implementation indicate that the approach is sufficiently efficient and precise to function as a practically useful tool during development of web applications. In future work, we plan to improve the tool to accommodate for attributes and to trace error messages all the way back to the JSP source (which is tricky because of the JSP tag file mechanism) and to perform a more extensive evaluation.

References

1. Chen, S., Hong, D., Shen, V.Y.: An experimental study on validation problems with existing HTML webpages. In: Proc. International Conference on Internet Computing, ICOMP 2005 (June 2005)
2. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
3. Doh, K.-G., Kim, H., Schmidt, D.A.: Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 256–272. Springer, Heidelberg (2009)
4. Goldfarb, C.F.: The SGML Handbook. Oxford University Press, Oxford (1991)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
6. Kirkegaard, C., Møller, A.: Static analysis for java servlets and JSP. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 336–352. Springer, Heidelberg (2006)
7. Minamide, Y.: Static approximation of dynamically generated Web pages. In: Proc. 14th International Conference on World Wide Web, WWW 2005, pp. 432–441. ACM, New York (May 2005)
8. Minamide, Y., Tozawa, A.: XML validation for context-free grammars. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 357–373. Springer, Heidelberg (2006)
9. Møller, A., Schwartzbach, M.I.: The design space of type checkers for XML transformation languages. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 17–36. Springer, Heidelberg (2005)
10. Møller, A., Schwarz, M.: HTML validation of context-free languages. Technical report, Department of Computer Science, Aarhus University (2011), `http://cs.au.dk/~amoeller/papers/htmlcfg/`
11. Nishiyama, T., Minamide, Y.: A translation from the HTML DTD into a regular hedge grammar. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 122–131. Springer, Heidelberg (2008)
12. Thiemann, P.: Grammar-based analysis of string expressions. In: Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI 2005 (2005)
13. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot – a Java optimization framework. In: Proc. IBM Centre for Advanced Studies Conference, CASCON 1999. IBM (November 1999)
14. Warmer, J., van Egmond, S.: The implementation of the Amsterdam SGML parser. Electronic Publishing 2(2), 65–90 (1988)