BRICS RS-06-8 Kirkegaard & Møller: Static Analysis for Java Servlets and JSP

BRICS

Basic Research in Computer Science

Static Analysis for Java Servlets and JSP

Christian Kirkegaard
Anders Møller

BRICS Report Series **RS-06-8**

ISSN 0909-0878 **April 2006**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark
Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/06/8/

Static Analysis for Java Servlets and JSP

Christian Kirkegaard and Anders Møller

BRICS*, University of Aarhus, Denmark
{ck,amoeller}@brics.dk

Abstract. We present an approach for statically reasoning about the behavior of Web applications that are developed using Java Servlets and JSP. Specifically, we attack the problems of guaranteeing that all output is well-formed and valid XML and ensuring consistency of XHTML form fields and session state. Our approach builds on a collection of program analysis techniques developed earlier in the JWIG and XACT projects, combined with work on balanced context-free grammars. Together, this provides the necessary foundation concerning reasoning about output streams and application control flow.

1 Introduction

Java Servlets [16] and JSP (JavaServer Pages) [17] constitute a widely used platform for Web application development. Applications that are developed using these or related technologies are typically structured as a collection of program fragments (servlets or JSP pages) that receive user input, produce HTML or XML output, and interact with databases. These fragments are connected via forms and links in the generated pages, using deployment descriptors to declaratively map URLs to program fragments. This way of structuring applications causes many challenges to the programmer. In particular, it is difficult to ensure, at compile time, the following desirable properties:

– all output should be well-formed and valid XML (according to, for example, the schema for XHTML 1.0);
– the forms and fields that are produced by one program fragment that generates an XHTML page should always match what is expected by another program fragment that takes care of receiving the user input; and
– session attributes that one program fragment expects to be present should always have been set previously in the session.

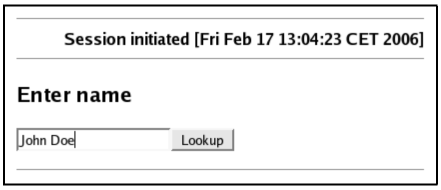
Our aim is to develop a program analysis system that automatically can check these properties for a given Web application.

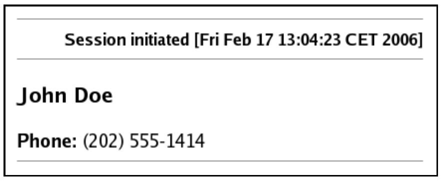
The small example program shown on the following page illustrates some of the many challenges that may arise.

* Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

```
public class Entry extends javax.servlet.http.HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession();
    String url = response.encodeURL(request.getContextPath()+"/show");
    session.setAttribute("timestamp", new Date());
    response.setContentType("application/xhtml+xml");
    PrintWriter out = response.getWriter();
    Wrapper.printHeader(out, "Enter name", session);
    out.print("<form action=\""+url+"\" method=\"POST\">"+
              "<input type=\"text\" name=\"NAME\"/>"+
              "<input type=\"submit\" value=\"lookup\"/>"+
              "</form>");
    Wrapper.printFooter(out);
  }
}

public class Show extends javax.servlet.http.HttpServlet {
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
    Directory directory = new Directory("ldap://ldap.widgets.org");
    String name = misc.encodeXML(request.getParameter("NAME"));
    response.setContentType("application/xhtml+xml");
    PrintWriter out = response.getWriter();
    Wrapper.printHeader(out, name, request.getSession());
    out.print("<b>Phone:</b> "+directory.phone(name));
    Wrapper.printFooter(out);
  }
}

public class Wrapper {
  static void printHeader(PrintWriter pw, String title,
                          HttpSession session) {
    pw.print("<html xmlns=\"http://www.w3.org/1999/xhtml\">"+
             "<head><title>"+title+"</title></head><body>"+
             "<hr noshade size=\"1\"/>"+
             "<div align=\"right\"><small>"+
             "Session initiated ["+session.getAttribute("timestamp")+"]"+
             "</small></div><hr noshade size=\"1\"/>"+
             "<h3>"+title+"</h3>");
  }

  static void printFooter(PrintWriter pw) {
    pw.print("<hr noshade size=\"1\"/></body></html>");
  }
}
```

This program contains two servlets: one named Entry that produces an XHTML page with a form where the user enters a name, and one named Show that receives the user input and produces a reply as another XHTML page based on information from an external database. We assume that the deployment descriptor maps the relative URL enter to the first servlet and show to the second one. Also, misc.encodeXML is a method that escapes special XML characters (for example, converting < to <). At runtime, the pages may look as follows:





In order for the program to work as intended, the programmer must consider many aspects, even for such a tiny program, as the following questions indicate:

- do all open begin tags produced by printHeader match the end tags produced by printFooter?
- does getAttribute("timestamp") always return strings that are legal as XML character data? (for example, ‘<’ should not appear here)
- does the form action URL that is produced by Enter in fact point to the Show servlet? (this depends on the value of the action and method attributes and the deployment descriptor mapping)
- is the parameter NAME always present when the Show servlet is executed? (checking this requires knowledge of the presence of form fields in the XHTML pages that lead to this servlet)
- is the attribute timestamp always present in the session state when the Show servlet is executed? (if not, a null reference would appear)

To answer such questions statically, one must have a clear picture of which string fragments are being printed to the output stream and how the servlets are connected in the application. Presently, programmers resort to informal reasoning and incomplete testing in order to obtain confidence of the correctness of the program. A more satisfactory situation would of course be to have *static guarantees* provided by a *fully automatic* analysis tool.

As the desirable properties listed above are clearly undecidable, the analysis we present is necessarily approximative. We design our analysis to be conservative in the sense that it may produce spurious warnings, but a program that passes the analysis is guaranteed to satisfy the properties. Naturally, we aim for an analysis that has sufficient precision and performance to be practically useful.

Application servers handle JSP through a simple translation to servlets [17]. This means that by focusing our analysis efforts on servlets, we become able to handle JSP, and applications that combine servlets and JSP, essentially for free.

Contributions Our contributions are the following:

- We show how to obtain a context-free grammar that conservatively approximates the possible output of servlet/JSP applications using a variant of the Java string analysis [8].
- On top of the string analysis, we apply theory of balanced grammars by Knuth [12] and grammar approximations by Mohri and Nederhof [14] to check that the output is always well-formed XML.
- On top of the well-formedness checking, we show how a balanced context-free grammar can be converted into a summary graph, which is subsequently validated relative to an XML schema using an existing algorithm [10].
- By analyzing the form and link elements that appear in the summary graph together with the deployment descriptor of the application, we explain how to obtain an inter-servlet control flow graph of the application.
- Based on the knowledge of the control flow, we give examples of derived analyses for checking that form fields and session state are used consistently.

Together, the above components form a coherent analysis system for reasoning about the behavior of Web application that are built using Java Servlets and JSP. The system has a *front-end* that converts from Java code to context-free grammars and a *back-end* that converts context-free grammars to summary graphs and checks well-formedness, validity, and other correctness properties. Our approach can be viewed as combining and extending techniques from the JWIG and XACT projects [7, 11, 10] and applying them to a mainstream Web application development framework.

Perhaps surprisingly, the analysis of well-formedness and validity can be made both sound and complete relative to the grammar being produced in the front-end. (The completeness, however, relies on an assumption that certain well-defined contrived situations do not occur in the program being analyzed).

The goal of the present paper is to outline our analysis system, with particular focus on the construction of context-free grammars and the translation from context-free grammars to summary graphs. The limited space prevents us from describing the details of each component, so we instead base our presentation on a running example. The system is at the time of writing not yet fully implemented; we return to this issue in Section 6.

Although we here focus on Java-based Web applications, we are not relying on language features that are specific to Java. In particular, the approach we present could also be applied to the .NET or PHP platforms where Web applications are typically also built from loosely connected program fragments that each produce XHTML output and receive form input.

Related Work We are not aware of previous attempts to statically analyze the aspects mentioned above for Java Servlets and JSP applications. The most closely related work is that of Minamide [13] who combines string analysis with HTML validation for PHP. In [13], a variant of the technique from [8] is used to produce a context-free grammar from a PHP program. HTML validation is performed either by extracting and checking sample documents or by considering only documents with bounded depth, which results in neither sound nor complete analysis results.

There are other related interesting connections between XML data and context-free grammars, in particular, the work by Berstel and Boasson [3] and Brüggemann-Klein and Wood [5]. The paper [3] uses Knuth's results to check some aspects of XML well-formedness for a given context-free grammar, but it does not take the full step to validity. The paper [5] only considers grammars that correspond to well-formed XML documents, whereas our scenario involves arbitrary context-free grammars that need to be checked for well-formedness and validity.

Inter-servlet control flow analysis is closely related to workflow and business protocols for Web services. Much work is going on in designing workflow languages and Web service composition languages to be used for modeling and analyzing properties during the design phase of Web application development (examples are WS-BPEL [2] and YAWL [19]). Our work complements this in the sense that the analysis we present is able to reverse engineer workflows from the source code of existing Web applications (although that is not the focus of the present paper). This is related to process mining [9] but using source code instead of system logs, and thereby obtaining conservative results.

As mentioned, our technique builds on our earlier work on JWIG and XACT. JWIG [7] is a Java-based framework for Web application development where session control-flow is explicit and XHTML pages are built in a structured manner that permits static analysis of validity and form field consistency. XACT [11] is a related language for expressing XML transformations. The notion of *summary graphs*, which is essential to our analysis system, comes from these projects – a summary graph is a representation of a potentially infinite set of XML structures that may appear in a running JWIG or XACT program. The paper [10] describes an algorithm for validating a summary graph relative to a schema written in XML Schema.

Overview We first, in Section 2, describe how to analyze the output stream and produce a context-free grammar that approximates the possible output of a given Web application. Section 3 explains the well-formedness check and the construction of a balanced grammar. In Section 4 we then show how to convert the balanced grammar into a summary graph and check validity relative to an XML schema.

Section 5 describes the construction of the inter-servlet control flow graph, based on the summary graph and the deployment descriptor. We also sketch how to use the summary graph and the control-flow information to check consistency of the use of form fields and session state. Finally, in Section 6 we discuss challenges and considerations for implementing the entire analysis system and expectations for its performance and precision.

We defer the technical details to appendices: in Appendix A we recapitulate Knuth's algorithm for checking balancing of the language of a CFG and introduce some notation used in the following appendix; Appendix B presents our extension of Knuth's algorithm for constructing a balanced grammar for a balanced language; and Appendix C explains the precision of our analysis.

2 Analyzing the Output Stream

A servlet sends data to its clients by writing string values to a special output stream, which is allocated by the Web server for each request. Our analysis must trace these output streams and keep track of all the string values written to them. Given a Web application, the analysis produces for each servlet entry point a context-free grammar whose language is guaranteed to contain all data that can possibly be written to the corresponding output stream at runtime.

To keep track of string values, we first run the Java string analysis as described in [8] with the parameters of each `write`, `print`, and `append` invocation on output streams as hotspots. For each invocation, the result is a regular language containing all the possible string values that may occur at those program points.

The subsequent analysis of output streams is a variant of that of `StringBuffer`s in the string analysis [8]. In both cases the basic problem is to keep track of the possible *sequences* of side-effecting operations that may be performed on certain objects. However, there are only append-like operations on output streams, and, since append is an associative operation, this makes the handling of interprocedural data-flow somewhat simpler in our case.

For each method in the Web application, we produce a flow graph where edges represent control flow and nodes have the following kinds:

- append: an append operation corresponding to a `write`, `print`, or `append` operation on an output stream, where the argument is given by a regular language of string values as produced by the preliminary string analysis;
- invoke: a method invocation carrying information about its possible targets;
- nop: a join point (for example, for a `while` statement or a method exit).

Constructing such a flow graph, even for a single servlet, is not trivial. The Java language imposes many challenges, such as, virtual method dispatching, exceptions, and data transfer via instance fields and arrays. Additionally, the Java standard library allows stream objects to be nested in different ways (using `BufferedStream`, `PrintWriter`, etc.). Fortunately, most of the hard work can be done using the Soot framework [18], much like in our earlier applications of Soot [8, 7, 11]. We also need to keep track of the relevant output streams, but that can be done easily with Soot's alias analysis capabilities. The request dispatching mechanism in the Servlet API can be handled similarly.

As an example, we obtain the flow graph shown in Figure 1 for the example program from Section 1.

We use the following terminology about context-free grammars. A *context-free grammar (CFG)* G is a quadruple (V, Σ, S, P) where V is the nonterminal alphabet, Σ is the terminal alphabet (in our grammars, Σ is the Unicode alphabet), $V \cap \Sigma = \emptyset$, $S \subseteq V$ is a set of start nonterminals, and P is a finite set of productions of the form $A \rightarrow \theta$ where $A \in V$ and $\theta \in U^*$, using U to denote the combined alphabet $V \cup \Sigma$. We write $\alpha A \omega \Rightarrow \alpha\theta\omega$ when $A \rightarrow \theta$ is in P and $\alpha, \omega \in U^*$, and $\Rightarrow^+$ and $\Rightarrow^*$ are respectively the transitive closure and the reflexive transitive closure of $\Rightarrow$. The *language* of G is defined as

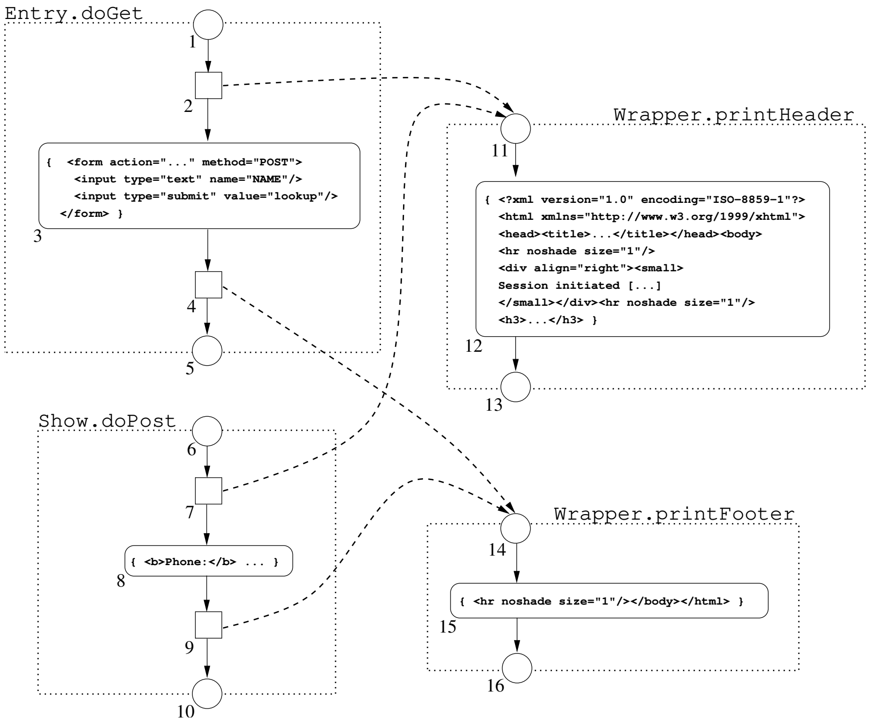


Fig. 1. Flow graph for the example program. (We here depict append nodes as rounded boxes, invoke nodes are squares, nop nodes are circles, and dotted edges represent method boundaries.)

$\mathcal{L}(G) = \{x \in \Sigma^* \mid \exists s \in S : s \Rightarrow^+ x\}$. The language of a nonterminal A is $\mathcal{L}_G(A) = \{x \in \Sigma^* \mid A \Rightarrow^+ x\}$. We sometimes omit the subscript G in $\mathcal{L}_G$ when it can be inferred from the context.

Given a flow graph, we derive a CFG $G = (V, \Sigma, S, P)$ where each flow graph node n is associated with a nonterminal $A_n \in V$ such that $\mathcal{L}(A_n)$ is the set of strings that can be output starting from n:

- for an append node n with an edge to m and whose label is L, we add a production $A_n \rightarrow R_L A_m$ where R_L is the start nonterminal for a linear sub-grammar for L;
- for an invoke node n with a successor m and a possible target method represented by a node t, we add $A_n \rightarrow A_t A_m$; and
- for a nop node n with a successor m we add $A_n \rightarrow A_m$, and for one with no successors we add $A_n \rightarrow \epsilon$.

The start nonterminals are those that correspond to the servlet entry points.

Example The grammar for the example flow graph has $V = \{A_1, \ldots, A_{16}, R_3, R_8, R_{12}, R_{15}\}$, $S = \{A_1, A_6\}$, and P contains the following productions:

$$
\begin{array}{llll}
A_1 \to A_2 & A_6 \to A_7 & A_{11} \to A_{12} & A_{14} \to A_{15} \\
A_2 \to A_{11}A_3 & A_7 \to A_{11}A_8 & A_{12} \to R_{12}A_{13} & A_{15} \to R_{15}A_{16} \\
A_3 \to R_3A_4 & A_8 \to R_8A_9 & A_{13} \to \epsilon & A_{16} \to \epsilon \\
A_4 \to A_{14}A_5 & A_9 \to A_{14}A_{10} & & \\
A_5 \to \epsilon & A_{10} \to \epsilon & &
\end{array}
$$

$$
\begin{array}{ll}
R_3 & \to \llbracket \texttt{<form action="}\ldots\texttt{" method="POST">}\ldots\texttt{</form>} \rrbracket \\
R_8 & \to \llbracket \texttt{Phone:}\ldots \rrbracket \\
R_{12} & \to \llbracket \texttt{<html xmlns="http://www.w3.org/1999/xhtml">} \\
 & \quad \texttt{<head><title>}\ldots\texttt{</title></head><body>}\ldots \rrbracket \\
R_{15} & \to \llbracket \texttt{<hr noshade size="1"/></body></html>} \rrbracket
\end{array}
$$

($\llbracket\cdot\rrbracket$ denotes a linear grammar for the given regular language.)

3 Checking Well-formedness using Balanced Grammars

The goal of this phase is to check for a given CFG G whether all strings in $\mathcal{L}(G)$ are well-formed XML documents. We simplify the presentation by ignoring XML comments, processing instructions, entity references, and the compact form of empty elements (e.g. that
</br> may be written as
), and we assume that all attributes are written on the form *name*="*value*".

This phase proceeds in a number of steps that consider different aspects of well-formedness. First, however, we need to be able to easily identify occurrences of the two characters </ in the language of the grammar. We achieve this by a simple preliminary grammar transformation that – without changing the language of the grammar – eliminates productions on the form $A \to \alpha\texttt{<}\omega$ where $\omega \in VU^* \wedge \texttt{/} \in \mathit{FIRST}(\omega)$ or $\omega \Rightarrow^* \epsilon \wedge \texttt{/} \in \mathit{FOLLOW}(A)$. (See, for instance, [1] for a definition of *FIRST* and *FOLLOW*.) From here on, </ is treated as a single alphabet symbol.

To be able to identify the XML structure in the grammar, we define six special forms of grammar productions:

$$
\begin{array}{ll}
C \to \texttt{<}\ T\ A\ \texttt{>}\ C\ \texttt{</}\ T\ \texttt{>} & \text{(element form)} \\
C \to X & \text{(text form)} \\
C \to C\ C & \text{(content sequence form)} \\
A \to W\ T\ \texttt{=}\ \texttt{"}\ V\ \texttt{"} & \text{(attribute form)} \\
A \to A\ A & \text{(attribute sequence form)} \\
A \to \epsilon & \text{(empty form)}
\end{array}
$$

Here, C represents nonterminals, called *content nonterminals*, whose productions are all on element form, text form, or content sequence form, and A represents nonterminals, called *attribute nonterminals*, whose productions are all on attribute form, attribute sequence form, or empty form. T represents nonterminals whose languages contain no whitespace and no $\texttt{<}$, $\texttt{>}$, or $\texttt{=}$ symbols, W represents nonterminals whose languages consist of nonempty whitespace, X

represents nonterminals whose languages do not contain `<`, and V means the same as X except that it also excludes `"`. We say that a CFG is on *tag-form* if every start nonterminal $s \in S$ is a content nonterminal. Our aim is to convert G into an equivalent grammar on tag-form and check the various well-formedness requirements on the way.

3.1 Step 1: Obtaining a Balanced Grammar

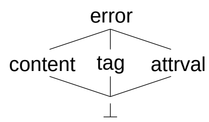
We now view `<` (which marks the beginning of a start tag) as a left parenthesis and `</` (which marks the beginning of an end tag) as a right parenthesis. A necessary condition for $\mathcal{L}(G)$ to be well-formed is that the language in this view is balanced. (A language L is *balanced* if the parentheses balance in every string $x \in L$.) To check this property, we simply apply Knuth's algorithm [12] as described in detail in Appendix A. If the grammar passes this check, Knuth moreover gives us an equivalent *completely qualified* grammar G', as also explained in Appendix A.

As the next step towards tag-form, we will now convert G' into a balanced grammar. (A CFG is *balanced* if every nonterminal is balanced in the sense that the parentheses balance in all derivable strings; for a formal definition see [12] or Appendix B.) Balanced grammars have the useful property that in every production that contains a left parenthesis (`<` in our case), the matching right parenthesis (`</`) appears in the same production. Again we resort to Knuth: in [12], Knuth shows how a CFG that has a balanced language can be converted to a balanced grammar – however, under the assumption that the language has *bounded associates*. Our grammars generally do not have this property (one can easily write a servlet that results in any desirable grammar), so we need to modify Knuth's algorithm to accommodate for a more general setting. Although $\mathcal{L}(G')$ is balanced, there may in fact not exist a balanced grammar G'' with $\mathcal{L}(G') = \mathcal{L}(G'')$, as observed in [12]. Hence we resort to approximation (using a local variant of [14]): the grammar G'' that we produce has the property that it is balanced and $\mathcal{L}(G') \subseteq \mathcal{L}(G'')$. Surprisingly, the loss of precision incurred by this approximation is limited to the degree that it does not affect precision of our well-formedness and validity analyses. A detailed explanation of this rather technical algorithm is given in Appendix B, and proofs of soundness and relative completeness are presented in Appendix C.

3.2 Step 2: Transforming to Tag-form

The symbols in {`<`, `>`, `"`} are essential for our further transformation to tag-form since they function as context delimiters in XML documents in the sense that they delimit the *tag*, *element content*, and *attribute value* contexts, respectively. Given a balanced grammar $G = (V, \Sigma, S, P)$ we will in the following classify nonterminals and symbols occurring on right-hand sides of productions in P according to their possible contexts. If such classification can be uniquely determined, we will use the contexts to extract a grammar on tag-form for $\mathcal{L}(G)$, otherwise we have evidence that some strings in $\mathcal{L}(G)$ are not well-formed.

Let $\mathcal{C}$ be a lattice with values $\bot, \mathsf{tag}$, $\mathsf{content}$, $\mathsf{attrval}$, and error ordered by



and define a function $\delta : \mathcal{C} \times \Sigma \to \mathcal{C}$ by

$$\delta(c,\sigma) = \begin{cases} c & \text{if } \sigma \notin \{\texttt{<}, \texttt{>}, \texttt{"}\} \text{ or } c = \bot \\ \mathsf{tag} & \text{if } (\sigma = \texttt{<} \text{ and } c = \mathsf{content}) \text{ or } (\sigma = \texttt{"} \text{ and } c = \mathsf{attrval}) \\ \mathsf{attrval} & \text{if } (\sigma = \texttt{"} \text{ and } c = \mathsf{tag}) \text{ or } (\sigma = \texttt{>} \text{ and } c = \mathsf{attrval}) \\ \mathsf{content} & \text{if } \sigma = \texttt{>} \text{ and } (c = \mathsf{tag} \text{ or } c = \mathsf{content}) \\ \mathsf{error} & \text{otherwise} \end{cases}$$

Intuitively, δ determines transitions on $\mathcal{C}$ according to the context delimiters $\{\texttt{<}, \texttt{>}, \texttt{"}\}$ and is the identity function on all other symbols.

We may now define a constraint system on the grammar G expressed as a function $\Delta : \mathcal{C} \times U^* \to \mathcal{C}$ defined by the following rules:

$$\Delta(\mathsf{content}, s) \sqsupseteq \mathsf{content} \quad \text{for all } s \in S$$

$$\Delta_G(c, A) \sqsupseteq \Delta_G(c, \theta) \quad \text{for all } A \to \theta \in P$$

$$\Delta(c,x) \sqsupseteq \begin{cases} c & \text{when } x = \epsilon \\ \Delta(\delta(c,\sigma), y) & \text{when } x = \sigma y \text{ where } \sigma \in \Sigma, y \in U^* \\ \Delta(\Delta(c,\theta), y) & \text{when } x = Ay \text{ where } A \in V, y \in U^* \text{ and } A \to \theta \in P \end{cases}$$

The constraint system will always have a unique least solution Δ_G, which can be found using a standard fixed-point algorithm. (This is the case because a finite subset of U^* containing all nonterminals and all prefixes of right-hand sides of productions in P is enough to fulfill the constraints.) Furthermore, if $\Delta_G(\mathsf{content}, s) = \mathsf{error}$ for some $s \in S$ then $\mathcal{L}(G)$ contains a non-well-formed string. In that case, we can issue a precise warning message by producing a derivation starting from s and using productions $A \to \theta$ with $\Delta_G(c, \theta) = \mathsf{error}$.

Assume now that $\Delta_G(\mathsf{content}, s) \neq \mathsf{error}$ for all $s \in S$. The balanced grammar G can then be converted as follows into an equivalent grammar on tag-form.

First, we will ensure that nonterminals occur in unique contexts in all derivations. For every $A \in V$ and $c \in \{\mathsf{content}, \mathsf{tag}, \mathsf{attrval}\}$ where $\Delta_G(c, A) \neq \bot$, create an annotated nonterminal A_c with the same productions as A. Then make A_c a start nonterminal if $A \in S$ and replace every production $B_{c_1} \to \alpha A \omega$ where $\Delta_G(c_1, \alpha) = c_2$ with a production $B_{c_1} \to \alpha A_{c_2} \omega$. All unannotated nonterminals and productions are now unreachable and can be removed.

Now that the grammar is balanced with respect to `<` and `</` and each nonterminal is used in only one context in any derivation, it is straightforward to bring the grammar on tag-form (except for the attribute nonterminals) by repeatedly applying Transformation 1 and Transformation 2 from [12] to eliminate all nonterminals $A \in V$ where $\Delta_G(c, A) \neq c$. We can handle attribute nonterminals similarly by considering a few more context delimiters (whitespace and `=`). Due to the limited space, we omit the tedious details.

Example The extracted CFG for the example program in Section 2 has a balanced language and our transformation results in the following grammar on tag-form with C_1 being the only start nonterminal:

$C_1 \rightarrow \texttt{< html } A_1 \texttt{ > } C_2\ C_4 \texttt{ </ html >}$
$C_2 \rightarrow \texttt{< head > } C_3 \texttt{ </ head >}$
$C_3 \rightarrow \texttt{< title > } X_1 \texttt{ </ title >}$
$C_4 \rightarrow \texttt{< body > } C_{10}\ C_{11}\ C_{10}\ C_8\ C_9\ C_{10} \texttt{ </ body >}$
$C_5 \rightarrow \texttt{< form } A_5\ A_6 \texttt{ > } C_6\ C_7 \texttt{ </ form >}$
$C_6 \rightarrow \texttt{< input } A_7\ A_8 \texttt{ ></ input >}$
$C_7 \rightarrow \texttt{< input } A_9\ A_{10} \texttt{ ></ input >}$
$C_8 \rightarrow \texttt{< h3 > } X_1 \texttt{ </ h3 >}$
$C_9 \rightarrow C_5 \mid C_{13}$
$C_{10} \rightarrow \texttt{< hr } A_2\ A_3 \texttt{ ></ hr >}$
$C_{11} \rightarrow \texttt{< div } A_4 \texttt{ > } C_{12} \texttt{ </ div >}$
$C_{12} \rightarrow \texttt{< small > } X_2 \texttt{ </ small >}$
$C_{13} \rightarrow \texttt{< b >Phone:</ b >}$

$A_1 \rightarrow \texttt{␣xmlns="http://www.w3.org/1999/xhtml"}$
$A_2 \rightarrow \texttt{␣noshade=""}$
$A_3 \rightarrow \texttt{␣size="1"}$
$A_4 \rightarrow \texttt{␣align="right"}$
$A_5 \rightarrow \texttt{␣action=" } V_1 \texttt{ "}$
$A_6 \rightarrow \texttt{␣method="POST"}$
$A_7 \rightarrow \texttt{␣type="text"}$
$A_8 \rightarrow \texttt{␣name="NAME"}$
$A_9 \rightarrow \texttt{␣type="submit"}$
$A_{10} \rightarrow \texttt{␣value="lookup"}$

$X_1 \rightarrow \texttt{Enter␣name} \mid \mathcal{L}_{\mathrm{CDATA}}$
$X_2 \rightarrow \texttt{Session␣initiated␣[} \mathcal{L}_{\mathrm{DATE}} \texttt{]}$
$V_1 \rightarrow \texttt{contextpath/show}$

(To simplify the presentation we have inlined constant string values and chains of productions on sequence form.) $\mathcal{L}_{\mathrm{CDATA}}$ is the set of all strings that can be returned from `misc.encodeXML`, $\mathcal{L}_{\mathrm{DATE}}$ are the legal date string values, and `contextpath` denotes the application context path as obtained by `getContextPath`. These regular languages are obtained by the preliminary string analysis.

3.3 Step 3: Checking Well-formedness

The previous steps have checked a number of necessary conditions for well-formedness. Now that we have the grammar on tag-form, we can easily check the remaining properties:

– All start productions must be on element form. (In other words, there is always exactly one root element.)
– For every production $C_1 \rightarrow \texttt{<}\ T_1\ A\ \texttt{>}\ C_2\ \texttt{</}\ T_2\ \texttt{>}$ on element form, both $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ must be singleton languages. (Otherwise, one could derive a string where a start tag does not match its end tag.)
– For every production $C_1 \rightarrow \texttt{<}\ T_1\ A\ \texttt{>}\ C_2\ \texttt{</}\ T_2\ \texttt{>}$ on element form, the attributes corresponding to A must have disjoint names. More precisely, whenever $A \Rightarrow^{+} \alpha A_1 \phi A_2 \omega$ where $\alpha, \phi, \omega \in U^*$ and $A_i \rightarrow W_i\ T_i\ \texttt{=}\ \texttt{"}\ V_i\ \texttt{"}$ for $i = 1, 2$, we check that $\mathcal{L}(T_1) \cap \mathcal{L}(T_2) = \emptyset$. If the sub-grammars of T_1 and T_2 are linear, this check is straightforward; otherwise, since the property is generally undecidable we sacrifice completeness and issue a warning.

The only way sub-grammars that correspond to attribute names are not linear is that the program being analyzed uses a recursive method to build individual attribute names in a contrived way where a part of a name is written to the output stream *before* the recursive call and another part is written *after* the call.

(We give an example in Appendix C). With the exception of this pathological case, the checks described above are passed if and only if $\mathcal{L}(G)$ contains only well-formed XML documents. (Our running example passes the well-formedness check.)

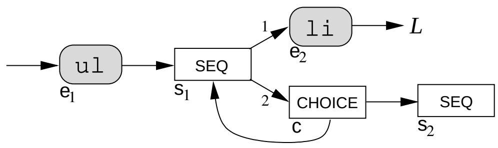
4 Checking Validity using Summary Graphs

A *summary graph* is a finite structure that represents a potentially infinite set of XML trees, as defined in [11, 10]. We here give a brief description of a variant of this formalism, tailored to our present setting.

A summary graph contains finite sets of nodes of various kinds: element nodes ($\mathcal{N}_\mathcal{E}$), attribute nodes ($\mathcal{N}_\mathcal{A}$), text nodes ($\mathcal{N}_\mathcal{T}$), sequence nodes ($\mathcal{N}_\mathcal{S}$), and choice nodes ($\mathcal{N}_\mathcal{C}$). (The definition of summary graphs used in earlier papers also involves gap nodes, which we do not need here.) Let $\mathcal{N} = \mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{T} \cup \mathcal{N}_\mathcal{S} \cup \mathcal{N}_\mathcal{C}$. The graph has a set of root nodes $\mathcal{R} \subseteq \mathcal{N}$. The map *contents* : $\mathcal{N}_\mathcal{E} \cup \mathcal{N}_\mathcal{A} \to \mathcal{N}$ connects element nodes and attribute nodes with descriptions of their contents. For sequence nodes it returns sequences of nodes, *contents* : $\mathcal{N}_\mathcal{S} \to \mathcal{N}^*$, and for choice nodes it returns sets of nodes, *contents* : $\mathcal{N}_\mathcal{C} \to 2^{\mathcal{N}}$. The map *val* : $\mathcal{N}_\mathcal{T} \cup \mathcal{N}_\mathcal{A} \cup \mathcal{N}_\mathcal{E} \to REG$, where REG are all regular string languages over the Unicode alphabet, assigns a set of strings to each text node, element node, and attribute node, in the latter two cases representing their possible names.

A summary graph may be viewed as a generalized XML tree that permits choices, loops, and regular sets of possible attribute/element names and text values. The *language* $\mathcal{L}(SG)$ of a summary graph SG is intuitively the set of XML trees that can be obtained by unfolding it, starting from a root node.

As an example, consider the set of all `ul` lists with one or more `li` items that each contain a string from the regular language L. It can be described by a summary graph with six nodes $\mathcal{N} = \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{s}_1, \mathtt{s}_2, \mathtt{c}, \mathtt{t}\}$, roots $\mathcal{R} = \{\mathtt{e}_1\}$, and maps *contents* $= \{\mathtt{e}_1 \mapsto \mathtt{s}_1,\ \mathtt{e}_2 \mapsto \mathtt{t},\ \mathtt{s}_1 \mapsto \mathtt{e}_2\,\mathtt{c},\ \mathtt{s}_2 \mapsto \epsilon,\ \mathtt{c} \mapsto \{\mathtt{s}_1, \mathtt{s}_2\}\}$ and *val* $= \{\mathtt{e}_1 \mapsto \{\mathtt{ul}\},\ \mathtt{e}_2 \mapsto \{\mathtt{li}\},\ \mathtt{t} \mapsto L\}$. This is illustrated as follows:



The rounded boxes represent the element nodes $\mathtt{e}_1$ and $\mathtt{e}_2$, the SEQ boxes represent the sequence nodes $\mathtt{s}_1$ and $\mathtt{s}_2$ (edges out of $\mathtt{s}_1$ are ordered according to their indices), and the CHOICE box represents the choice node $\mathtt{c}$. The text node $\mathtt{t}$ is represented by its associated language L.

From the Xact project, we have an algorithm that can check for a given summary graph SG and a schema S, written in either DTD or XML Schema, whether or not every XML tree in $\mathcal{L}(SG)$ is valid according to S. (See [10] for a description of the algorithm.) Hence, our only remaining task in order to be able to validate the output of the servlets is to convert the balanced grammar on tag-form that we produced and checked for well-formedness in Section 3 into a summary graph. Fortunately, this is straightforward to accomplish, as explained in the following.

Starting from the start nonterminals S and their productions, each production $p \in P$ is converted to a summary graph node n_p according to its form. Also, each nonterminal A is converted to a choice node n_A with $contents(n_A) = \{n_p \mid p \text{ is a production of } A\}$:

element form For a production $p = C_1 \rightarrow \texttt{<}\ T_1\ A\ \texttt{>}\ C_2\ \texttt{</}\ T_2\ \texttt{>}$, n_p becomes an element node. We know from the well-formedness check that $\mathcal{L}(T_1) = \mathcal{L}(T_2)$ is some singleton language $\{s\}$, so we set $name(n_p) = \{s\}$. To capture the attributes and contents, a sequence node n_q is also added, and we set $contents(n_p) = n_q$ and $contents(n_q) = n_A\, n_{C_2}$.

text form For a production $p = C \rightarrow X$, the sub-grammar starting from X is converted to an equivalent sub-graph rooted by n_p, using only sequence nodes, choice nodes, and text nodes. We omit the details.

attribute form For a production $p = A \rightarrow W\ T\ \texttt{=}\ \texttt{"}\ V\ \texttt{"}$, n_p becomes an attribute node. As in the previous case, the sub-grammar rooted by V is converted to an equivalent sub-graph rooted by a node n_V, and we let $contents(n_p) = n_V$. From the well-formedness check, we know that the sub-grammar of T is linear, so its language is regular and we set $name(n_p)$ accordingly.

content or attribute sequence form For a production $p = C \rightarrow C_1\ C_2$, n_p becomes a sequence node with $contents(n_p) = n_{C_1}\, n_{C_2}$. Productions on attribute sequence form are converted similarly.

empty form For a production $p = A \rightarrow \epsilon$, n_p becomes a sequence node with $contents(n_p) = \epsilon$.

The root nodes $\mathcal{R}$ are the nodes that correspond to the start nonterminals.

For the example program from Section 1, we obtain the summary graph shown in Figure 2. (Note that since the program has no recursive methods, there are no loops in the graph.) Running the XACT validator on this summary graph and the schema for XHTML gives the result “Valid!”, meaning that the program is guaranteed to output only valid XHTML documents.

5 Analyzing Inter-Servlet Control Flow

Servlet/JSP applications are typically structured as collections of dynamic pages that are connected via a deployment descriptor, `web.xml`, together with links (`<a href="...">`) and forms (`<form action="...">`) appearing in generated XHTML documents. Since links and forms are intertwined with general page layout and various kinds of data, it is often a challenging task to recognize and apprehend the complete control flow of applications consisting of more than a few servlets or JSP pages. We will now briefly describe how to further benefit from the summary graphs to obtain an *inter-servlet* control flow graph for an application.

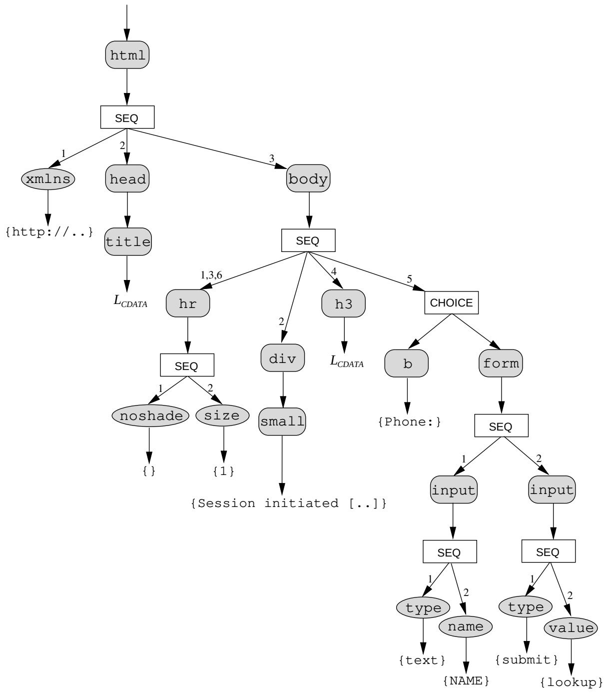


Fig. 2. Summary graph for the example program. (We depict element nodes as rounded boxes, attribute nodes as ellipses, and sequence and choice nodes as SEQ and CHOICE boxes, respectively. Edges out of sequence nodes are ordered according to the indices.)

The goal is to produce a graph with nodes corresponding to the `doGet` and `doPost` methods of each servlet class and edges corresponding to the possible control flow via links or forms in the generated documents. The challenge in producing such a graph is associating a set of possible servlet classes to the links and forms appearing in generated documents by using the URL mappings of the deployment descriptor.

Given a summary graph corresponding to the output of a servlet method we recognize the links and forms by searching (that is, unfolding according to the *contents* map, starting from the roots) for element nodes named `a` or `form`, and further, searching for their attribute nodes with names `href` and `action`, respectively. From each of the attribute values, we can extract a regular language of all possible target URLs and compare with the mappings described by the deployment descriptor to get the corresponding set of servlet classes. This set forms the inter-servlet flow edges out of the method. By applying the process to all servlet methods we obtain an inter-servlet control flow graph, which is guaranteed to be sound because the summary graphs are sound approximations of the possible XHTML output.

The inter-servlet control flow graph for our running example is like the one in Figure 1, however extended with an inter-servlet flow edge from the exit node n_5 of the `Entry.doGet` method to the entry node n_6 of the `Show.doPost` method.

The inter-servlet control flow graph provides a whole-program view of the Web application. This is useful for visualizing the flow to the programmer and for checking reachability properties of the application workflow. It also serves as the foundation for a number of interesting derived analyses. One such analysis is consistency checking of form fields (as explained in detail in the JWIG paper [7]), which guarantees that all request parameters expected by a servlet exist as form fields in the XHTML output of every preceeding servlet in the flow graph. A related analysis is consistency checking of session state, which can guarantee that every use of a session state variable has been preceeded by a definition. Clearly, such analyses are only feasible if the inter-servlet control flow is known, and, as sketched above, the summary graphs are a key to obtain precise knowledge of this flow.

6 Implementation Considerations and Conclusion

We have presented an approach for analyzing servlet/JSP applications to detect errors in the output being generated (XML well-formedness and validity) and outlined how to obtain and apply knowledge of the control flow in the application. The front-end, which constructs a CFG for the program being analyzed, is sound; the back-end, which constructs a summary graph from the CFG and analyzes well-formedness and validity is both sound and complete relative to the CFG (under the assumption that certain well-defined contrived patterns do not occur in the program).

We have chosen an approach of imposing as few restrictions as possible on the programs being analyzed (see the definition of "contrived" in Appendix C and its use in Section 3.3). An alternative approach, which might of course lead to a simpler analysis, would be to restrict the class of programs that the analysis can handle or sacrifice soundness. The trade-off we have chosen investigates the possibilities in the end of this design spectrum that is most flexible seen from the programmer's point of view.

Only a complete implementation and experiments on real applications can tell whether the precision and performance are sufficient for practical use. However, we have reasons to believe that this is the case. Regarding the front-end, it is our experience from the JWIG, XACT, and string analysis projects [7, 11, 8] that the extraction of flow graphs from Java programs works well in practice – regarding both precision and performance – and the extraction of CFGs from flow graphs is both precise and efficient. Similarly, the analysis of summary graphs in the back-end has also shown to work well in practice. The only remaining question is whether the grammar manipulations can be done efficiently, but our preliminary experiments indicate that this is the case. We are presently implementing the grammar manipulations and connecting the components of the analysis system, which will hopefully give more confidence to the practical feasibility of the approach.

Acknowledgments We thank Aske Simon Christensen for inspiring discussions about various aspects of the program analysis.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. Assaf Arkin et al. Web Services Business Process Execution Language Version 2.0, December 2005. OASIS, Committee Draft.
3. Jean Berstel and Luc Boasson. Formal properties of XML grammars and languages. *Acta Informatica*, 38(9):649–671, 2002. Springer-Verlag.
4. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (third edition), February 2004. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.
5. Anne Brüggemann-Klein and Derick Wood. Balanced context-free grammars, hedge grammars and pushdown caterpillar automata. In *Proc. Extreme Markup Languages*, 2004.
6. Aske Simon Christensen. *Something to do with Java.* PhD thesis, BRICS, Department of Computer Science, University of Aarhus, December 2005.
7. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.
8. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.
9. M. H. Jansen-Vullers, Wil M. P. van der Aalst, and Michael Rosemann. Mining configurable enterprise information systems. *Data & Knowledge Engineering*, 56(3):195–244, 2006.
10. Christian Kirkegaard and Anders Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, 2005. Presented at Programming Language Technologies for XML, PLAN-X '06.
11. Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.
12. Donald E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11:269–289, 1967.
13. Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *Proc. 14th International Conference on World Wide Web, WWW '05*, pages 432–441. ACM, May 2005.
14. Mehryar Mohri and Mark-Jan Nederhof. *Robustness in Language and Speech Technology*, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers, 2001.
15. Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.
16. Sun Microsystems. Java Servlet Specification, Version 2.4, 2003. Available from `http://java.sun.com/products/servlet/`.
17. Sun Microsystems. JavaServer Pages Specification, Version 2.0, 2003. Available from `http://java.sun.com/products/jsp/`.
18. Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference, CASCON '99*. IBM, November 1999.
19. Wil M. P. van der Aalst, Lachlan Aldred, Marlon Dumas, and Arthur H. M. ter Hofstede. Design and implementation of the YAWL system. In *Proc. 16th International Conference on Advanced Information Systems Engineering, CAiSE '04*, volume 3084 of *LNCS*. Springer-Verlag, June 2004.

A Checking Balancing of a Context-Free Language

We here recapitulate Knuth's algorithm for checking balancing of the language of a CFG (Theorem 1 in [12]).

Let $G = (V, \Sigma, S, P)$ be a CFG (as defined in Section 2) and $U = V \cup \Sigma$. Assume that Σ contains two distinguished symbols, `(` and `)`, and let $T = \Sigma \backslash \{\texttt{(}, \texttt{)}\}$. The functions $c, d : \Sigma^* \to \mathbb{N}$ are defined as follows where $a \in \Sigma$ and $x \in \Sigma^*$:

$$c(a) = \begin{cases} 1 & \text{if } a = \texttt{(} \\ 0 & \text{if } a \in T \\ -1 & \text{if } a = \texttt{)} \end{cases} \qquad d(a) = \begin{cases} 0 & \text{if } a = \texttt{(} \\ 0 & \text{if } a \in T \\ 1 & \text{if } a = \texttt{)} \end{cases}$$

$$c(\epsilon) = d(\epsilon) = 0$$

$$c(xa) = c(x) + c(a)$$

$$d(xa) = max(d(x), d(a) - c(x))$$

Intuitively, $c(x)$ is the excess of left parentheses over right parentheses in x, and $d(x)$ is the greatest deficiency of left parentheses from right parentheses in any prefix of x. A string $x \in \Sigma^*$ is *balanced* if $c(x) = d(x) = 0$, and a language $L \subseteq \Sigma^*$ is balanced if every $x \in L$ is a balanced.

We first remove useless nonterminals from G (using the standard technique). For each nonterminal in G we can now find strings $\alpha, \phi, \omega \in \Sigma^*$ such that $s \Rightarrow^+ \alpha A \omega \Rightarrow^+ \alpha\phi\omega$ for some $s \in S$. Then define $c(A) = c(\phi)$ and $d(A) = c(\alpha)$. The functions c and d are now extended to the domain U^* in the same way as they above are extended from single alphabet symbols to strings.

Knuth shows that $\mathcal{L}(G)$ is balanced if and only if the following properties hold:

- $c(s) = 0$ for each $s \in S$;
- $c(A) = c(\theta)$ for each $A \to \theta$ in P; and
- there exists a function $d_1 : U^* \to \mathbb{N}$ where
  - $d_1(a) = d(a)$ for each $a \in \Sigma$;
  - $d_1(s) = 0$ for each $s \in S$;
  - $0 \leq d_1(A) \leq d(A)$ for each $A \in V$; and
  - $d_1(A) \geq d_1(\theta)$ for each $A \to \theta$.

Furthermore, Knuth shows that if $\mathcal{L}(G)$ is balanced then we can construct a grammar G' where $\mathcal{L}(G) = \mathcal{L}(G')$ such that G' is *completely qualified*, that is, d_1 has the property that $d_1(A) = d_1(\theta)$ for each production $A \to \theta$ in P. It follows directly that a grammar is completely qualified if and only if $c(A) = c(x)$ and $d_1(A) = d_1(x)$ whenever $x \in \mathcal{L}(A)$.

B Constructing a Balanced Grammar for a Balanced Language

In this section we recapitulate Knuth's algorithm for constructing a balanced grammar from a CFG with a balanced language (Theorem 3 in [12]) and present an extension for CFGs whose languages do not have bounded associates.

Let $G = (V, \Sigma, S, P)$ be a completely qualified CFG and assume that $\mathcal{L}(G)$ is a balanced language. Also, let $c, d_1 : U^* \to \mathbb{N}$ be functions as defined in Appendix A. G is a *balanced grammar* if for each of its productions $A \to \theta$ in P, the right-hand side θ is a balanced string.

For notational convenience we will allow the notation *, representing *zero-or-more* occurrences. More precisely, for a nonterminal A, we assume the implicit productions $A^* \to AA^* \mid \epsilon$ in P.

To construct balanced grammars we will need to distinguish between matching and free parentheses. The parentheses in $x\texttt{(}y\texttt{)}z \in U^*$ are said to *match* if y is balanced. The left parenthesis in $x\texttt{(}y \in U^*$ is *free* if $d_1(y) = 0$. The right parenthesis in $x\texttt{)}y \in U^*$ is *free* if $c(x) \leq 0$ and $d_1(x) = -c(x)$.

For a string $u = u_1 u_2 \ldots u_n \in U^*$ we form its *parenthesis image* $\mathcal{I}(u)$ by replacing every symbol u_i by a sequence of $d_1(u_i)$ right parentheses followed by $c(u_i) + d_1(u_i)$ left parentheses and then removing all matching parentheses. The result is a string over the alphabet $\{\texttt{(},\texttt{)}\}$ with $d_1(u)$ free right parentheses and $c(u) + d_1(u)$ free left parentheses. The parenthesis image of a nonterminal is defined by $\mathcal{I}(A) = \mathcal{I}(\theta)$ if P contains a production $A \to \theta$ (this is well-defined since G is completely qualified).

Using the parenthesis images of the productions in P we construct a directed graph $\mathcal{D}_G$. The nodes of $\mathcal{D}_G$ are labeled $[A\texttt{)}u]$ or $[A\texttt{(}u]$ where $A \in V$ and $u \in \mathbb{N}$. The label $[A\texttt{)}u]$ represents the u'th free right parenthesis in strings derived from A, and similarly, $[A\texttt{(}u]$ represents the u'th free left parenthesis in strings derived from A. For each production $A \to \theta$ in P we include an edge in $\mathcal{D}_G$ for each parenthesis in $\mathcal{I}(A)$ that does not correspond to an actual parenthesis symbol in θ. More precisely, $\mathcal{D}_G$ has an edge $[A\texttt{)}u] \longrightarrow [B\texttt{)}v]$ if P contains a production $A \to \alpha B \omega$ where the u'th right parenthesis in $\mathcal{I}(\alpha B \omega)$ corresponds to the v'th right parenthesis in $\mathcal{I}(B)$ (and similarly for the left parentheses). The important property of $\mathcal{D}_G$ is that it is empty if and only if G is a balanced grammar.

Knuth gives a simple algorithm for transforming G such that the edges in $\mathcal{D}_G$ are removed without changing $\mathcal{L}(G)$, thereby transforming G into an equivalent balanced grammar. The algorithm progresses by repeatedly eliminating sink nodes from $\mathcal{D}_G$, which clearly only works when $\mathcal{D}_G$ has no cycles. Knuth only considers languages with *bounded associates* and observes that this property implies $\mathcal{D}_G$ being acyclic.

Contrasting Knuth, we permit unrestricted CFGs whose languages in general may have unbounded associates. This means that the resulting $\mathcal{D}_G$ graph may contain cycles and that Knuth's algorithm does not immediately work. In the following we will show how to eliminate cycles in $\mathcal{D}_G$. Intuitively, we replace a set of grammar productions corresponding to a cycle in $\mathcal{D}_G$ by a different set of grammar productions without the cycle, in such a way that the transformed grammar has a slightly larger, but still balanced, language.

Assume that $\mathcal{D}_G$ has a simple cycle

$$[A_0\rangle\, u_0] \longrightarrow \cdots \longrightarrow [A_i\rangle\, u_i] \longrightarrow \cdots \longrightarrow [A_{n-1}\rangle\, u_{n-1}] \longrightarrow [A_0\rangle\, u_0]$$

We can then (by repeatedly inlining the productions of A_{i+1} in the productions of A_i) consider the productions of A_0 on the equivalent form

$$A_0 \to \alpha_1\, A_0\, \omega_1 \mid \ \ldots \ \mid \alpha_m\, A_0\, \omega_m \mid \phi_1 \mid \ \ldots \ \mid \phi_k \tag{1}$$

where each of the first m productions gives rise to a cycle in $\mathcal{D}$ from $[A_0\rangle\, u_0]$ to itself, and the remaining k productions do not have this property. The resulting grammar is denoted G_1.

Next, the productions of A_0 are replaced by

$$A_0 \to X\, X^*\, Z\, Y^*\, Y \mid X\, X^*\, Z \mid Z\, Y^*\, Y \mid Z \tag{2}$$

where X, Y, and Z are new nonterminals defined by

$$\begin{aligned} X &\to \alpha_1 \mid \ldots \mid \alpha_m \\ Y &\to \omega_1 \mid \ldots \mid \omega_m \\ Z &\to \phi_1 \mid \ldots \mid \phi_k \end{aligned}$$

and finally, the productions of X and Y are inlined in the productions of A_0 (that is, X and Y can be thought of as abbreviations rather than nonterminals). The resulting grammar is denoted G_2. (A cycle involving left parentheses is handled similarly.) The step from G_1 to G_2 is an application of Mohri and Nederhof's algorithm for constructing a regular approximation of a CFG [14]; however, we apply it *locally* rather than to a complete CFG (and we use a slightly different notation). Note that the four productions in (2) together have the same language as $A_0 \to X^* Z Y^*$ – the reasons for choosing the form in (2) should be clear from the proof below.

Proposition 1 (Correctness of cycle removal). *Let G be a completely qualified CFG with a balanced language and a $\mathcal{D}_G$ cycle through $[A_0\rangle\, u_0]$. The transformed grammar G_2 given by the above construction has the following properties:*

(a) G_2 is completely qualified;
(b) $\mathcal{L}(G_2)$ is a balanced language; and
(c) $\mathcal{D}_{G_2}$ has strictly fewer simple cycles than $\mathcal{D}_G$.

(The case with left parentheses is symmetric.)

Proof.

(a) The production inlining steps clearly do not affect the set of strings that can be derived from a given production, so G_1 is completely qualified. From the proof of Theorem 3 in [12], we have that $c(X) = c(Y) = 0$, which implies that $c(XX^*) = c(Y^*Y) = 0$, $d_1(XX^*) = d_1(X)$, and $d_1(Y^*Y) = d_1(Y)$. Since G_1 is completely qualified and only the productions of A_0 are changed, it follows by the definitions of c and d_1 that G_2 is also completely qualified.

(b) As noted in part (a), $\mathcal{L}(G) = \mathcal{L}(G_1)$, so $\mathcal{L}(G_1)$ is a balanced language. Since $\mathcal{L}_{G_1}(A_0) \cap \mathcal{L}_{G_2}(A_0) \neq \emptyset$ and both G_1 and G_2 are completely qualified we have that both $c(A_0)$ and $d_1(A_0)$ are unchanged by the step from G_1 to G_2. Therefore, this step does not affect the free parentheses in any string derived from A_0, which implies that $\mathcal{L}(G_1)$ is balanced if and only if $\mathcal{L}(G_2)$ is balanced.

(c) The production inlining steps clearly do not change the number of simple cycles, so no new cycles are introduced in G_1. Consider now the step from G_1 to G_2. The entities X^* and Y^* cannot participate in any $\mathcal{D}_{G_2}$ cycles since $\mathcal{I}(XX^*ZY^*Y)$ cannot have any parentheses that come from X^* or Y^*. Furthermore, any edge in $\mathcal{D}_{G_2}$ on the form $[A_0\rangle u] \longrightarrow [B\rangle v]$ for some u, v and $B \neq Z$ must also be present in $\mathcal{D}_{G_1}$. Since the only productions being changed from G_1 to G_2 are ones with A_0 on the left-hand side, this implies that no new cycles have been introduced. (The case with left parentheses is symmetric.) As the final step, we show that a cycle has actually been removed. The only possible outgoing edge in $\mathcal{D}_{G_2}$ from $[A_0\rangle u_0]$ leads to $[Z\rangle u_0]$. By construction, none of the productions on the form $A_0 \rightarrow \phi_i$ give rise to a cycle in $\mathcal{D}_{G_1}$ through $[A_0\rangle u_0]$, so $\mathcal{D}_{G_2}$ has no path from $[Z\rangle u_0]$ to $[A_0\rangle u_0]$ for any v. Hence, the cycle in $\mathcal{D}_{G_1}$ through $[A_0\rangle u_0]$ has no counterpart in $\mathcal{D}_{G_2}$. (Again, the case with left parentheses is symmetric.) □

If $\mathcal{D}_{G_2}$ contains another cycle, then we can repeat the above process to also eliminate that cycle. Termination of this repeated process is guaranteed because the number of simple cycles is finite and strictly decreasing in each round.

The result of the entire transformation process is a completely qualified grammar $\widehat{G}$ with a balanced language and no cycles in $\mathcal{D}_{\widehat{G}}$. We can then apply Knuth's algorithm from Theorem 3 in [12] to get an equivalent balanced grammar.

C Soundness and Relative Completeness

Recall that our analysis has a front-end that extracts a CFG from the Java program and a back-end that transforms the CFG to tag-form, checks well-formedness, and extracts a summary graph whose language is validated with respect to an XML schema. We will now consider soundness and completeness of this combined analysis of well-formedness and validity.

We say that a string $x \in \Sigma^*$ is *well-formed* if it is a well-formed XML document [4], and if well-formed, we say that x is *valid* with respect an XML schema $\mathcal{S}$ if it belongs to the language $\mathcal{L}(\mathcal{S})$ of the schema. A language $L \subseteq \Sigma^*$ is well-formed or valid with respect to $\mathcal{S}$ if every string in L is well-formed or valid with respect to $\mathcal{S}$, respectively. We use *single-type tree languages* for modeling schemas, which includes DTD and XML Schema [15].

The extraction of a CFG from a program can be made sound by constructing a flow graph that includes every possible runtime trace of the program, and by conservatively modeling all open and unknown parts (see e.g. [6]). Several well-known techniques can be applied to obtain good precision of the flow graph construction, but getting a precise answer is of course generally undecidable. The extraction of flow graphs and CFGs is therefore necessarily incomplete.

Perhaps more surprisingly, we will argue that the analysis of well-formedness and validity is both sound and complete *relative* to the CFGs produced by the front-end – as long as certain well-defined contrived situations do not occur in the program being analyzed. This property is not immediately apparent because the transformation from a CFG to a balanced grammar, as described in Appendices A and B, might have to approximate certain sub-grammars, which leads to a larger language of the resulting balanced grammar.

Let us say that a grammar G is *contrived* if there are element or attribute names in $\mathcal{L}(G)$ that are produced by non-linear sub-grammars. Such grammars can only appear in unnatural cases where a program recursively constructs names by printing some parts to the output stream *before* a recursive method call and printing other parts *after* the call as in this example:

```
void m(OutputStream out, int n) {
  out.print("<");
  rec(out, n);
  out.print(">");
}
```

```
void rec(OutputStream out, int n) {
  if (n==0) out.print("x></x");
  else {
    out.print("x");
    rec(out, n-1);
    out.print("x");
  }
}
```

The output language for the method `m` is $\{\texttt{<}x^n\texttt{></}x^n\texttt{>} \mid n \geq 1\}$, and a sub-grammar describing the tag names is clearly non-linear. Certainly, this program structure is not occurring in real servlet/JSP application code, and we will consequently assume that grammars being produced by the front-end are uncontrived.

Proposition 2 (Soundness and relative completeness). *Let $\mathcal{S}$ be a schema with a single-type tree language (the paper [15] explains how DTD and XML Schema fit into this category). Let G be a CFG with a balanced language (treating* `<` *as left parenthesis and* `</` *as right parenthesis as in Section 3) and let $\widehat{G}$ be the corresponding balanced grammar produced by the transformations described in Appendices A and B. Assume that G is uncontrived. Then $\mathcal{L}(G)$ is well-formed and valid with respect to $\mathcal{S}$ if and only if $\mathcal{L}(\widehat{G})$ is well-formed and valid with respect to $\mathcal{S}$.*

Proof. The only step that is not language preserving is from G_1 to G_2 in Appendix B. This step can only add strings to the language, so $\mathcal{L}(G) \subseteq \mathcal{L}(\widehat{G})$, which implies the "if" direction (soundness). The other direction (completeness) is shown in the following, first for well-formedness and then for validity.

When transforming from G_1 to G_2, we might produce strings in $\mathcal{L}(G_2) \backslash \mathcal{L}(G_1)$ only when productions on the form (1) are replaced by productions on the form (2), which, as noted in Appendix B, has the same language as the form $A_0 \to X^* \; Z \; Y^*$. This replacement is only performed when there is a $\mathcal{D}_G$ cycle through $[A_0 \rangle u]$ for some u, which implies that a parenthesis derived via α_i (or X^*) cannot match a parenthesis derived via ω_j (or Y^*) for any i, j and every string derived from Z in has at least one free parenthesis in X^*ZY^*. Every G_2 derivation of a string $x \in \mathcal{L}(G_2)\backslash\mathcal{L}(G_1)$ must by construction have the form

$$s \Rightarrow^+ \varphi_1\, A_0\, \varphi_2 \Rightarrow^+ \varphi_1\, \alpha\, \theta\, \omega\, \varphi_2 = x \tag{3}$$

where $\alpha \in \mathcal{L}(X)^*$, $\theta \in \mathcal{L}(Z)$, and $\omega \in \mathcal{L}(Y)^*$. However, by the definitions of X and Y we can find corresponding derivations in G_1:

$$s \Rightarrow^+ \varphi_1\, A_0\, \varphi_2 \Rightarrow^+ \varphi_1\, \alpha\, \theta\, \omega'\, \varphi_2 \tag{4}$$

$$s \Rightarrow^+ \varphi_1\, A_0\, \varphi_2 \Rightarrow^+ \varphi_1\, \alpha'\, \theta\, \omega\, \varphi_2 \tag{5}$$

where $\alpha' \in L(X)^*$ and $\omega' \in L(Y)^*$. Let us consider what this means to well-formedness of x. Since $\varphi_1\alpha\theta$ is a prefix of a well-formed string $\varphi_1\alpha\theta\omega'\varphi_2$, it makes sense to say that every symbol in $\varphi_1\alpha\theta$ belongs to some tag or character data in $\varphi_1\alpha\theta\omega'\varphi_2$, and similarly, every symbol in $\theta\omega\varphi_2$ belongs to some tag or character data in $\varphi_1\alpha'\theta\omega\varphi_2$. The substring θ must contain `<` or `</`, so every symbol in x belongs to a tag or some character data in a well-defined manner. The only question is now whether the tags in x are balanced.

Assume for a contradiction that the tags in x are not balanced. Since x is balanced with respect to `<` and `</`, the unbalancing must be caused by a start tag t_S whose name is not equal to the name of the corresponding end tag t_E. Since G is assumed to be uncontrived and (4) is well-formed, the entire name of t_S comes from the prefix $\varphi_1\alpha\theta$, and similarly, since (5) is also well-formed, the entire name of t_E comes from the suffix $\theta\omega\varphi_2$. From the observations above, $\alpha\theta\omega$ always contains a free parenthesis from θ. If that parenthesis is a `<`, then the matching `</` in x must come from φ_2. However, t_S matches t_E, so the name of t_E must come from θ. If we replace ω by ω' in x, then we have obtained a well-formed string without changing t_S and t_E, which contradicts the assumption that the tags in x are not balanced. The case where the free parenthesis from θ is of type `</` is symmetric. Hence, x is well-formed.

Let us now consider validity. As noted above, $\alpha\theta\omega$ always contains a free parenthesis (`<` or `</`) from θ. No tag (or character data) from α can therefore be sibling, ancestor, or descendant of any tag (or character data) from ω in the XML tree structure corresponding to x. By assumption, $\mathcal{L}(\mathcal{S})$ is a single-type tree language, so validity of the parent node of α is independent of ω and vice versa. Since both strings derived in (4) and (5) are valid with respect to $\mathcal{S}$, this means that x must also be valid. □

Proposition 2 gives that producing balanced grammars from uncontrived CFGs is sound and complete with respect to analysis of well-formedness and validity. The transformation of a balanced grammar to tag-form as described in Section 3 is language preserving. Also, the grammar remains uncontrived by the transformations, so disjointness of attribute names is decidable as required for completeness of the well-formedness check. The extraction of summary graphs presented in Section 4 is obviously sound and complete with respect to validity. We conclude that the analysis of well-formedness and validity is sound and complete relative to CFGs that are not contrived in the sense defined above.

Recent BRICS Report Series Publications

RS-06-8 **Christian Kirkegaard and Anders Møller.** ***Static Analysis for Java Servlets and JSP*****. April 2006. 22 pp.**

RS-06-7 **Petr Jančar and Jiří Srba.** ***Undecidability Results for Bisimilarity on Prefix Rewrite Systems*****. April 2006. 20 pp. Presented at** ***FoSSaCS 2006*****, LNCS 3921:277–291.**

RS-06-6 **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Bas Luttik.** ***A Finite Equational Base for CCS with Left Merge and Communication Merge*****. March 2006. 22 pp.**

RS-06-5 **Kristian Støvring.** ***Extending the Extensional Lambda Calculus with Surjective Pairing is Conservative*****. March 2006. 18 pp. To appear in** ***Logical Methods in Computer Science*****. Supersedes RS-05-35.**

RS-06-4 **Olivier Danvy and Kevin Millikin.** ***A Rational Deconstruction of Landin's J Operator*****. February 2006. ii+26 pp. To appear in the post-reviewed proceedings of the 17th International Workshop on the** ***Implementation and Application of Functional Languages*** **(IFL'05), Dublin, Ireland, September 2005.**

RS-06-3 **Małgorzata Biernacka and Olivier Danvy.** ***A Concrete Framework for Environment Machines*****. February 2006. ii+29 pp. To appear in the** ***ACM Transactions on Computational Logic*****. Supersedes BRICS RS-05-15.**

RS-06-2 **Mikkel Baun Kjærgaard and Jonathan Bunde-Pedersen.** ***A Formal Model for Context-Awareness*****. February 2006. 26 pp.**

RS-06-1 **Luca Aceto, Taolue Chen, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***On the Axiomatizability of Priority*****. January 2006. 25 pp.**

RS-05-38 **Małgorzata Biernacka and Olivier Danvy.** ***A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*****. December 2005. iii+39 pp. Revised version of BRICS RS-05-22.**

RS-05-37 **Gerth Stølting Brodal, Kanela Kaligosi, Irit Katriel, and Martin Kutz.** ***Faster Algorithms for Computing Longest Common Increasing Subsequences*****. December 2005. 16 pp.**