BRICS

Basic Research in Computer Science

BRICS RS-01-9 Brabrand et al.: Static Validation of Dynamically Generated HTML

Static Validation of Dynamically Generated HTML

Claus Brabrand
Anders Møller
Michael I. Schwartzbach

BRICS Report Series RS-01-9

ISSN 0909-0878 February 2001

Static Validation of Dynamically Generated HTML

Claus Brabrand Anders Møller
Michael I. Schwartzbach

BRICS,[*] Department of Computer Science
Ny Munkegade, building 540
8000 Aarhus C, Denmark
{brabrand,amoeller,mis}@brics.dk

Abstract

We describe a static analysis of `<bigwig>` programs that efficiently decides if all dynamically computed XHTML documents presented to the client will validate according to the official DTD. We employ two interprocedural flow analyses to construct a graph summarizing the possible documents. This graph is subsequently analyzed to determine validity of those documents.

1 Introduction

Increasingly, HTML documents are dynamically generated by scripts running on a Web server, for instance using PHP, ASP, or Perl. This makes it much harder for authors to guarantee that such documents are really *valid*, meaning that they conform to the official DTD for HTML 4.01 or XHTML 1.0 [7]. Static HTML documents can easily be validated by tools made available by W3C and others. So far, the best possibility for a script author is to validate the dynamic HTML documents after they have been produced at runtime. However, this is an incomplete and costly process which does not provide any static guarantees about the behavior of the script. Alternatively, scripts may be restricted to use a collection of pre-validated templates, but this is generally not sufficiently expressive.

[*]Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

We present a novel technique for static validation of dynamic XHTML documents that are generated by a script. Our work takes place in the context of the `<bigwig>` language [1, 8], which is a full-fledged programming language for developing interactive Web services. In `<bigwig>`, XHTML documents are first-class citizens that are subjected to computations like all other data values. We instrument the compiler with an interprocedural flow analysis that extracts a grammatical structure covering the class of XHTML documents that a given program may produce. Based on this information, the compiler statically determines if all documents in the given class conform to the DTD for XHTML 1.0. To accomplish this, we need to reformulate DTDs in a novel way that may be interesting in its own right. The analysis has efficiently handled all available examples. Our technique can be generalized to more powerful grammatical descriptions.

2 XHTML Documents in `<bigwig>`

XHTML documents are just XML trees. In the `<bigwig>` language, XML *fragments* are first-class data values. Fragments are more general since they may contain *gaps*, which are named placeholders that can be *plugged* with fragments and strings. When a complete document has been built, it can be shown to the client who subsequently continues the session. This very expressive plug-and-show mechanism is described in [8, 1]. Note that the number of gaps may both grow and shrink as the result of a plug operation. Also, gaps may appear in a non-local manner, as exemplified by the `what` gap being plugged with the fragment `<b>BRICS</b>` in the following simple example in the `<bigwig>` syntax:

```
service {
  html Cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolor=[color]>
      <[contents]>
    </body>
  </html>;

  html Greeting = <html>
    Hello <[who]>, welcome to <[what]>.
  </html>;

  html Person = <html>
```

```
    <i>Stranger</i>
  </html>;

  session Welcome() {
    html H;
    H = Cover<[color="#9966ff",
               contents=Greeting<[who=Person]];
    show H<[what=<html><b>BRICS</b></html>];
  }
}
```

Here, `color` is an *attribute gap* which can only be plugged with a string value. Fragments are delimited by `<html>`...`</html>`. For compatibility, we do not distinguish between HTML and XHTML.

The `<bigwig>` compiler already contains an interprocedural flow analysis that keeps track of gaps and input fields in fragments to enable type checking [8]. However, the validity of the resulting documents has not been considered before. Note that `<bigwig>` is as general as all other languages for producing XML trees, since it is possible to define for each different element a tiny fragment like:

```
<html><ul type=[type]><[items]></ul></html>
```

that corresponds to a constructor function. The typical use of larger fragments is mostly a convenience for the `<bigwig>` programmer.

XML Fragments

We now formally define an abstract XML fragment. We are given an alphabet $\mathbf{\Sigma}$ of characters, an alphabet $\mathbf{E}$ of element names, an alphabet $\mathbf{A}$ of attribute names, an alphabet $\mathbf{G}$ of fragment gap names, and an alphabet $\mathbf{H}$ of attribute gap names. For simplicity, all alphabets are assumed to be disjoint. An *XML fragment* is generated by Φ in the following grammar:

$$
\begin{array}{lll}
\Phi & \rightarrow \epsilon & \\
 & \rightarrow \bullet & \\
 & \rightarrow g & g \in \mathbf{G} \\
 & \rightarrow e(\Delta)\Phi & e \in \mathbf{E} \\
 & \rightarrow \Phi_1\Phi_2 & \\
\Delta & \rightarrow \epsilon & \\
 & \rightarrow (a = s) & a \in \mathbf{A},\ s \in \mathbf{\Sigma}^* \\
 & \rightarrow (a = h) & a \in \mathbf{A},\ h \in \mathbf{H} \\
 & \rightarrow \Delta_1\Delta_2 &
\end{array}
$$

Element attributes are generated by Δ. The $\bullet$ symbol represents an arbitrary sequence of character data. We ignore the actual data, since those are never constrained by DTDs, unlike attribute values which we accordingly represent explicitly. We introduce a function:

$$gaps : (\Phi \cup \Delta) \rightarrow 2^{\mathbf{G} \cup \mathbf{H}}$$

which gives the set of gap names occurring in a fragment or attribute list:

$$\begin{array}{rcl} gaps(\epsilon) & = & \emptyset \\ gaps(\bullet) & = & \emptyset \\ gaps(g) & = & \{g\} \\ gaps(e(\delta)\phi) & = & gaps(\delta) \cup gaps(\phi) \\ gaps(\phi_1\phi_2) & = & gaps(\phi_1) \cup gaps(\phi_2) \\ gaps(a = s) & = & \emptyset \\ gaps(a = h) & = & \{h\} \\ gaps(\delta_1\delta_2) & = & gaps(\delta_1) \cup gaps(\delta_2) \end{array}$$

Programs

We represent a `<bigwig>` program abstractly as a flow graph with atomic statements at each program point. The actual syntax for `<bigwig>` is very liberal and resembles C or Java code with control structures and functions. However, it is a simple task to extract the normalized representation. A program uses a set X of XML fragment variables and a set Y of string variables. The atomic statements are:

x_i = x_j;	(fragment variable assignment)
x_i = ϕ;	(fragment constant assignment)
y_i = y_j;	(string variable assignment)
y_i = s;	(string constant assignment)
y_i = $\bullet$;	(arbitrary string assignment)
x_i = x_j<[g=x_k];	(fragment gap plugging)
x_i = x_j<[h=y_k];	(attribute gap plugging)
show x_i;	(client interaction)

where $x \in X$ and $y \in Y$ for each x and y. The assignments have the obvious semantics. The plug statement replaces all occurrences of a named gap with the given value. The **show** statement implicitly plugs all remaining gaps with ϵ before the fragment is displayed to the client.

3 Summary Graphs

A program contains a finite collection of XML fragments that are identified through a mapping function:

$$\mathbf{f} : \mathbf{N} \rightarrow \Phi$$

where $\mathbf{N}$ is a finite set of fragment indices. A program also contains a finite collection of string constants, which we shall denote by $\mathcal{C} \subseteq \mathbf{\Sigma}^*$. We now define a *summary graph* as a triple:

$$G = (R, E, \alpha)$$

where $R \subseteq \mathbf{N}$ is a set of *roots*, $E \subseteq \mathbf{N} \times \mathbf{G} \times \mathbf{N}$ is a set of *edges*, and $\alpha : \mathbf{N} \times \mathbf{H} \rightarrow \mathcal{S}$ is a labeling function, where $\mathcal{S} = 2^{\mathcal{C}} \cup \{\bullet\}$. Intuitively, $\bullet$ denotes the set of all strings.

Each summary graph G defines a set of XML fragments, denoted $\mathcal{L}(G) \subseteq \Phi$. Intuitively, this set is obtained by unfolding the graph from each root while performing all possible pluggings enabled by the edges and the labeling function. Formally, we define:

$$\mathcal{L}(G) = \{\phi \in \Phi \mid \exists r \in R : G, r \vdash \mathbf{f}(r) \Rightarrow \phi\}$$

where the derivation relation $\Rightarrow$ is defined for templates as:

$$\frac{}{G, n \vdash \epsilon \Rightarrow \epsilon} \qquad \frac{}{G, n \vdash \bullet \Rightarrow \bullet}$$

$$\frac{(n, g, m) \in E \quad G, m \vdash \mathbf{f}(m) \Rightarrow \phi}{G, n \vdash g \Rightarrow \phi}$$

$$\frac{G \vdash \delta \Rightarrow \delta' \quad G, n \vdash \phi \Rightarrow \phi'}{G, n \vdash e(\delta)\phi \Rightarrow e(\delta')\phi'}$$

$$\frac{G, n \vdash \phi_1 \Rightarrow \phi_1' \quad G, n \vdash \phi_2 \Rightarrow \phi_2'}{G, n \vdash \phi_1\phi_2 \Rightarrow \phi_1'\phi_2'}$$

and for attribute lists as:

$$\frac{\alpha(h) \neq \bullet \quad s \in \alpha(h)}{G \vdash (a = h) \Rightarrow (a = s)}$$

$$\frac{\alpha(h) = \bullet \quad s \in \mathbf{\Sigma}^*}{G \vdash (a = h) \Rightarrow (a = s)}$$

$$\frac{G \vdash \delta_1 \Rightarrow \delta_1' \quad G \vdash \delta_2 \Rightarrow \delta_2'}{G \vdash \delta_1\delta_2 \Rightarrow \delta_1'\delta_2'}$$

4 Gap Track Analysis

To obtain sufficient precision of the validation analysis, we first perform an initial analysis that tracks the origins of gaps. The lattice is simply:

$$\mathcal{T} = (\mathbf{G} \cup \mathbf{H}) \to 2^{\mathbf{N}}$$

ordered by pointwise subset inclusion. For each program point ℓ we wish to compute an element of the derived lattice:

$$\mathit{TrackEnv}_\ell : X \to \mathcal{T}$$

which inherits its structure from $\mathcal{T}$. Each atomic statement defines a transfer function $\mathit{TrackEnv}_\ell \to \mathit{TrackEnv}_\ell$ which models its semantics in a forward manner. If the argument is χ, then the results are:

x_i `=` x_j`;`	$\chi[x_i \mapsto \chi(x_j)]$
x_i `=` ϕ`;`	$\chi[x_i \mapsto \mathit{tfrag}(\phi, n)]$, where ϕ has index n
x_i `=` x_j`<[`g`=`x_k`];`	$\chi[x_i = \mathit{tplug}(\chi(x_j), g, \chi(x_k))]$
x_i `=` x_j`<[`h`=`y_k`];`	$\chi[x_i = \mathit{tplug}(\chi(x_j), h, \lambda p.\emptyset)]$

where we make use of some auxiliary functions:

$$\mathit{tfrag}(\phi, n) = \lambda p.\text{if } p \in \mathit{gaps}(\phi) \text{ then } \{n\} \text{ else } \emptyset$$

$$\mathit{tplug}(\tau_1, p, \tau_2) = \lambda q.\text{if } p = q \text{ then } \tau_2(q) \text{ else } \tau_1(q) \cup \tau_2(q)$$

The *tfrag* function states that all gaps in the given fragment originates from just there. The *tplug* function adds all origins from the fragment being inserted and removes the existing origins for the gap being plugged. For the remaining statement types, the transfer function is the identity function. It is easy to see that all transfer functions are monotonic, so we can compute the least fixed point in the usual manner [6]. The end result is for each program point ℓ an environment $\mathit{track}_\ell : X \to \mathcal{T}$, which we use in the following as a conservative, upper approximation of the origins of the gaps.

5 Summary Graph Analysis

We wish to compute for every program point and for every variable a summary of its possible values. A set of XML fragments is represented by a summary graph and a set of string values by an element of $\mathcal{S}$.

Lattices

To perform a standard data flow analysis, we need both of these representations to be lattices. The set $\mathcal{S}$ is clearly a lattice, ordered by set inclusion and with $\bullet$ as an extra top element. The set of summary graphs, called $\mathcal{G}$, is also a lattice with the ordering defined by:

$$G_1 \sqsubseteq G_2 \;\;\Leftrightarrow\;\; R_1 \subseteq R_2 \;\wedge\; E_1 \subseteq E_2 \;\wedge\; \alpha_1 \sqsubseteq \alpha_2$$

where the ordering on $\mathcal{S}$ is lifted pointwise to labeling functions α. Clearly, both $\mathcal{S}$ and $\mathcal{G}$ are finite lattices. For each program point we wish to compute an element of the derived lattice:

$$Env_\ell = (X \to \mathcal{G}) \times (Y \to \mathcal{S})$$

which inherits its structure from the constituent lattices.

Transfer Functions

Each atomic statement defines a transfer function $Env_\ell \to Env_\ell$, which models its semantics. If the argument is the pair of functions (χ, γ) and ℓ is the entry program point of the statement, then the results are:

x_i = x_j;	$(\chi[x_i \mapsto \chi(x_j)], \gamma)$
x_i = ϕ;	$(\chi[x_i \mapsto \mathit{frag}(n)], \gamma)$, where ϕ has index n
y_i = y_j;	$(\chi, \gamma[y_i \mapsto \gamma(y_j)])$
y_i = s;	$(\chi, \gamma[y_i \mapsto \{s\}])$
y_i = $\bullet$;	$(\chi, \gamma[y_i \mapsto \bullet])$
x_i = x_j<[g=x_k];	$(\chi[x_i \mapsto \mathit{gplug}(\chi(x_j), g, \chi(x_k), \mathit{track}_\ell(x_j))], \gamma)$
x_i = x_j<[h=y_k];	$(\chi[x_i \mapsto \mathit{hplug}(\chi(x_j), h, \gamma(y_k), \mathit{track}_\ell(x_j))], \gamma)$
show x_i;	(χ, γ)

where we make use of some auxiliary functions:

$$\mathit{frag}(n) = (\{n\}, \emptyset, \lambda(m,h).\emptyset)$$

$$\begin{aligned}\mathit{gplug}(G_1, g, G_2, \tau) = (&R_1,\\ &E_1 \cup E_2 \cup\\ &\quad \{(n, g, m) \mid n \in \tau(g) \;\wedge\; m \in R_2\},\\ &\alpha_1 \sqcup \alpha_2)\end{aligned}$$

$$\begin{aligned} hplug(G,h,s,\tau) = (&R, E,\\ &\lambda(n,h').\text{if } n \in \tau(h) \text{ then } \alpha(n,h') \sqcup s\\ &\qquad\qquad\qquad\qquad \text{else } \alpha(n,h')) \end{aligned}$$

where $G_i = (R_i, E_i, \alpha_i)$ and $G = (R, E, \alpha)$. The *frag* function constructs a tiny summary graph whose language contains only the given fragment. The *gplug* function joins the two summary graphs and adds edges from all relevant fragment gaps to the roots of the summary graph being inserted. The *hplug* function adds additional string values to the relevant attribute gaps. A careful inspection shows that all transfer functions are monotonic.

The Analysis

Since we are working with monotonic functions on finite lattices, we can use standard techniques to compute a least fixed point [6]. The proof of soundness is omitted here, but it is similar to the one presented in [8]. The end result is for each program point ℓ an environment $summary_\ell : X \to \mathcal{G}$ such that $\mathcal{L}(summary_\ell(x_i))$ contains all possible XML fragments that x_i may contain at ℓ. Those fragments that are associated with **show** statements are required to validate. First, we must model the implicit plugging of empty fragments and strings into the remaining gaps, so for the statement:

show x_i;

with entry program point q, the summary graph that must validate with respect to the XHTML DTD is:

$$close(summary_\ell(x_i), track_\ell(x_i))$$

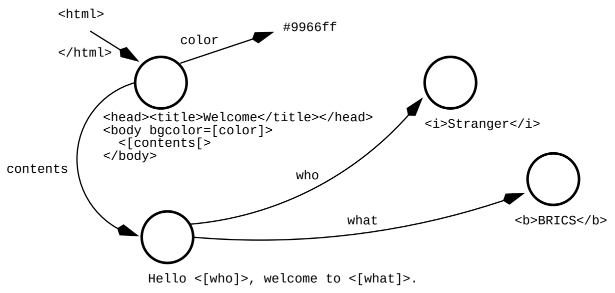
where *close* is defined by:

$$\begin{aligned} close(G,\tau) = (&R,\\ &E \cup \{(n,g,m_\epsilon) \mid n \in \tau(g)\},\\ &\lambda(n,h).\text{if } n \in \tau(h) \text{ then } \alpha(n,h) \sqcup \{\epsilon\}\\ &\qquad\qquad\qquad\qquad \text{else } \alpha(n,h)) \end{aligned}$$

where $G = (R, E, \alpha)$ and it is assumed that $\mathbf{f}(m_\epsilon) = \epsilon$. The *close* function adds edges to an empty fragment for all remaining fragments gaps, and adds the empty string as a possibility for all remaining attribute gaps.

The Example Revisited

For the small `<bigwig>` example in Section 2, the summary graph describing the document being shown to the client is inferred to be:



As expected for this simple case, the language of the summary graph contains exactly the single fragment actually being computed. Note that the XHTML fragment is implicitly completed with the `<html>` element.

6 An Abstract DTD for XHTML

XHTML 1.0 is described by an official DTD [7]. We use a more abstract formalism which is in some ways more restrictive and in others strictly more expressive. In any case, the DTD for XHTML 1.0 can be captured along with some restrictions that merely appear as comments in the official version. We define an abstract DTD to be a quintuple:

$$D = (\mathcal{N}, \rho, \mathcal{A}, \mathcal{E}, \mathcal{F})$$

where $\mathcal{N} \subseteq \mathbf{E}$ is a set of *declared* element names, $\rho \in \mathcal{N}$ is a *root* element name, $\mathcal{A} : \mathcal{N} \to 2^{\mathbf{A}}$ is an $\mathcal{N}$-indexed family of attribute name declarations, $\mathcal{E} : \mathcal{N} \to 2^{\mathcal{N}^\bullet}$ a family of element name declarations, and $\mathcal{F} : \mathbf{E} \to \Psi$ a family of formulas. Here, $\mathcal{N}^\bullet = \mathcal{N} \cup \{\bullet\}$, where $\bullet$ represents arbitrary character data. A formula has the syntax:

$$\begin{array}{lll}
\Psi & \to \Psi \wedge \Psi & \\
 & \to \Psi \vee \Psi & \\
 & \to \neg\Psi & \\
 & \to \mathit{true} & \\
 & \to \mathbf{attr}(a) & a \in \mathbf{A} \\
 & \to \mathbf{elem}(e) & e \in \mathcal{N}^\bullet \\
 & \to \mathbf{value}(a, \{s_1, \ldots, s_k\}) & a \in \mathbf{A},\ k \geq 1,\ s_i \in \mathbf{\Sigma}^* \\
 & \to \mathbf{order}(e_1, e_2) & e_i \in \mathcal{N}^\bullet
\end{array}$$

We define the language of D as follows:

$$\mathcal{L}(D) = \{\rho(\delta)\phi \mid D \models \rho(\delta)\phi\}$$

where the acceptance relation $\models$ on fragments is defined by:

$$\frac{}{D \models \epsilon} \qquad \frac{}{D \models \bullet} \qquad \frac{}{D \models g}$$

$$\frac{D \models \phi_1 \qquad D \models \phi_2}{D \models \phi_1\phi_2}$$

$$\frac{\begin{array}{cc} names(\delta) \subseteq \mathcal{A}(e) & D, \delta, \phi \models \mathcal{F}(e) \\ set(\phi) \subseteq \mathcal{E}(e) & D \models \phi \end{array}}{D \models e(\delta)\phi}$$

On formulas, the $\models$ relation is defined by:

$$\frac{D, \delta, \phi \models \psi_1 \quad D, \delta, \phi \models \psi_2}{D, \delta, \phi \models \psi_1 \wedge \psi_2}$$

$$\frac{D, \delta, \phi \models \psi_1}{\phi \models \psi_1 \vee \psi_2} \quad \frac{D, \delta, \phi \models \psi_2}{\phi \models \psi_1 \vee \psi_2}$$

$$\frac{}{D, \delta, \phi \models \mathit{true}} \quad \frac{D, \delta, \phi \not\models \psi}{D, \delta, \phi \models \neg\psi}$$

$$\frac{a \in names(\delta)}{D, \delta, \phi \models \mathbf{attr}(a)} \quad \frac{before(word(\phi), e_1, e_2)}{D, \delta, \phi \models \mathbf{order}(e_1, e_2)}$$

$$\frac{a \notin names(\delta)}{D, \delta, \phi \models \mathbf{value}(a, \{s_1, \ldots, a_k\})}$$

$$\frac{(a, s_i) \in atts(\delta) \quad 1 \leq i \leq k}{D, \delta, \phi \models \mathbf{value}(a, \{s_1, \ldots, s_k\})}$$

$$\frac{exists(word(\phi), e)}{D, \delta, \phi \models \mathbf{elem}(e)}$$

The function *names* is:

$$\begin{array}{rcl} names(\epsilon) & = & \emptyset \\ names(a = s) & = & \{a\} \\ names(a = h) & = & \{a\} \\ names(\delta_1\delta_2) & = & names(\delta_1) \cup names(\delta_2) \end{array}$$

the function *atts* is:

$$\begin{array}{rcl}
atts(\epsilon) & = & \emptyset \\
atts(a = s) & = & \{(a, s)\} \\
atts(a = h) & = & \{(a, h)\} \\
atts(\delta_1\delta_2) & = & atts(\delta_1) \cup atts(\delta_2)
\end{array}$$

the function *set* is:

$$\begin{array}{rcl}
set(\epsilon) & = & \emptyset \\
set(\bullet) & = & \{\bullet\} \\
set(g) & = & \emptyset \\
set(e(\delta)\phi) & = & \{e\} \\
set(\phi_1\phi_2) & = & set(\phi_1) \cup set(\phi_2)
\end{array}$$

the function *word* is:

$$\begin{array}{rcl}
word(\epsilon) & = & \epsilon \\
word(\bullet) & = & \bullet \\
word(g) & = & \epsilon \\
word(e(\delta)\phi) & = & e \\
word(\phi_1\phi_2) & = & word(\phi_1)word(\phi_2)
\end{array}$$

and the auxiliary predicates are:

$$\begin{array}{rcl}
exists(w_1 \cdots w_k, e) & \equiv & \exists 1 \leq i \leq k : w_i = e \\
before(w_1 \cdots w_k, e_1, e_2) & \equiv & \forall 1 \leq i, j \leq k : \\
& & \quad w_i = e_1 \wedge w_j = e_2 \Rightarrow i \leq j
\end{array}$$

Two common abbreviations are $\mathbf{unique}(e) \equiv \mathbf{order}(e, e)$ ("e occurs at most once") and $\mathbf{exclude}(e_1, e_2) \equiv \neg(\mathbf{elem}(e_1) \wedge \mathbf{elem}(e_2))$ ("e_1 and e_2 exclude each other").

Standard DTDs use restricted regular expressions to describe content sequences. Instead, we use boolean combinations of four basic predicates, each of which corresponds to a simple regular language. This is less expressive, since for example we cannot express that a content sequence must have exactly three occurrences of a given element. It is also, however, more expressive than DTDs since we allow the requirements on contents and attributes to be mixed in a formula. While the two formalism are thus theoretically incomparable, our experience is that actual DTDs are within the scope of our abstract notion.

Examples for XHTML

The DTD for XHTML 1.0 can easily be expressed in our formalism. The root element ρ is `html` and some examples of declarations and formulas are:

$$\begin{array}{lcl}
\mathcal{A}(\texttt{html}) & = & \{\texttt{xmlns}, \texttt{lang}, \texttt{xml:lang}, \texttt{dir}\} \\
\mathcal{E}(\texttt{html}) & = & \{\texttt{head}, \texttt{body}\} \\
\mathcal{F}(\texttt{html}) & = & \textbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \textbf{elem}(\texttt{head}) \wedge \\
 & & \textbf{elem}(\texttt{body}) \wedge \textbf{unique}(\texttt{head}) \wedge \\
 & & \textbf{unique}(\texttt{body}) \wedge \textbf{order}(\texttt{head}, \texttt{body}) \\
 & & \\
\mathcal{A}(\texttt{head}) & = & \{\texttt{lang}, \texttt{xml:lang}, \texttt{dir}, \texttt{profile}\} \\
\mathcal{E}(\texttt{head}) & = & \{\texttt{script}, \texttt{style}, \texttt{meta}, \texttt{link}, \texttt{object}, \texttt{isindex} \\
 & & \ \texttt{title}, \texttt{base}\} \\
\mathcal{F}(\texttt{head}) & = & \textbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \textbf{elem}(\texttt{title}) \wedge \\
 & & \textbf{unique}(\texttt{title}) \wedge \textbf{unique}(\texttt{base}) \\
 & & \\
\mathcal{A}(\texttt{input}) & = & \{\texttt{id}, \texttt{class}, \texttt{style}, \texttt{title}, \texttt{lang}, \texttt{xml:lang}, \\
 & & \ \texttt{dir}, \texttt{onclick}, \texttt{ondblclick}, \texttt{onmousedown}, \\
 & & \ \texttt{onmouseup}, \texttt{onmouseover}, \texttt{onmousemove}, \\
 & & \ \texttt{onmouseout}, \texttt{onkeypress}, \texttt{onkeydown}, \\
 & & \ \texttt{onkeyup}, \texttt{type}, \texttt{name}, \texttt{value}, \texttt{checked}, \\
 & & \ \texttt{disabled}, \texttt{readonly}, \texttt{size}, \texttt{maxlength}, \\
 & & \ \texttt{src}, \texttt{alt}, \texttt{usemap}, \texttt{tabindex}, \texttt{accesskey}, \\
 & & \ \texttt{onfocus}, \texttt{onblur}, \texttt{onselect}, \texttt{onchange}, \\
 & & \ \texttt{accept}, \texttt{align}\} \\
\mathcal{E}(\texttt{input}) & = & \emptyset \\
\mathcal{F}(\texttt{input}) & = & \textbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \\
 & & \textbf{value}(\texttt{checked}, \{\texttt{checked}\}) \wedge \\
 & & \textbf{value}(\texttt{disabled}, \{\texttt{disabled}\}) \wedge \\
 & & \textbf{value}(\texttt{readonly}, \{\texttt{readonly}\}) \wedge \\
 & & \textbf{value}(\texttt{align}, \{\texttt{top}, \texttt{middle}, \texttt{bottom}, \texttt{left}, \texttt{right}\}) \wedge \\
 & & \textbf{value}(\texttt{type}, \{\texttt{text}, \texttt{password}, \texttt{checkbox}, \texttt{radio}, \\
 & & \qquad \texttt{submit}, \texttt{reset}, \texttt{file}, \texttt{hidden}, \texttt{image}, \\
 & & \qquad \texttt{button}\}) \wedge \\
 & & (\textbf{value}(\texttt{type}, \{\texttt{submit}, \texttt{reset}\}) \vee \textbf{attr}(\texttt{name}))
\end{array}$$

In five instances we were able to express requirements that were only stated as comments in the official DTD, such as the last conjunct in $\mathcal{F}(\texttt{input})$. The full description of XHTML is available at `http://www.brics.dk/bigwig/xhtml/`.

Exceptions in `<bigwig>`

In one situation does `<bigwig>` allow non-standard XHTML notation. In the official DTD, the `ul` element is required to contain at least one `li` element. This is inconvenient, since the items of a list are often generated iteratively from a vector that may be empty. To facilitate this style of programming, `<bigwig>` allows empty `ul` elements but removes them at runtime before the XHTML is sent to the client. Accordingly, the abstract DTD that we employ differs from the official one in this respect. Similar exceptions are allowed for other kinds of lists and for tables.

7 Validating Summary Graphs

For every **`show`** statement, the flow analysis computes a summary graph $G = (R, E, \alpha)$. We must now for all such graphs decide the validation requirement:

$$\mathcal{L}(G) \subseteq \mathcal{L}(D)$$

for an abstract DTD $D = (\mathcal{N}, \rho, \mathcal{A}, \mathcal{E}, \mathcal{F})$. The root element name requirement of D is first checked separately by verifying that:

$$\forall r \in R : \exists \delta \in \Delta, \phi \in \Phi : \ \mathbf{f}(r) = \rho(\delta)\phi$$

Then for each sub-ragment $e(\delta)\phi$ of a fragment with index n in G we perform the following checks:

- $e \in \mathcal{N}$ (the element is defined)
- $names(\delta) \subseteq \mathcal{A}(e)$ (the attributes are declared)
- $occurs(n, \phi) \subseteq \mathcal{E}(e)$ (the content is declared)
- $D \vdash \mathcal{F}(e)$ (the constraint is satisfied)

The relation $\vdash$ is given by:

$$\frac{D \vdash \psi_1 \quad D \vdash \psi_2}{D \vdash \psi_1 \wedge \psi_2}$$

$$\frac{D \vdash \psi_1}{D \vdash \psi_1 \vee \psi_2} \qquad \frac{D \vdash \psi_2}{D \vdash \psi_1 \vee \psi_2}$$

$$\frac{D \not\vdash \psi}{D \vdash \neg \psi}$$

$$\frac{a \in \mathit{names}(\delta)}{D \vdash \mathbf{attr}(a)} \qquad \frac{e \in \mathit{occurs}(n,\phi)}{D \vdash \mathbf{elem}(e)}$$

$$\frac{a \notin \mathit{names}(\delta)}{D \vdash \mathbf{value}(a, s_1, \ldots, s_k)}$$

$$\frac{(a, s_i) \in \mathit{atts}(\delta) \quad 1 \leq i \leq k}{D \vdash \mathbf{value}(a, s_1, \ldots, s_k)}$$

$$\frac{(a, h) \in \mathit{atts}(\delta) \quad \alpha(n,h) \subseteq \{s_1, \ldots, s_k\}}{D \vdash \mathbf{value}(a, s_1, \ldots, s_k)}$$

$$\frac{\mathit{order}(n, \phi, e_1, e_2)}{D \vdash \mathbf{order}(e_1, e_2)}$$

where *occurs* is the least function satisfying:

$$\begin{array}{rcl}
\mathit{occurs}(n,\epsilon) & = & \emptyset \\
\mathit{occurs}(n,\bullet) & = & \{\bullet\} \\
\mathit{occurs}(n,g) & = & \displaystyle\bigcup_{(n,g,m)\in E} \mathit{occurs}(m,\mathbf{f}(m)) \\
\mathit{occurs}(n,e(\delta)\phi) & = & \{e\} \\
\mathit{occurs}(n,\phi_1\phi_2) & = & \mathit{occurs}(n,\phi_1) \cup \mathit{occurs}(n,\phi_2)
\end{array}$$

and *order* is the most restrictive function satisfying:

$$\begin{array}{rcl}
\mathit{order}(n,\epsilon,e_1,e_2) & = & \mathit{true} \\
\mathit{order}(n,\bullet,e_1,e_2) & = & \mathit{true} \\
\mathit{order}(n,g,e_1,e_2) & = & \displaystyle\bigwedge_{(n,g,m)\in E} \mathit{order}(m,\mathbf{f}(m),e_1,e_2) \\
\mathit{order}(n,e(\delta)\phi,e_1,e_2) & = & \mathit{true} \\
\mathit{order}(n,\phi_1\phi_2,e_1,e_2) & = & \mathit{order}(n,\phi_1,e_1,e_2) \wedge \\
 & & \mathit{order}(n,\phi_2,e_1,e_2) \wedge \\
 & & \neg\,(e_2 \in \mathit{occurs}(n,\phi_1) \wedge \\
 & & \quad e_1 \in \mathit{occurs}(n,\phi_2))
\end{array}$$

In the implementation we ensure termination by applying memoization to the numerous calls to *occurs* and *order*.

Note that the validation algorithm is sound and complete with respect to summary graphs: if a graph is rejected, then its language contains a fragment that is not in the language of the abstract DTD. Thus, in the whole validation analysis the only source of imprecision is the flow analysis that constructs the summary graph.

8 Experiments

The validation analysis has been fully implemented as part of the `<bigwig>` system. It has then been applied to all available benchmarks, some of which are shown in the following table:

Name	Lines	Fragments	Size	Shows	Sec
`chat`	65	3	(0,5)	2	0.1
`guess`	75	6	(0,3)	6	0.1
`calendar`	77	5	(8,6)	2	0.1
`xbiff`	561	18	(4,12)	15	0.1
`webboard`	1,132	37	(34,18)	25	0.6
`cdshop`	1,709	36	(6,23)	25	0.5
`jaoo`	1,941	73	(49,14)	17	2.4
`bachelor`	2,535	137	(146,64)	15	8.2
`courses`	4,465	57	(50,45)	17	1.3
`eatcs`	5,345	133	(35,18)	114	6.7

The entries for each benchmark are its name, the lines of code derived from a pretty print of the source with all macros expanded, the number of fragments, the size $(|E|, |\alpha|)$ of the largest summary graph, the number of program points with **`show`** statements, and the analysis time in seconds (on an 800 MHz Pentium III Linux PC).

The analysis found numerous validation errors in all benchmarks, which could then be fixed to yield flawless services. No false errors were reported. As seen in the table above, the enhanced compiler remains efficient and practical. The `bachelor` service constructs unusually complicated documents, which explains its high complexity.

Error Diagnostics

The `<bigwig>` compiler provides detailed diagnostic messages in case of validation errors. For the flawed example:

```
service {
  html Cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolo=[color]>
      <table><[contents]></table>
    </body>
  </html>;
  html Greeting = <html>
    <td>Hello <[who]>,<br clear=[clear]>
        welcome to <[what]>.
    </td>
  </html>;

  html Person = <html>
    <i>Stranger</i>
  </html>;

  session Welcome() {
    html H;
    H = Cover<[color="#9966ff",
               contents=Greeting<[who=Person],
               clear="righ"];
    show H<[what=<html><b>BRICS</b></html>];
  }
}
```

the compiler generates the following messages:

```
brics.wig:4:
  warning: illegal attribute 'bgcolo' in 'body'
  fragment: <body bgcolo=[color]><form>...</form></body>

brics.wig:5:
  warning: possible illegal subelement 'td' of 'table'
  fragment: <table><[contents]></table>
  contents: td

brics.wig:10:
  warning: possible element constraint violation at 'br'
  fragment: <br clear=[clear]/>
  constraint: value(clear,{left,all,right,clear,none})
```

9 Related Work

There are other languages for constructing XML documents that also consider validity. The Xduce language [2, 3] is a functional language in which XML fragments are data types, with a constructor for each element name and pattern matching for deconstruction. A type is a regular expression over $\mathbf{E}^{\bullet}$. Type inference for pattern variables is supported. In comparison, we have a richer language and consequently need more expressive types that also describe the existence and capabilities of gaps. It seems unlikely that anything simpler than summary graphs would work. Also, we do not rely on type annotations.

Since we perform an interprocedural flow analysis, we obtain a high degree of polymorphism that is difficult to express in a traditional type system. The XMλ language [5] compares similarly to our approach.

10 Extensions and Future Work

Instead of our four basic predicates we could allow general regular expressions over the alphabet $\mathbf{E}^\bullet$. We could then still validate a summary graph, but this would reduce to deciding if a general context-free language is a subset of a regular language, which has an unwieldy algorithm compared to the simple transitive closures that we presently rely upon. Fortunately, our restricted regular languages appear sufficient. It is also possible to include many features from a richer XML schema language such as DSD [4], in particular regular expression constraints on attribute values and context dependency. Finally, we could enrich `<bigwig>` with a set of operators for combining and deconstructing XML fragments. All such ideas readily permit analysis by means of summary graphs.

11 Conclusion

We have combined a standard interprocedural flow analysis with a generalized validation algorithm to enable the `<bigwig>` compiler to guarantee that all HTML or XHTML documents shown to the client are valid according to the official DTD. Our technique generalizes in a straightforward manner to arbitrary XML languages that can be described by DTDs. In fact, we can even handle more expressive grammatical formalisms. The analysis has proved to be feasible for programs of realistic sizes. All this lends further support to the unique design of dynamic documents in `<bigwig>`. Since our algorithm is parameterized with an abstract DTD, it is possible to customize the validation. A useful example is an abstract DTD that describes the subset of XHTML that works safely on both the Explorer and Netscape browser.

References

[1] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. Submitted for publication. Available from `http://www.brics.dk/bigwig/`.

[2] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Workshop on the Web and Databases (WebDB2000)*, 2000.

[3] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *Symposium on Principles of Programming Languages (POPL'01)*. ACM, 2001.

[4] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. DSD: A schema language for XML. In *Workshop on Formal Methods in Software Practice (FMSP'00)*. ACM, 2000.

[5] Erik Meijer and Mark Shields. XMλ: A functional language for constructing and manipulating XML documents. Draft, 1999.

[6] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[7] Steven Pemberton et al. *XHTML 1.0: The Extensible HyperText Markup Language*. W3C, January 2000. W3C Recommendation, `http://www.w3.org/TR/xhtml1`.

[8] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic Web documents. In *Principles of Programming Languages (POPL'00)*. ACM, 2000.