Static Validation of Dynamically Generated HTML

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach
BRICS
Department of Computer Science
University of Aarhus, 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 data-flow analyses to construct a graph summarizing the possible documents. This graph is subsequently analyzed to determine validity of those documents. By evaluating the technique on a number of realistic benchmarks, we demonstrate that it is sufficiently fast and precise to be practically useful.

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 [9]. 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.

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 [2, 10], 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 data-flow analysis that extracts a grammatical structure, called a *summary graph*, 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. Furthermore, our technique can be generalized to more powerful grammatical descriptions.

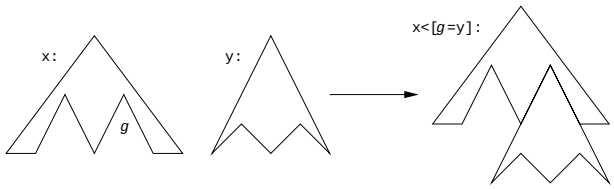
PASTE'01, June 18-19, 2001, Snowbird, Utah, USA..

Outline

First, in Section 2, we give a brief introduction to dynamically generating XHTML documents in the <bigwig> language. Section 3 formally defines the notion of summary graphs. In Sections 4 and 5, the two parts of the data-flow analysis are specified. Then, in Section 6, a notion of abstract DTDs is defined and used for specifying XHTML 1.0. Section 7 describes the algorithm for validating summary graphs with respect to abstract DTDs. In Section 8 we evaluate our implementation on ten <bigwig> programs. Finally, in Sections 9 and 10 we briefly describe related techniques and plans and ideas for future work.

2. XHTML DOCUMENTS IN <bigwig>

XHTML documents are just XML trees. In the <bigwig> language, XML *templates* are first-class data values that may be passed and stored as any other values. Templates are more general than XML trees since they may contain *gaps*, which are named placeholders that can be *plugged* with templates and strings: If x is an XML template with a gap named *g* and y is another XML template or a text string, then the plug operation, x<[*g*=y], results in a new template which is copy of x where a copy of y has been inserted into the *g* gap:



A <bigwig> service consists of a number of *sessions*. A session thread can be invoked by a client who is subsequently guided through a number of interactions, controlled by the service code on the server. A *document* is a template where all gaps have been filled. When a complete XHTML document has been built on the server, it can be shown to the client who fills in the input fields, selects menu options, etc., and then continues the session by submitting the input to the session thread.

This plug-and-show mechanism provides a very expressive way of dynamically constructing Web documents. It is described in more detail in [10, 2] where a thorough comparison with other mechanisms is given and other aspects of <bigwig> are described. Since templates can be plugged into templates, these are *higher-order* templates, as opposed to the less flexible templates in the Mawl language [6, 1] where only strings can be plugged in.

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 template BRICS in the following simple example in the actual <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>];
  }
}
```

This service contains four constant templates and a session which when invoked will assemble a document using plug operations and show it to the client. Note that color is an *attribute gap* which can only be plugged with a string value, while the other gaps can also be plugged with templates. Constant templates are delimited by <html>...</html>. Implicitly, the mandatory surrounding <html> element is added to a document before being shown. Also, <head>, <title>, and <body> elements and a form with a default submit button is added if not already present. To simplify the presentation, we do not distinguish between HTML and XHTML since there are only minor syntactical differences. In the implementation, we allow HTML syntax but convert it to XHTML.

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 template like:

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

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

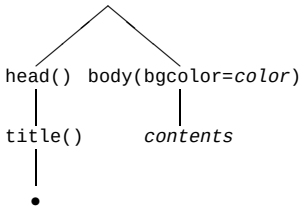
The <bigwig> compiler already contains an interprocedural data-flow analysis that keeps track of gaps and input fields in templates to enable type checking of plug and show operations [10]. That analysis statically ensures that the gaps are present when performing a plug operation and that the input fields in the documents being shown match the code that receives the values. However, the validity of the documents being shown has not been considered before, neither for <bigwig> or—to our knowledge—for any other programming language with such a flexible document construction mechanism.

XML Templates

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

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

An XML template is a list of ordered trees where the internal nodes are *elements* with *attributes* and the leaves are either empty nodes, *character data* nodes, or *gap* nodes. 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. As an example, we view the cover template abstractly as follows if we ignore character data nodes consisting only of white-space:



We introduce a function:

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

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

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

A template ϕ with a unique root element and with $gaps(\phi) = \emptyset$ is considered a complete *document*.

Programs

We represent a <bigwig> program abstractly as a control-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. For <bigwig> it is a simple task to extract the normalized representation. If the underlying language had a richer control structure, for instance with inheritance and virtual methods or higher-order functions, we would need a preliminary control-flow analysis to provide the control-flow graph.

A program uses a set X of XML template variables and a set Y of string variables. The atomic statements are:

x_i = x_j;	(template variable assignment)
x_i = ϕ;	(template 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];	(template 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 template is displayed to the client. Also, the template is implicitly plugged into a wrapper template like the following:

```
<html>
  <head><title></title></head>
  <body>
    <form action="...">
      <[doc]>
      <input type="submit" value="continue">
    </form>
  </body>
</html>
```

for completing the document and adding a "continue" button. The `<head>`, `<title>`, `<body>`, and `<input>` elements are of course only added if not already present. Since we here ignore input fields in documents, the **receive** part of **show** statements is omitted in this description.

3. SUMMARY GRAPHS

Given a program control-flow graph, we wish to extract a finite representation of all the templates that can possibly be constructed at runtime. A program contains a finite collection of constant XML templates that are identified through a mapping function:

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

where $\mathbf{N}$ is the finite set of indices of the templates occurring in the program. 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 $\alpha : \mathbf{N} \times \mathbf{H} \to \mathcal{S}$ is an attribute 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 templates, which is called the *language* of G and is denoted $\mathcal{L}(G)$. 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:

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

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

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

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

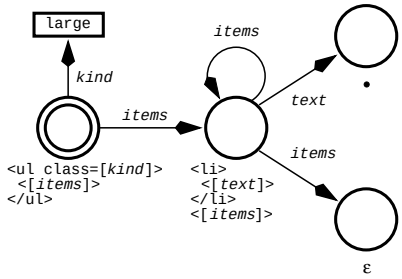
and for attribute lists as:

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

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

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

As an example, consider the following summary graph consisting of four template nodes, four plug edges, and a single attribute labeling:



Template nodes, root nodes, and attribute labels are drawn as circles, double circles, and boxes, respectively. The language of this summary graph is the set of all `ul` lists of class `large` with one or more character data items.

4. GAP TRACK ANALYSIS

To obtain sufficient precision of the actual validation analysis, we first perform an initial analysis that tracks the origins of gaps. We show in Section 5 exactly why this information is necessary.

Lattices

The lattice for this analysis 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 $\mathcal{T}$. Intuitively, an element of this lattice tells us for a given variable x and a gap name g whether or not g can occur in the value of x, and if it can, which constant templates g can originate from.

Transfer Functions

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 of applying this transfer function 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:

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

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

For the remaining statement types, the transfer function is the identity function. The *tfrag* function states that all gaps in the given template originates from just that template. The *tplug* function adds all origins from the template being inserted and removes the existing origins for the gap being plugged.

The Analysis

It is easy to see that all transfer functions are monotonic, so we can compute the least fixed point iteratively in the usual manner [8]. The end result is for each program point ℓ an environment $track_\ell : X \to \mathcal{T}$, which we use in the following as a conservative, upper approximation of the origins of the gaps. We omit the proof of correctness.

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 templates 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 a 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 = •;	$(\chi, \gamma[y_i \mapsto \bullet])$
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)

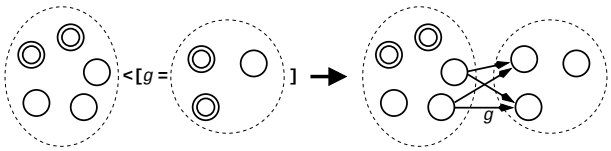
where we make use of some auxiliary functions:

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

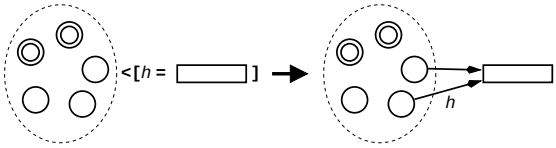
$$\mathit{gplug}(G_1, g, G_2, \tau) = (R_1, E_1 \cup E_2 \cup \{(n,g,m) \mid n \in \tau(g) \wedge m \in R_2\}, \alpha_1 \sqcup \alpha_2)$$

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

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



The *hplug* function adds additional string values to the relevant attribute gaps:



We are now in a position to point out the need for the gap track analysis specified in Section 4. Without that initial analysis, the τ argument to *gplug* and *hplug* would always have to be the set $\mathbf{N}$ of all constant template indices to maintain soundness. Plugging a value into a gap g would then be modeled by adding an edge from all nodes having a g gap, even from nodes that originate from completely unrelated parts of the source code or nodes where the g gaps already have been filled. For instance, it is likely that a program building lists as in the summary graph example in Section 4 would contain other templates with a gap named `items`. Requiring each gap name to appear only in one constant template would solve the problem, but such a restriction would limit the flexibility of the document construction mechanism significantly. Hence, we rely on a program analysis to disregard the irrelevant nodes when adding plug edges.

The Analysis

Since we are working with monotonic functions on finite lattices, we can again use standard iterative techniques to compute a least fixed point [8]. The proof of soundness is omitted here, but it is similar to the one presented in [10]. The end result is for each program point ℓ an environment $\mathit{summary}_\ell : X \to \mathcal{G}$ such that $\mathcal{L}(\mathit{summary}_\ell(x_i))$ contains all possible XML templates that x_i may contain at ℓ. Those templates that are associated with **show** statements are required to validate with respect to the XHTML specification. We assume that the implicitly surrounding continue-button wrapper from Section 2 has been added already. Still, we must model the implicit plugging of empty templates 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:

$$\mathit{close}(\mathit{summary}_\ell(x_i), \mathit{track}_\ell(x_i))$$

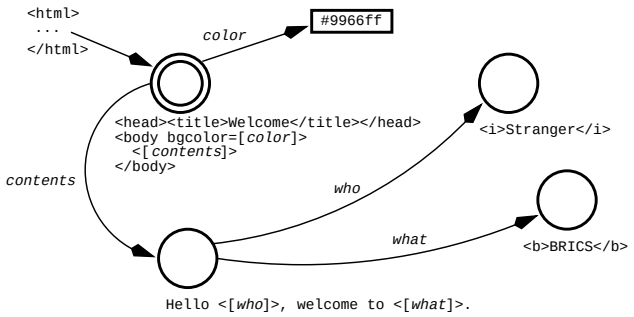
where *close* is defined by:

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

where $G = (R, E, \alpha)$ and it is assumed that $\mathbf{f}(m_\epsilon) = \epsilon$. The $close$ function adds edges to an empty template for all remaining templates 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 template actually being computed: Note that the XHTML template is implicitly completed with the `<html>` fragment.

6. AN ABSTRACT DTD FOR XHTML

XHTML 1.0 is described by an official DTD [9]. 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 $\mathcal{F} : \mathcal{N} \to \Psi$ a family of formulas. We let $\mathcal{N}^\bullet = \mathcal{N} \cup \{\bullet\}$, where $\bullet$ represents arbitrary character data.

Intuitively, an abstract DTD consists of a number of element declarations whereof one is designated as the root. Each element declaration consists of an element name, a set of allowed attribute names, a set of allowed contents, and a formula constraining the use of the element with respect to its attribute values and contents. A formula has the syntax:

$$\begin{array}{lll}
\Psi \to \Psi \wedge \Psi & & \\
\quad \to \Psi \vee \Psi & & \\
\quad \to \neg\Psi & & \\
\quad \to \mathit{true} & & \\
\quad \to \mathbf{attr}(a) & a \in \mathbf{A} & \\
\quad \to \mathbf{content}(c) & c \in \mathcal{N}^\bullet & \\
\quad \to \mathbf{order}(c_1, c_2) & c_i \in \mathcal{N}^\bullet & \\
\quad \to \mathbf{value}(a, \{s_1, \ldots, s_k\}) & a \in \mathbf{A},\ k \geq 1,\ s_i \in \mathbf{\Sigma}^* &
\end{array}$$

We define the *language* of D as follows:

$$\mathcal{L}(D) = \{\rho(\delta)\phi \mid D \models \rho(\delta)\phi \wedge \mathit{gaps}(\phi) = \emptyset\}$$

That is, the language is the set of documents where the root element is ρ and the acceptance relation $\models$ is satisfied. This relation is defined inductively on templates as follows:

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

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

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

For each element, it is checked that its attributes and contents are declared and that the associated formula is satisfied. The auxiliary functions names and set are formally defined by:

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

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

On formulas, the $\models$ relation is defined relative to the attributes and contents of an element:

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

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

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

$$\frac{a \in \mathit{names}(\delta)}{D, \delta, \phi \models \mathbf{attr}(a)} \qquad \frac{\mathit{exists}(\mathit{word}(\phi), c)}{D, \delta, \phi \models \mathbf{content}(c)}$$

$$\frac{\mathit{before}(\mathit{word}(\phi), c_1, c_2)}{D, \delta, \phi \models \mathbf{order}(c_1, c_2)}$$

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

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

The $\mathbf{attr}(a)$ formula checks whether an attribute of name a is present, and $\mathbf{content}(c)$ checks whether c occurs in the contents. The $\mathbf{value}(a, \{s_1, \ldots, s_k\})$ formula checks whether an a attribute has one of the values in $s_1, \ldots, s_k$ or is absent, and $\mathbf{order}(c_1, c_2)$ checks that no occurence of c_1 comes after an occurence of c_2 in the contents sequence. The auxiliary functions atts and word and the predicates exists and before are formally defined by:

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

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

$$exists(w_1 \cdots w_k, c) \equiv \exists 1 \leq i \leq k : w_i = c$$

$$before(w_1 \cdots w_k, c_1, c_2) \equiv \forall 1 \leq i, j \leq k : w_i = c_1 \wedge w_j = c_2 \Rightarrow i \leq j$$

Two common abbreviations are $\mathbf{unique}(c) \equiv \mathbf{order}(c, c)$ ("c occurs at most once") and $\mathbf{exclude}(c_1, c_2) \equiv \neg(\mathbf{content}(c_1) \wedge \mathbf{content}(c_2))$ ("c_1 and c_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 XML languages described by DTDs or by more advanced schema languages typically 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:

$$\begin{aligned} \mathcal{A}(\texttt{html}) &= \{\texttt{xmlns}, \texttt{lang}, \texttt{xml:lang}, \texttt{dir}\} \\ \mathcal{E}(\texttt{html}) &= \{\texttt{head}, \texttt{body}\} \\ \mathcal{F}(\texttt{html}) &= \mathbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \mathbf{content}(\texttt{head}) \wedge \\ &\quad \mathbf{content}(\texttt{body}) \wedge \mathbf{unique}(\texttt{head}) \wedge \\ &\quad \mathbf{unique}(\texttt{body}) \wedge \mathbf{order}(\texttt{head}, \texttt{body}) \end{aligned}$$

$$\begin{aligned} \mathcal{A}(\texttt{head}) &= \{\texttt{lang}, \texttt{xml:lang}, \texttt{dir}, \texttt{profile}\} \\ \mathcal{E}(\texttt{head}) &= \{\texttt{script}, \texttt{style}, \texttt{meta}, \texttt{link}, \texttt{object}, \texttt{isindex} \\ &\quad \texttt{title}, \texttt{base}\} \\ \mathcal{F}(\texttt{head}) &= \mathbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \mathbf{content}(\texttt{title}) \wedge \\ &\quad \mathbf{unique}(\texttt{title}) \wedge \mathbf{unique}(\texttt{base}) \end{aligned}$$

$$\begin{aligned} \mathcal{A}(\texttt{input}) &= \{\texttt{id}, \texttt{class}, \texttt{style}, \texttt{title}, \texttt{lang}, \texttt{xml:lang}, \\ &\quad \texttt{dir}, \texttt{onclick}, \texttt{ondblclick}, \texttt{onmousedown}, \\ &\quad \texttt{onmouseup}, \texttt{onmouseover}, \texttt{onmousemove}, \\ &\quad \texttt{onmouseout}, \texttt{onkeypress}, \texttt{onkeydown}, \\ &\quad \texttt{onkeyup}, \texttt{type}, \texttt{name}, \texttt{value}, \texttt{checked}, \\ &\quad \texttt{disabled}, \texttt{readonly}, \texttt{size}, \texttt{maxlength}, \\ &\quad \texttt{src}, \texttt{alt}, \texttt{usemap}, \texttt{tabindex}, \texttt{accesskey}, \\ &\quad \texttt{onfocus}, \texttt{onblur}, \texttt{onselect}, \texttt{onchange}, \\ &\quad \texttt{accept}, \texttt{align}\} \\ \mathcal{E}(\texttt{input}) &= \emptyset \\ \mathcal{F}(\texttt{input}) &= \mathbf{value}(\texttt{dir}, \{\texttt{ltr}, \texttt{rtl}\}) \wedge \\ &\quad \mathbf{value}(\texttt{checked}, \{\texttt{checked}\}) \wedge \\ &\quad \mathbf{value}(\texttt{disabled}, \{\texttt{disabled}\}) \wedge \\ &\quad \mathbf{value}(\texttt{readonly}, \{\texttt{readonly}\}) \wedge \\ &\quad \mathbf{value}(\texttt{align}, \{\texttt{top}, \texttt{middle}, \texttt{bottom}, \\ &\qquad \texttt{left}, \texttt{right}\}) \wedge \\ &\quad \mathbf{value}(\texttt{type}, \{\texttt{text}, \texttt{password}, \texttt{checkbox}, \\ &\qquad \texttt{radio}, \texttt{submit}, \texttt{reset}, \texttt{file}, \\ &\qquad \texttt{hidden}, \texttt{image}, \texttt{button}\}) \wedge \\ &\quad (\mathbf{value}(\texttt{type}, \{\texttt{submit}, \texttt{reset}\}) \vee \mathbf{attr}(\texttt{name})) \end{aligned}$$

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. In the implementation, these fragment removal rules are specified the same way as the element constraints in the abstract DTD for XHTML, so essentially, we have just moved a few of the DTD constraints into a separate file.

7. VALIDATING SUMMARY GRAPHS

For every **`show`** statement, the data-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:

$$\forall r \in R : \mathbf{f}(r) = \rho(\delta)\phi \quad \text{for some } \delta \text{ and } \phi$$

Then for each sub-template $e(\delta)\phi$ of a template 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)
- $n, \delta, \phi \Vdash \mathcal{F}(e)$ (the constraint is satisfied)

The validity relation $\Vdash$ is given by:

$$\frac{n, \delta, \phi \Vdash \psi_1 \quad n, \delta, \phi \Vdash \psi_2}{n, \delta, \phi \Vdash \psi_1 \wedge \psi_2}$$

$$\frac{n, \delta, \phi \Vdash \psi_1}{n, \delta, \phi \Vdash \psi_1 \vee \psi_2} \qquad \frac{n, \delta, \phi \Vdash \psi_2}{n, \delta, \phi \Vdash \psi_1 \vee \psi_2}$$

$$\frac{}{n, \delta, \phi \Vdash true} \qquad \frac{n, \delta, \phi \nVdash \psi}{n, \delta, \phi \Vdash \neg\, \psi}$$

$$\frac{a \in names(\delta)}{n, \delta, \phi \Vdash \mathbf{attr}(a)} \qquad \frac{c \in occurs(n, \phi)}{n, \delta, \phi \Vdash \mathbf{content}(c)}$$

$$\frac{order(n, \phi, c_1, c_2)}{n, \delta, \phi \Vdash \mathbf{order}(c_1, c_2)}$$

$$\frac{a \notin names(\delta)}{n, \delta, \phi \Vdash \mathbf{value}(a, \{s_1, \ldots, s_k\})}$$

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

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

where $occurs$ is the least function satisfying:

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

and $order$ is the most restrictive function satisfying:

$$\begin{array}{rcl} order(n,\epsilon,c_1,c_2) & = & true \\ order(n,\bullet,c_1,c_2) & = & true \\ order(n,g,c_1,c_2) & = & \bigwedge_{(n,g,m)\in E} order(m,\mathbf{f}(m),c_1,c_2) \\ order(n,e(\delta)\phi,c_1,c_2) & = & true \\ order(n,\phi_1\phi_2,c_1,c_2) & = & order(n,\phi_1,c_1,c_2) \wedge \\ & & order(n,\phi_2,c_1,c_2) \wedge \\ & & \neg\,(c_2 \in occurs(n,\phi_1) \wedge \\ & & \quad c_1 \in occurs(n,\phi_2)) \end{array}$$

The definition of the validity relation is straightforward. It duals the definition of the acceptance relation in Section 6, except that we now have to take gaps into account. Only the auxiliary functions, $occurs$ and $order$, are non-trivial. The function $occurs(n,\phi)$ finds the subset of $\mathcal{N}^\bullet$ that can occur as contents of the current element after plugging some gaps according to the summary graph, and $order(n,\phi,c_1,c_2)$ checks that it is not possible to obtain an c_2 before an c_1 in the contents ϕ. These two functions are defined as fixed points because the summary graphs may contain loops. In the implementation we ensure termination by applying memoization to the numerous calls to $occurs$ and $order$.

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

Also note that our notion of abstract DTDs has a useful locality property: All requirements defined by an abstract DTD specify properties of single XML document nodes and their attributes and immidiate contents, so if some requirement is not fulfilled by a given summary graph, it is possible to give a precise error message.

8. EXPERIMENTS

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

Name	Lines	Templates	Size	Shows	Time
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 templates, the size $(|E|,|\alpha|)$ of the largest summary graph, the number of **show** statements, and the analysis time in seconds (on an 800 MHz Pentium III with Linux).

The `chat` benchmark is a simple chat service, `guess` is a number guessing game, `calendar` shows a monthly calendar, `xbiff` is a soccer match reservation system, `webboard` is a bulletin board service, `cdshop` is a demonstration of an online shop, `jaoo` is a conference administration system, `bachelor` is a student management service, `courses` is a course administration system, and `eatcs` is a collection of services used by the EATCS organization. Some of the benchmarks are taken from the `<bigwig>` documentation, others are services currently being used or developed at BRICS.

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:

```
 1 service {
 2   html cover = <html>
 3     <head><title>Welcome</title></head>
 4     <body bgcolo=[color]>
 5       <table><[contents]></table>
 6     </body>
 7   </html>;
 8
 9   html greeting = <html>
10     <td>Hello <[who]>,<br clear=[clear]>
11         welcome to <[what]>.
12     </td>
13   </html>;
14
15   html person = <html>
16     <i>Stranger</i>
17   </html>;
18
19   session welcome() {
20     html h;
21     h = cover<[color="#9966ff",
22                contents=greeting<[who=person],
23                clear="righ"];
24     show h<[what=<html><b>BRICS</b></html>];
25   }
26 }
```

the compiler generates the following messages for the single **show** statement:

```
--- brics.wig:24: HTML validation:
brics.wig:4:
  warning: illegal attribute 'bgcolo' in 'body'
  template: <body bgcolo=[color]><form>...</form></body>

brics.wig:5:
  warning: possible illegal subelement 'td' of 'table'
  template: <table><[contents]></table>
  contents: td
  plugs: contents:{brics.wig:22}

brics.wig:10:
  warning: possible element constraint violation at 'br'
  template: <br clear=[clear]/>
  constraint: value(clear,{left,all,right,clear,none})
  plugs: clear:{brics.wig:23}
```

At each error message, a line number of an XML element is printed together with an abbreviated form of the involved template, the

names of the root elements of each template that can be plugged into the gaps, the constraint being violated, and the line numbers of the involved plug operations. Such reasonably precise error diagnostics is clearly useful for debugging.

9. RELATED WORK

There are other languages for constructing XML documents that also consider validity. The XDuce language [3, 4] is a functional language in which XML templates 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 data-flow analysis, we obtain a high degree of polymorphism that is difficult to express in a traditional type system. The XMλ language [7] compares similarly to our approach.

The initial design of the `<bigwig>` template mechanism was inspired by the Mawl language [6, 1]. The main difference is that Mawl only allows strings to plugged into the gaps. Validating that Mawl programs only generate valid XHTML is therefore as easy as validating static documents, but such a simple document construction mechanism often becomes too restrictive for practical use. We have shown that using a highly flexible mechanism does not require validity guarantees to be sacrificed.

Most Web services are currently written either in Perl using CGI, in embedded scripting languages such as ASP, PHP, or JSP, or as server-integrated modules, for instance with Apache. Common to all these approaches is that there is no inherent type system for HTML or XML documents. In general, documents are constructed by concatenating text strings. These strings contain HTML or XML tags, attributes, etc., but the compiler or interpreter is completely unaware of that. This means that even *well-formedness*, that is, that tags are balanced and nested properly, which is one requirement for validity, becomes difficult to verify. We get that for free during parsing of the individual constant XML fragments and can concentrate on the many other validity requirements given by specific DTDs.

However, a common way of programming services in these languages is to use HTML or XML *constructor functions* to build documents more abstractly as trees instead of strings. This style is not enforced by the language, but if used consistently well-formedness is guaranteed. The difference between this and the `<bigwig>` style is that gaps in `<bigwig>` templates may appear non-locally, as described in Section 1, which gives a higher degree of flexibility. Since the constructor-based style is subsumed under the `<bigwig>` style as also described in Section 1, the summary graph technique could be applied for other languages.

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 [5], in particular context dependency and regular expression constraints on attribute values and character data.

Since our technique is parameterized in the choice of the abstract DTD, it easily generalizes to many other XML languages that can be described by such abstract DTDs. Finally, we could enrich `<bigwig>` with a set of operators for combining and deconstructing XML templates, making it a general XML transformation language. All such ideas readily permit analysis by means of summary graphs. However, a method for translating a DTD into a summary graph will be required.

11. CONCLUSION

We have combined a data-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. The analysis is efficient and does not generate many spurious error messages in practice. Furthermore, it provides precise error diagnostics in case a given program fails to verify.

Since our algorithm is parameterized with an abstract 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 the `<bigwig>` language.

12. REFERENCES

[1] David Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. Mawl: a domain-specific language for form-based services. In *IEEE Transactions on Software Engineering*, June 1999.

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

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

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

[5] 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.

[6] David A. Ladd and J. Christopher Ramming. Programming the web: An application-oriented language for hypermedia services. In *4th Intl. World Wide Web Conference (WWW4)*, 1995.

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

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

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

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