Extending Java for High-Level Web Service Construction

Aske Simon Christensen, Anders Møller*, and Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark

We incorporate innovations from the `<bigwig>` project into the Java language to provide high-level features for Web service programming. The resulting language, JWIG, contains an advanced session model and a flexible mechanism for dynamic construction of XML documents, in particular XHTML. To support program development we provide a suite of program analyses that at compile time verify for a given program that no runtime errors can occur while building documents or receiving form input, and that all documents being shown are valid according to the document type definition for XHTML 1.0.

We compare JWIG with Servlets and JSP which are widely used Web service development platforms. Our implementation and evaluation of JWIG indicate that the language extensions can simplify the program structure and that the analyses are sufficiently fast and precise to be practically useful.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Languages, Design, Verification

Additional Key Words and Phrases: Interactive Web services, data-flow analysis, XML

1. INTRODUCTION

The Java language is a natural choice for developing modern Web services. Its built-in network support, strong security guarantees, concurrency control, and widespread deployment in both browsers and servers, together with popular development tools, make it relatively easy to create interactive Web services. In particular, JavaServer Pages (JSP) [Sun Microsystems 2001b] and Servlets [Sun Microsystems 2001a], which are both Java-based technologies, have become immensely popular. However, both JSP, Servlets, and many other similar and widely used technologies, such as ASP, PHP, and CGI/Perl, suffer from some problematic shortcomings, as we will argue in the following and try to address.

1.1 Sessions and Web Pages

In general, JSP, Servlets, and related approaches provide only low-level solutions to two central aspects of Web service design: *sessions* and *dynamic construction of Web pages*.

A session is conceptually a sequential thread on the server that has local data, may access data shared with other threads, and can perform several interactions

*Corresponding author.
Authors' address: Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark.
Email: {aske,amoeller,mis}@brics.dk

with a client. With standard technologies, sessions must be encoded by hand, which is tedious and error-prone. More significantly, it is difficult to understand the control-flow of an entire service from the program source since the interactions between the client and the server are distributed among several seemingly unrelated code fragments. This makes maintenance harder for the programmer if the service has a complicated control-flow. Also, it prevents compilers from getting a global view of the code to perform whole-service program analyses.

The dynamic construction of Web pages is typically achieved by print statements that piece by piece construct HTML fragments from text strings. Java is a general-purpose language with no inherent knowledge of HTML, so there are no compile-time guarantees that the resulting documents are valid HTML. For static pages, it is easy to verify validity [Oskoboiny 2001], but for pages that are dynamically generated with a full programming language, the problem is in general undecidable. Not even the much simpler property of being well-formed can be guaranteed in this way. Instead of using string concatenation, document fragments may be built in a more controlled manner with libraries of tree constructor functions. This automatically ensures well-formedness, but it is more tedious to use and the problem of guaranteeing validity is still not solved.

The inability to automatically extract the control-flow of the sessions in a service raises another problem. Typically, the dynamically generated HTML pages contain input forms allowing the client to submit information back to the server. However, the HTML page with the form is constructed by one piece of the service code, while a different piece takes care of receiving the form input. These two pieces must agree on the input fields that are transmitted, but when the control-flow is unknown to the compiler, this property cannot be statically checked. Thorough and expensive runtime testing is then required, and that still cannot give any guarantees.

The `<bigwig>` language [Brabrand et al. 2002] is a research language designed to overcome these problems. Its core is a strongly typed C-like language. On top is a high-level notion of sessions where client interactions resemble remote procedure calls such that the control-flow is explicit in the source code [Brabrand et al. 1999]. Also, XHTML [Pemberton et al. 2000], the XML version of HTML, is a built-in data type with operations for dynamically constructing documents. The values of this data type are well-formed XHTML fragments which may contain "named gaps". Such fragments can be combined with a special "plug" operation which inserts one fragment into a gap in another fragment. This proves to be a highly flexible but controlled way of building documents.

In `<bigwig>`, the client interactions and the dynamic document construction are checked at compile time using a specialized type system [Sandholm and Schwartzbach 2000] and a program analysis [Brabrand et al. 2001] performing a conservative approximation of the program behavior to attack the problems mentioned above. More specifically, `<bigwig>` services are verified at compile time to ensure that (1) a plug operation always finds a gap with the specified name in the given fragment, (2) the code that receives form input is presented with the expected fields, and (3) only valid XHTML 1.0 is ever sent to the clients.

1.2 Contributions

In this paper we obtain similar benefits for Java applications. Our specific contributions are to show the following:

—how the session model and the dynamic document model of `<bigwig>` can be integrated smoothly into Java;

—how the type system from [Sandholm and Schwartzbach 2000] and the program analysis from [Brabrand et al. 2001] can be combined, generalized, and applied to Java to provide the strong static guarantees known from `<bigwig>` in a more powerful computational model; and

—how our service model subsumes and extends both the Servlet style and the JSP style of defining Web services.

The main weaknesses of `<bigwig>` is its core language and built-in libraries, which increasingly have shown to be inadequate when developing complex Web services. By replacing the core with Java, these limitations are amended. However, this is a nontrivial task, primarily because the type system and program analyses used in `<bigwig>` are not directly applicable to the much more complex Java language. As a consequence, the type system from [Sandholm and Schwartzbach 2000] is no longer used. Instead all type checking is based on a generalization of the notion of summary graphs introduced in [Brabrand et al. 2001].

The integration of the `<bigwig>` session model and dynamic document model into Java is achieved using a class library together with some extensions of the language syntax. The resulting language is called JWIG. Unlike `<bigwig>`, programs in Java and JWIG do not satisfy the closed world assumption, which is a requirement for our global flow analysis. This is handled by extending the analysis to make sound and conservative assumptions about the parts of the program that are not known at the time of the analysis.

JWIG generalizes `<bigwig>` in other ways as well. An HTML document is allowed to contain multiple forms, each of which may submitted independently. Fewer restrictions are imposed on form fields, so that they may appear in any combination, including, for example, mixtures of radio buttons and text fields with the same name, which was disallowed in `<bigwig>`. The JWIG analyses are extended to handle the fully general case. The analysis itself is also generalized, since we can now validate against DSD2 schemas, which have an expressive power similar to XML Schema, rather than the variant of DTDs that `<bigwig>` was restricted to consider. Additionally, a notion of *code gaps* is introduced, which provide a generalized version of the JSP-style code snippets occurring inside XML documents. Finally, the `<bigwig>` analyses relied on modifying the code to ensure that joining branches produced compatible XML documents; in contrast, the JWIG analysis is able to analyze the unmodified code.

When running a JWIG service without applying the static analyses, a number of special runtime errors may occur: If one of the three correctness properties mentioned in the previous section is violated, an exception is thrown. The goal of the static analyses is to ensure at compile time that these exceptions never occur. In addition to having this certainty, we can eliminate the overhead of performing runtime checks.

Such guarantees cannot be given for general Servlet or JSP programs, or for the many related languages that are being used for server-side programming, such as ASP, PHP and CGI/Perl. However, we show that the structures of such programs are special cases in JWIG: both the script-centered and the page-centered styles can be emulated by the session-centered, so none of their benefits are lost.

The predominant format of the data that is being transmitted between the servers and the clients in interactive Web services is HTML or the XML variant, XHTML. Our current implementation of JWIG uses XHTML 1.0, but the approach generalizes in a straightforward manner to an arbitrary interaction language described by an XML schema, such as WML or VoiceXML.

A cornerstone in our program analyses is the notion of *summary graphs*, which provide suitable abstractions of the sets of XML fragments that appear at runtime. We show how these graphs can be obtained from a data-flow analysis and that they comprise a precise description of the information needed to verify the correctness properties mentioned above.

Throughout each phase of our program analysis, we will formally define in what sense the phase is correct and we will give a theoretical bound on the worst-case complexity. We expect the reader to be familiar with Java and monotone data-flow analysis, and to have a basic understanding of HTML and XML.

1.3 Problems with Existing Approaches

In the following we give a more thorough explanation of the support for sessions and dynamic documents in JSP and Servlets and point out some related problems.

The overall structure of a Web service written with Servlets resembles that of CGI scripts. When a request is received from a client, a thread is started on the server. This thread generates a response, usually an HTML page, and perhaps has some side-effects such as updating a database. Before terminating it sends to the client the response, which is dynamically created by printing strings to an output stream. We call this a *script-centered* approach. The main advantages of Servlets compared to CGI scripts are higher performance and a convenient API for accessing the HTTP layer. A Servlet engine typically uses a thread pool to avoid the overhead of constantly starting threads. Also, Servlets have the general Java benefits of a sandboxed execution model, support for concurrency control, and access to the large set of available Java packages.

A small Servlet program is shown in Figure 1. This program consists of three individual Servlets: `Enter`, `Hello`, and `Goodbye`. A client running this service is guided through a sequence of interactions which we call a *session*: first, the service prompts for the client's name using the `Enter` Servlet, then the name and the total number of invocations are shown using the `Hello` Servlet, and finally a "goodbye" page is shown using the `Goodbye` servlet. The `ServletContext` object contains information shared to all sessions, while the `HttpSession` object is local to each session. Both kinds of state are accessed via a dictionary interface. Note that the three Servlets are tied together implicitly by the `action`s of the generated forms. An alternative code structure could be obtained by combining the three Servlets into a single one, using an extra session attribute to store the type of the current interaction. The Servlet API hides the details of cookies and URL rewriting which is used to track the client throughout the session. HTML documents are generated by printing

```
public class Enter extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Servlet Demo</title></head><body>" +
                "<form action=\"Hello\">" +
                "Enter your name: <input name=\"handle\">" +
                "<input type=\"submit\" value=\"Continue\"></form>" +
                "</body></html>");
} }

public class Hello extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String name = (String) request.getParameter("handle");
    if (name==null) {
      response.sendError(response.SC_BAD_REQUEST, "Illegal request");
      return;
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Servlet Demo</title></head><body>");
    ServletContext context = getServletContext();
    if (context.getAttribute("users")==null)
      context.setAttribute("users", new Integer(0));
    int users = ((Integer) context.getAttribute("users")).intValue() + 1;
    context.setAttribute("users", new Integer(users));
    HttpSession session = request.getSession(true);
    session.setAttribute("name", name);
    out.println("<form action=\"Goodbye\">" +
                "Hello " + name + ", you are user number " + users +
                "<input type=\"submit\" value=\"Continue\"></form>" +
                "</body></html>");
} }

public class Goodbye extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    HttpSession session = request.getSession(false);
    if (session==null) {
      response.sendError(response.SC_BAD_REQUEST, "Illegal request");
      return;
    }
    String name = (String) session.getAttribute("name");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Servlet Demo</title></head><body>" +
                "Goodbye " + name + "</body></html>");
    session.invalidate();
} }
```

Fig. 1. Example Servlet program. HTML documents are constructed by printing string fragments to a stream, and the session flow is encoded into the `action` attributes in the forms.

```
<html><head><title>JSP Demo</title></head><body>
Hello <%
  String name = request.getParameter("who");
  if (name==null) name = "stranger";
  out.print(name);
%>!
<p>
This page was last updated: <%= new java.util.Date() %>
</body></html>
```

Fig. 2. Example JSP page. Code snippets are embedded within the HTML code using special `<%...%>` tags. When a client requests the page, the code snippets are replaced by the strings that result from the evaluation.

strings to the output stream.

A JSP service turns the picture inside out by being defined by an HTML document with embedded code snippets. We call this a *page-centered* approach. Figure 2 shows a simple JSP program which dynamically inserts the current time together with a title and a user name based on the input parameters. This approach is quite similar to ASP and PHP, except that the underlying language and its runtime model are safer and better designed. An implementation of JSP typically performs a simple translation into Servlets. This model fits into situations where the service presents pages that are essentially static but with a few dynamic fragments inserted. For more complicated services the code tends to dominate the pages, such that they converge toward straight Servlet implementations.

Both JSP and Servlets allow only a single interaction with the client before termination. To simulate a sequential thread, the client is given a session identifier that is stored either as a cookie, as an SSL session key, or in a hidden form field. The local state associated with the thread is stored as attributes in an `HttpSession` object from which it must be recovered when execution is later resumed. Thus, a sequence of interactions must be encoded by the programmer in a state machine fashion where transitions correspond to individual interactions. This is somewhat cumbersome and precludes cases where the resident local state includes objects in the heap or a stack of pending method invocations. However, it should be noted that the state machine model implies other advantages when it is applicable. Specifically, it allows robust and efficient implementations as evidenced by J2EE engines that ensure scalability and transaction safety by dividing session interactions into atomic transitions. If a Web service runs on a J2SE engine, then only the disadvantages exist since every interaction is then typically handled by a new thread anyway.

Data that is shared between several session threads must in JSP and Servlets be stored in a dictionary structure or an external databases. This is in many cases adequate, but still conceptually unsatisfying in a language that otherwise supports advanced scoping mechanisms, such as nested classes.

Finally, JSP and Servlets offer little support for the dynamic construction of XHTML or general XML documents. JSP allows the use of static templates in which gaps are filled with strings generated by code fragments. Servlets generate all documents as strings. These techniques cannot capture well-formedness of the generated documents, let alone validation according to some schema definition.

Well-formedness can be ensured by relying on libraries such as JDOM [Hunter and McLaughlin 2001], where XML documents are constructed as tree data structures. However, this loses the advantages from JSP of using human readable templates and validity can still only be guaranteed by expensive runtime checks.

Another concern in developing Web services is to provide a separation between the tasks of programmers and designers. With JSP and Servlets, the designer must work through the programmer who maintains exclusive control of the markup tags being generated. This creates a bottleneck in the development process.

The JWIG language is designed to attack all of these problems within the Java framework. For a more extensive treatment of Servlets, JSP, and the many related languages, we refer to the overview article on `<bigwig>` [Brabrand et al. 2002]. The central aspects of JWIG are an explicit session model and a notion of higher-order XML templates, which we will explain in detail in the following sections.

1.4 Other Related Languages

There are several other language proposals that relate to our work.

Struts [McClanahan et al. 2002] is an application framework for building Web applications in Java. It offers support for the Model-View-Controller design pattern through special Servlet applications and custom JSP tags. A consistent use of the framework will make applications simpler to understand for the programmer, but the compiler will still just see a straight Servlet application running on the standard platform. Thus, the problems mentioned above for Servlets and JSP still apply.

Castor XML [Exolab Group 2002] is an XML data-binding framework for Java. From an XML Schema it can generate a collection of Java classes representing an object model of the corresponding XML documents. Marshaling and unmarshaling methods are automatically generated. XML documents may then be constructed in a less generic manner than using JDOM. However, manipulations are still constructor-based and there is no static guarantee that a constructed document will satisfy all the requirements of the originating schema. JAXB [Sun Microsystems 2002] is a similar initiative that works on an subset of XML Schema and allows runtime validation of constructed XML documents.

WSDL [Christensen et al. 2001] is an interface definition language that defines an abstract view of a Web service, listing the available operations whose arguments and results are typed using XML Schema. A WSDL document may serve as documentation for an existing Web service or dually be used to generate stub code in a particular implementation language, which could for example be JWIG.

SOAP [Box et al. 2000] is a concrete XML language for defining envelopes for XML documents that are to be transmitted between Web services. The main connection to JWIG is that our static analysis could be used to guarantee that all generated SOAP envelopes will be syntactically correct.

PHP [Atkinson 2000] and ASP [Homer et al. 2001] are both fundamentally similar to JSP, but use different underlying languages, which are less structured than Java. Thus, all the problems mentioned above still apply. The same holds for Server-Side JavaScript [Netscape 1999], which allows the evaluation of JavaScript on the server before an HTML document is sent to the client.

1.5 Outline

We begin by describing the special JWIG language constructs and packages in Section 2. These extensions are designed to allow flexible, dynamic construction of documents and coherent sessions of client interactions. In itself, this section explains and motivates the design of a Web programming framework that can be used independently of the remaining parts of this paper. In Section 3 we explain how to obtain JWIG program *flow graphs* that abstractly describe the flow of strings and XML templates through programs. This is done by analyzing class files that are obtained by compiling instances of our framework using any standard Java compiler. This analysis contains several technical challenges since we must consider all aspects of the full Java language. The obtained flow graphs form the basis of the data-flow analyses that are then described in Section 4. We use the standard monotone framework with a special lattice of *summary graphs* that denote possibly infinite sets of XML documents. The data-flow analysis computes for every program point and every variable a summary graph that conservatively approximates the set of XML values that may occur at runtime. To compute this information we need a preliminary *string analysis* which computes regular set approximations of the string values that may appear during execution of the program. In Section 5 we describe how the results from the summary graph analysis are used to verify that the runtime errors mentioned earlier cannot occur for a given program. This involves the use of a novel XML schema language, *Document Structure Description 2.0*, which we briefly describe and motivate. We provide an overview of our implementation in Section 6, and evaluate it on some benchmark programs to show that the analysis techniques are sufficiently fast and precise to be practically useful. Finally, we describe ideas for future work in Section 7.

2. THE JWIG LANGUAGE

The JWIG language is designed as an extension of Java. This extension consists of a service framework for handling session execution, client interaction and server integration, and some syntactic constructs and support classes for dynamic construction of XML documents.

2.1 Program Structure

A JWIG application is a subclass of the class `Service` containing a number of fields and inner classes. An instance of this class plays the role of a running *service*.

A service contains a number of different *sessions*, which are defined by inner classes. Each session class contains a `main` method, which is invoked when a client initiates a session. The fields of a service object are then accessible from all sessions. This provides a simple shared state that is useful for many applications. Concurrency control is not automatic but can be obtained in the usual manner through `synchronized` methods. Of course, external databases can also be applied, for instance using JDBC, if larger data sets are in use. The fields of a session object as well as local variables in methods are private to each session thread. This approach of applying the standard scope mechanisms of Java for expressing both shared state and per-session state is obviously simpler than using the `ServletContext` and `HttpSession` dictionaries in Servlets and JSP.

```
import java.io.*;
import dk.brics.jwig.runtime.*;

public class MyService extends Service {
  int counter = 0;
  synchronized int next() { return ++counter; }

  public class ExampleSession extends Session {
    XML wrapper =
      [[ <html><head><title>JWIG Demo</title></head>
         <body><[body]></body></html> ]];
    XML form =
      [[ <form><[contents]>
         <input type="submit" value="Continue" /></form> ]];
    XML hello =
      [[ Enter your name: <input name="handle" /> ]];
    XML greeting =
      [[ Hello <[who]>, you are user number <[count]> ]];
    XML goodbye =
      [[ Goodbye <[who]> ]];

    public void main() throws IOException {
      XML x = wrapper<[body=form];
      show x<[contents=hello];
      String name = receive handle;
      show x<[contents=greeting<[who=name,count=next()]];
      exit wrapper<[body=goodbye<[who=name]];
}}}
```

Fig. 3. Example JWIG program. The `MyService` service contains one session type named `ExampleSession` which has the same functionality as the Servlet service in Figure 1.

Figure 3 shows a JWIG service which is equivalent to the Servlet service from Figure 1. In the following, we describe the new language constructs for defining XML templates, showing documents to the clients, and receiving form input.

Session classes come in a number of different flavors, each with different purposes and capabilities, as indicated by its superclass:

—`Service.Session` is the most general kind of interaction pattern, allowing any number of interactions with the client while retaining an arbitrary session state. When a `Service.Session` subclass is initiated, a new thread is started on the server, which lives throughout all interactions with the client. At all intermediate interactions, after supplying the XML to be sent to the client, the thread simply sleeps, waiting for the client to respond.

—`Service.Page` is used for simple requests from the client. A `Service.Page` is similar to a `Service.Session`, except that it allows no intermediate client interactions. This is conceptually similar to the mechanism used in Servlet and JSP applications, where a short-lived thread is also initiated for each interaction.

—`Service.Seslet` is a special kind of session, called a *seslet*, that is used to interact with the service from applets residing on the client or with other Web services. A `Service.Seslet` does not interact with the client directly in the form of input forms and, for example, XHTML output; instead, it is parameterized with an

`InputStream` and an `OutputStream` which are used for communication with the applet or Web service on the client-side. The notion of seslets was introduced in [Brabrand et al. 2002].

For the remainder of this article we focus on `Service.Session`. In contrast to the session API in Servlets and JSP, it provides a clear view of the service flow because the sequences of interactions constituting sessions are explicit in the program control-flow. This *session-centered* approach originates from the MAWL project [Ladd and Ramming 1996].

We specify `Service.Page` by a separate class in order to illustrate that the script- and page-centered approaches are special cases of the session-centered approach, and to identify applications of these simpler interaction models to allow implementations to perform special optimizations.

2.2 Client Interaction

Communication with the client is performed through the `show` statement, which takes as argument an XML template to be transmitted to the client. A session terminates by executing the `exit` statement whose argument is an XML template that becomes the final page shown. Intermediate XML templates need not conform to any XML schema, but when they are used as arguments to `show` or `exit` statements they must be valid XHTML 1.0 [Pemberton et al. 2000]. Otherwise, a `ValidateException` is thrown.

During client interactions, the session thread is suspended on the server. Thus, the execution of the `show` statement behaves as a remote procedure call to the client. Return values are specified by means of form fields in the document. Such `show` operations may be performed at any place in the code and, in contrast to Servlets and JSP, the full state of the session thread—including the invocation stack—is automatically preserved. For all `form` elements, a default `action` attribute is automatically inserted with a URL pointing to the session thread on the server. The responses from the client are subsequently obtained using the `receive` expression, which takes as argument the name of an input field in the XHTML document that was last shown to the client and returns the corresponding value provided by the client as a `String`. If no such input field was shown to the client, then no corresponding value is transmitted and a `ReceiveException` is thrown. There may be several occurrences of a given input field. In this case, all the corresponding values may be received in order of occurrence in the document into a `String` array using the expression `receive[]`. The nonarray version is only permitted if the input field occurs exactly once; otherwise, a `ReceiveException` is thrown. The array version cannot fail: in case there are no fields, the empty array is produced.

Figure 4 illustrates the interactions of a session. In the JWIG example service in Figure 3, the `main` method contains three client interactions: two `show` statements and one `exit` statement. Clearly, the session flow is more explicit than in the corresponding Servlet code in Figure 1. The XHTML documents are constructed from five constant XML templates using plug operations as described in the next section.

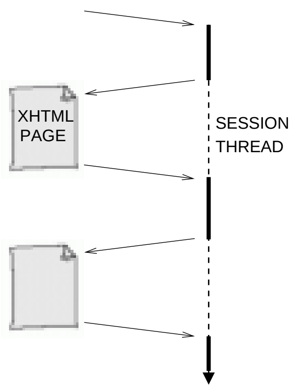


Fig. 4. Client-server sessions in Web services. On the left is the client's browser, on the right is a session thread running on the server. The tread is initiated by a client request and controls the sequence of session interactions.

2.3 Dynamic Document Construction

In Servlets and JSP, document fragments are generated dynamically by printing strings to a stream. In JWIG, we instead use a notion of XML *templates*. A template is a well-formed XML fragment which may contain named *gaps*. A special *plug* operation is used to construct new templates by inserting existing templates or strings into gaps in other templates. These templates are higher-order, because we allow the inserted templates to contain gaps which can be filled in later, in a way that resembles higher-order functions in functional programming languages. Templates are identified by a special data type, `XML`, and may be stored in variables and passed around as any other type. Once a complete XHTML document has been built, it can be used in a `show` statement.

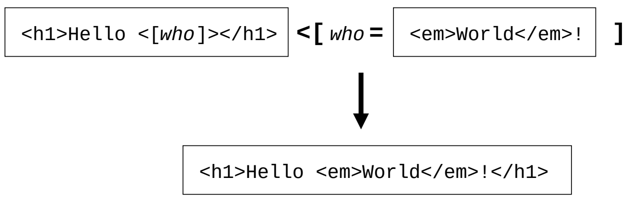
The idea of "contexts with holes" is of course widely used in computer science. One different aspect of our templates is that gap names are significant and may be captured by enclosing templates. Thus, no alpha renaming takes place.

Syntactically, the JWIG language introduces the following new expressions for dynamic XML document construction:

[[*xml* **]]**	(template constant)
exp_1 **<[**g **=** exp_2**]**	(the plug operator)
([[*xml* **]])** *exp*	(XML cast)
get *url*	(runtime template inclusion)

These expressions are used to define template constants, to plug templates together, to cast values to the `XML` type, and to include template constants at runtime, respectively. The *url* denotes a URL of a template constant located in a separate file, and *xml* is a well-formed XML template according to the following grammar:

xml	:	*str*	(character data)
	|	**<***name atts***>** *xml* **</***name***>**	(element)
	|	**<[***g***]>**	(template gap)
	|	**<{***stm***}>**	(code gap)
	|	*xml xml*	(sequence)



```
<h1>Hello <[who]></h1>  <[ who = <em>World</em>! ]
<h1>Hello <em>World</em>!</h1>
```

Fig. 5. The plug operation. The two XHTML templates in the top are combined to produce the one below by plugging into the *who* gap.

atts	:	ε	(empty)
	|	*name*=**"***str***"**	(attribute constant)
	|	*name*=**[***g***]**	(attribute gap)
	|	*atts atts*	(sequence)

Here, *str* denotes an arbitrary Unicode string, *name* an arbitrary identifier, *g* a gap name, and *stm* a statement block that returns a value of type `String` or `XML`. Actual XML values must of course be further constrained to be well-formed according to the XML 1.0 specification [Bray et al. 2000]. Moreover, in this description we abstract away all DTD information, comments, processing instructions, etc. In Figure 3, there are four template constants: `wrapper`, `hello`, `greeting`, and `goodbye`. The `greeting` template, for instance, contains two gaps named *who* and *count*, respectively.

XML templates can be composed using the plug operation exp_1 **<[**g **=** exp_2**]**. The result of this expression is a copy of exp_1 with all occurrences of the gap named g replaced by copies of exp_2. This is illustrated in Figure 5. If exp_2 is a string, all special XML characters (`<`, `>`, `&`, `'`, and `"`) are automatically escaped by the corresponding XML character references. If exp_1 contains no gaps named g, a `PlugException` is thrown. A gap that has not been plugged is said to be *open*. The exp_2 expression is required to be of type `String` or `XML`. If it is not one of these, it is coerced to `String`. There are three kinds of gaps: template gaps, attribute gaps, and code gaps. Both strings and templates may be plugged into template gaps, but only strings may be plugged into attribute gaps. Attempts to plug templates into attribute gaps will cause a `PlugException` to be thrown. When a template is shown, all remaining open gaps are removed in the following way: each template gap is replaced by the empty string, and for each attribute gap, the entire attribute containing the gap is removed. As an example, this removal of attributes is particularly useful for checkboxes and radio buttons where `checked` attributes either must have the value `checked` or be absent. One can use a template containing the attribute gap `checked=[c]` and then plug `checked` into `c` whenever the field should be checked and simply omit the plug otherwise.

The code gaps are not filled in using the plug operation: instead, when a template containing code gaps is shown, the code blocks in the code gaps are executed in document order. The resulting strings or templates are then inserted in place of the code. Because this does not happen until the template is shown, the code in code gaps can only access variables that are declared in the service class or locally within the code gap.

The plug operation is the only way of manipulating XML templates. Thus, our `XML` type is quite different from both the string output streams in Servlets and JSP and the explicit tree structures provided by, for instance, JDOM. We exploit this to obtain a compact and efficient runtime representation, and as a foundation for performing our program analyses. The notion of code gaps allows us to directly emulate the JSP style of writing services, which is often convenient, but while still having the explicit notion of sessions and the guarantees provided by the program analyses.

In order to be able to perform the static analyses later, we need a simple restriction on the use of forms and input fields in the XML templates: in `input` and `button` elements we require syntactically that attributes named `type` and `multiple` cannot occur as attribute gaps in elements defining input fields. The same restriction holds for `name` attributes, unless the `type` attribute has value `submit` or `image`. In addition, we make a number of simple modifications of the documents being shown and of the field values being received:

(1) In HTML and XHTML, lists, tables, and select menus are not allowed to have zero entries. However, it is often inconvenient to be required to perform special actions in those cases. Just before a document is shown, we therefore remove all occurrences of `<ul></ul>` and similar constructs. For select menus, we add a dummy option in case none are present. The time required to perform this postprocessing is negligible.

(2) If attempting to receive the selected value of a `select` menu that is not declared as `multiple` or the value of a `radio` button set, then, if no option is either preselected using `checked` or selected by the client, the null value is received instead of throwing a `ReceiveException`—even though no name-value pair is sent according to the XHTML 1.0/HTML 4.01 specification [Raggett et al. 1999].

(3) For `submit` and `image` fields, we change the corresponding returned name-value pair from X=Y to `submit`=X. This makes it easier to recognize the chosen submit button in case of graphical buttons.

(4) For `submit` buttons, a single name-value pair is produced from the `name` and the `value` attributes. However, for graphical submit buttons, that is, fields with `type="image"`, HTML/XHTML produces two name-value pairs, X`.x` and X`.y` for the click coordinates, where X is the value of the `name` attribute, but the `value` attribute is ignored. To obtain a clean semantics, we ensure by patching the returned list of pairs that, in every case, all three name-value pairs are produced. For instance, with graphical submit buttons we add a `submit`=X pair, and `submit.x` and `submit.y` contain the click coordinates. For normal submit buttons, `submit.x` and `submit.y` contain the value `-1`.

Clearly, these restrictions and modifications do not impose any practical limitations on the flexibility of the template mechanism. In fact, they serve as a convenience to the programmer since many special cases in XHTML need not be considered.

As for other Java types, casting is required when generic containers or methods are used. An XML template may be cast using the special syntax **`([[`***xml***`]])`***exp*.

This is a promise from the JWIG programmer that, in the program analyses described later, any value that will ever be contained in *exp* at this point may be replaced by the given constant. If this is the case, the cast is said to be *valid.* At runtime, a `CastException` is thrown if the sets of gaps and input fields in *exp* do not match those in *xml*. If casting to a template with exactly one occurrence of a given field, then it is required that the actual values also has exactly one occurrence of that field—except for radio buttons, where multiple occurrences count as one. For the gaps, two properties must be satisfied: if the template being cast to has any gaps of a given name, then at least one such gap must also exist in the actual value; and, if there is a template gap in the template being cast to, then the actual value cannot contain any attribute gaps of that name. Note that this runtime check is not complete since it only considers gaps and input fields and not XHTML validity. However, if invalid XHTML is produced, it will eventually result in a `ValidateException` at a `show` statement.

Alternatively, the ordinary cast `(XML)`*exp* may be used. It generates a `CastException` if the actual type of *exp* is not `XML`, but no promises are made about the gaps or input fields.

Large JWIG applications may easily involve hundreds of template constants. For this reason, there is support for specifying these externally, instead of inlined in the program source. The construct `get` *url* loads the XML template located at *url* at runtime. This template can then later be modified and reloaded by the running service.

When the service is analyzed, the template constant referred to by the `get` *url* construct is loaded and treated as a constant in the analysis. The analysis is then of course only valid as long as the template is unchanged. However, validity will be preserved if the template remains structurally the same. To obtain fresh security guarantees, it is simply required to reinvoke the program analyzer.

These features can also be used to support cooperation between the programmers and the Web page designers. For a first draft, the programmers can create some templates that have the correct structure but only a primitive design. While the program is being developed using these templates, the designers can work on providing a more sophisticated design. The program analyzer will ensure that the structure of gaps and fields is preserved. This eliminates a bottleneck that is sometimes present in Web projects, where designers are required to have changes implemented by programmers, since the HTML they produce must be intermixed with the code.

In addition to the main features mentioned above, a session object contains a number of fields and methods that control the current interaction, such as access control using HTTP authentication, cookies, environment variables, SSL encryption, and various timeout settings. The service object additionally contains a `checkpoint` method which serializes the shared data and stores it on disk. This is complemented by a `rollback` method which can be used in case of server crashes to improve robustness.

To summarize, we have now added special classes and language constructs to support session management, client interactions, and dynamic construction of XHTML documents. By themselves, we believe that these high-level language extensions aid development of Web services. The extensions may cause various exceptions: a

`ValidateException` if one attempts to show an XML document which is not valid XHTML 1.0; a `ReceiveException` if trying to receive an input field that occurs an incompatible number of times; a `PlugException` if a plug operation fails because no gaps of the given name and type exist in the template; and a `CastException` if an illegal cast is performed because the gaps or fields do not match. In the following sections, we show that it is possible to statically check whether the first three kinds of exceptions can occur. This is possible only because of the program structure that the new language constructs enforce. For instance, it is far from obvious that similar guarantees could be given for Servlet or JSP services.

2.4 Declarative Form Field Validation Using PowerForms

Many existing Web services apply intricate JavaScript client-side code in the Web documents for checking that input forms are filled in consistently. For instance, it is typical that certain fields must contain numbers or email addresses, or that some fields are optional depending on values entered in other fields. Proper error messages need to be generated when errors are detected such that the clients have the chance to correct them. Since client-side JavaScript execution cannot be trusted, extra checks need to be performed on the server.

The PowerForms language [Brabrand et al. 2000] has been developed to attack the problem of specifying form input validation requirements in a more simple and maintainable way based on regular expressions and boolean logic. Using an XML notation, a PowerForms document specifies form field validation requirements. Such a specification is automatically translated into JavaScript code that incrementally performs the checking on the client-side and into Java code that performs an extra check on the server-side. We omit a more thorough description of the PowerForms language and instead refer the reader to [Brabrand et al. 2000; Christensen and Møller 2002].

As an extra feature of JWIG, PowerForms has been fully integrated. PowerForms documents can be constructed using the `XML` data type and plug operations that have been described in the previous section. A variant of the `show` statement connects JWIG and PowerForms: `show` X `powerforms` P. When executed, if P is not a valid PowerForms document, a `ValidateException` is thrown. Otherwise P is translated into JavaScript code, which is inserted into X, and the resulting document is shown to the client as usual. When a reply is submitted and execution resumes on the server, the extra server-side check is performed. If JavaScript execution has been disabled or forged on the client and the submitted form input is not valid, an XHTML document with a detailed error message is returned, so the session cannot be continued until the errors are corrected by the client.

The program analysis, which is described later, for checking validity of the generated XHTML documents is additionally applied to check that the P document is a valid PowerForms document according to the document type definition of PowerForms.

2.5 Emulation of Script- and Page-Centered Models

The code structure in Servlets closely follows the request–response interaction pattern from HTTP. That structure is essentially the same as the one supported by `Service.Page` in JWIG, with two exceptions: (1) while shared data in Servlets is typ-

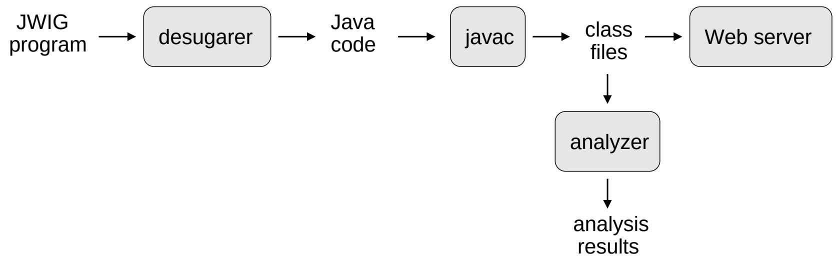


Fig. 6. The JWIG program translation process in our current implementation. The JWIG program is desugared into Java code, which is compiled to class files. These class files are used both in the Web server and to perform the program analyses.

ically managed by `ServletContext` objects using the `setAttribute` and `getAttribute` methods, we use nested classes in JWIG to distinguish between shared and local data; a shared variable is simply declared in the outer class and can be accessed as any other variable. (2) The construction of the reply HTML page is in Servlets generated by concatenating string fragments, whereas in JWIG, we use the more structured notion of templates with gaps and specialized plug operations.

Similarly, the JSP style, where code is embedded within constant HTML pages, can be emulated by the use of code gaps: A JSP file essentially has the same structure as a `Service.Page` program that just returns a constant HTML template containing a number of code gaps.

Since the JWIG program analyses depend on the fact that the HTML pages are constructed with templates and plug operations, it is presumably not feasible to automatically translate preexisting Servlet or JSP code into JWIG code or to apply our program analysis techniques to Servlets. However, using `Service.Page` and code gaps, JWIG programmers are free to apply the script-centered or page-centered programming styles, if any need should arise, or even to write services that apply a mixture of script-, page-, and session-centered code. In addition, the unique static guarantees of, for instance, XHTML validity, are available both when using `Service.Page` and the more general `Service.Session`. In this sense, JWIG subsumes and extends the Servlet and JSP styles.

For a simple Web service that is structured as a collection of individual pages, which the clients can browse freely, the page- and script-centered approaches work well. With such services, the main advantages of using JWIG are the simpler approach of managing shared data and the static guarantees of validity of the generated pages.

For more complex Web services, the need for tracking and guiding the individual clients through sessions is more important. The JWIG style of modeling interactions as remote procedure calls from the server to the client gives a more comprehensible control-flow in the code than possible with the Servlet or JSP style, where the service code is structured as a collection of event handlers. Not only the programmers but also the compiler can benefit from this increased comprehensibility: the JWIG style also makes it possible for a program analyzer to statically check the correspondence between input fields in the HTML pages and the server code receiving the values.

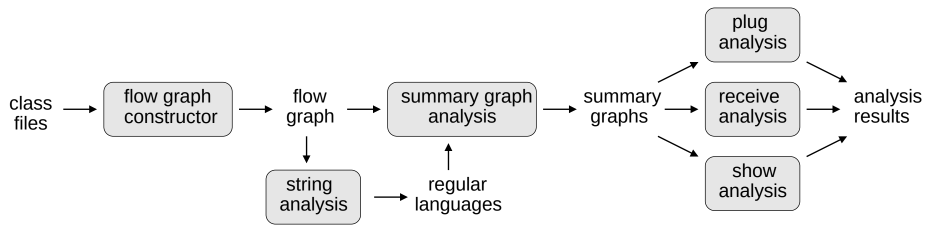


Fig. 7. Structure of the program analyzer.

2.6 The JWIG Program Translation Process

The steps involved in compiling, analyzing, and running a JWIG program are depicted in Figure 6. First, the special syntactic constructs of JWIG are translated into appropriate Java constructs by a simple source-to-source desugaring transformation. The resulting Java source files are compiled into Java class files as usual. These class files together with the accompanying externally specified XML template constants constitute the Web service. Of course, an implementation is not forced to have this structure: for instance, one could imagine a JWIG compiler that directly produces class files instead of going via Java code.

The analysis works on the class file level. When the analyzer is invoked, it is given a collection of class files to analyze. We call this collection the *application classes*; all others constitute the *nonapplication classes*. Exactly one of the application classes must be a subclass of `Service`. For efficiency reasons, the application classes can be just the few classes that actually constitute the JWIG service, not including all the standard Java classes that the program uses. Our analyses are designed to cope with this limited view of the program as an open system.

The soundness of the analyses that we describe in the following sections is based on a set of well-formedness assumptions:

—all invocation sites in the application classes must either always invoke methods in the application classes or always invoke methods in the nonapplication classes;

—no fields or methods of application classes are accessed by a nonapplication class;

—no XML operations are performed in nonapplication classes; and

—XML casts are always valid, according to the definition in Section 2.3.

These assumptions usually do not limit expressibility in practice. In some cases, the second assumption can be relaxed slightly, for instance, if some method called from a nonapplication class does not modify any `String` or `XML` value that will ever reach other application class methods. This makes it possible to safely use callback mechanisms such as the `Comparator` interface. The assumption about casts is deliberately quite strong: as ordinary casts, XML casts provide a back-door to the programmer to bypass the static type system.

The structure of the program analyzer is shown in Figure 7. From the class files, we first generate *flow graphs*. From these, we generate *summary graphs*, which we analyze in three different ways corresponding to the properties mentioned in the previous section.

2.7 An Example JWIG Program

We will use the following JWIG program throughout the remaining sections to illustrate the various phases of the analysis. This admittedly rather artificial service applies most JWIG-specific language constructs:

```
import dk.brics.jwig.runtime.*;

public class Greetings extends Service {
  String greeting = null;

  public class Welcome extends Session {
    XML cover = [[ <html>
                     <head><title>Welcome</title></head>
                     <body bgcolor=[color]>
                       <{
                        if (greeting==null)
                          return [[ <em>Hello World!</em> ]];
                        else
                          return [[ <b><[g]></b> ]] <[g=greeting];
                       }>
                       <[contents]>
                     </body>
                    </html> ]];

    XML getinput = [[ <form>Enter today’s greeting:
                       <input type="text" name="salutation">
                       <input type="submit"></form> ]];

    XML message = [[ Welcome to <[what]>. ]];

    public void main() {
      XML h = cover<[color="white",contents=message];
      if (greeting==null) {
        show cover<[color="red",contents=getinput];
        greeting = receive salutation;
      }
      exit h<[what=[[<b>BRICS</b>]]];
} } }
```

The first time the `Welcome` session is initiated, the client is prompted for a greeting text in one interaction, and then in the next interaction the greeting is shown together with a “Welcome to BRICS” message. For subsequent sessions, only the second interaction is performed.

3. FLOW GRAPH CONSTRUCTION

Given a JWIG program, we first construct an abstract flow graph as a basis for the subsequent data-flow analyses. The flow graph captures the flow of string and XML template values through the program and their uses in show, plug, and receive operations, while abstracting away other data types and Java features. We first define the structure of flow graphs, and then, in order to be able to define in what sense a translation of JWIG code into flow graphs is correct, we formally define their semantics. After that, we present a concrete translation of JWIG code into flow graphs.

3.1 Structure of Flow Graphs

A flow graph consists of nodes and edges. The *nodes* correspond to abstract statements:

x **=** *exp***;**	(assignment)
show *x***;**	(client interaction)
receive *f***;**	(receive field)
receive[] *f***;**	(receive field array)
nop;	(no operation)

where *exp* denotes an expression of one of the following kinds:

x	(variable read)
"*str*"	(string constant)
[[*xml* **]]**	(XML template constant)
x **<[** *g* **=** *y* **]**	(plug operation)
null	(null value)
anystring	(arbitrary string)

and x and y are program variables, g is a gap name, f is a field name, *str* is a string constant, and *xml* is an XML template constant that does not contain any code gaps. All code gaps in the original JWIG program are expressed using normal gaps and plug operations in the flow graph, as will be explained in Section 3.3.

We assume that every variable occurring in the flow graph has a declared type: STRING representing strings, or XML representing XML templates. These types are extended to expressions as one would expect, and `null` has the special type NULL. Let EXP_{STRING} denote the expressions of type STRING or NULL, and EXP_{XML} denote those of type XML or NULL.

The assignment statement evaluates its expression and assigns the value to the given variable. The variable and the expression must have the same type. All flow graph variables are assumed to be declared with a global scope. Evaluating expressions cannot have side-effects. The argument to `show` statements is always of type XML. As described later, we model `receive` expressions from the JWIG program as pairs of statements, each consisting of a `receive` statement and an assignment. The `receive` and `receive[]` statements record program locations where input field values are received. The last kind of flow-graph statement, `nop`, is the no-operation statement which we use to model all operations that are irrelevant to our analyses, except for recording split and join points.

The expressions basically correspond to those in concrete JWIG programs, except `anystring` which is used to model standard library string operations where we do not know the exact result. In the plug operation, the first variable always has type XML, and the second has type XML or STRING.

Each node in the graph is assigned an *entry label* and an *exit label* as in [Nielson et al. 1999], and additionally each XML template constant has a *template label.*

All labels are assumed to be unique. The union of entry labels and exit labels constitute the *program points* of the program.

The graph has two kinds of edges: *flow edges* and *receive edges*. A flow edge models the flow of data values between the program points. Each edge has associated one source node and one destination node, and is labeled with a set of program variables indicating which values that are allowed to flow along that edge. A receive edge goes from a `receive` node to a `show` node. Its presence indicates that the control-flow of the program may lead from the corresponding `show` statement to the `receive` statement without reaching another `show` first. We use these edges to describe from which `show` statements the received field can originate.

3.2 Semantics of Flow Graphs

Formally, the semantics of a flow graph is defined by a constraint system. Let V be the set of variables that occur in the flow graph, XML be the set of all XML templates, and $STRING$ be the set of all strings over the Unicode alphabet. Each program point ℓ is associated an environment $\mathcal{E}_\ell$:

$$\mathcal{E}_\ell : V \to 2^{XML \,\cup\, STRING}$$

The entire set of environments forms a lattice ordered by pointwise set inclusion. For each node in the graph we generate a constraint. Let *entry* and *exit* denote the entry and exit labels of a given node. If the statement of the node is an assignment, x `=` exp`;`, then the constraint is

$$\mathcal{E}_{exit}(y) = \begin{cases} \widehat{\mathcal{E}}_{entry}(exp) & \text{if } \; x = y \\ \mathcal{E}_{entry}(y) & \text{if } \; x \neq y \end{cases}$$

For all other nodes, the constraint is

$$\mathcal{E}_{exit} = \mathcal{E}_{entry}$$

The map $\widehat{\mathcal{E}}_\ell : EXP \to 2^{XML \,\cup\, STRING}$ defines the semantics of flow graph expressions given an environment $\mathcal{E}_\ell$:

$$\widehat{\mathcal{E}}_\ell(exp) = \begin{cases} \mathcal{E}_\ell(x) & \text{if } \; exp = x \\ \{str\} & \text{if } \; exp = \texttt{"}str\texttt{"} \\ \{xml\} & \text{if } \; exp = \texttt{[[}\; xml \;\texttt{]]} \\ \pi(\mathcal{E}_\ell(x), g, \mathcal{E}_\ell(y)) & \text{if } \; exp = x \;\texttt{<[}\; g \;\texttt{=}\; y \;\texttt{]} \\ \emptyset & \text{if } \; exp = \texttt{null} \\ STRING & \text{if } \; exp = \texttt{anystring} \end{cases}$$

The function π captures the meaning of the plug operation. Due to the previously mentioned type requirements, the first argument to π is always a set of XML template values. The function is defined by

$$\pi(A, g, B) = \bigcup_{xml \in A, b \in B} \{\overline{\pi}(xml, g, b)\}$$

where $\overline{\pi}$ is defined by induction in the XML template according to the definition in Section 2.3:

$$\overline{\pi}(xml, g, b) = \begin{cases} str & \text{if } xml = str \\ \texttt{<}name\ \overline{\pi}(attr, g, b)\texttt{>} & \\ \quad \overline{\pi}(xml', g, b)\ \texttt{</}name\texttt{>} & \text{if } xml = \texttt{<}name\ attr\texttt{>}\ xml'\ \texttt{</}name\texttt{>} \\ b & \text{if } xml = \texttt{<[}g\texttt{]>} \\ \texttt{<[}h\texttt{]>} & \text{if } xml = \texttt{<[}h\texttt{]>} \text{ and } h \neq g \\ \overline{\pi}(xml_1, g, b)\ \overline{\pi}(xml_2, g, b) & \text{if } xml = xml_1\ xml_2 \end{cases}$$

$$\overline{\pi}(attr, g, b) = \begin{cases} \epsilon & \text{if } attr = \epsilon \\ name\texttt{="}str\texttt{"} & \text{if } attr = name\texttt{="}str\texttt{"} \\ name\texttt{="}b\texttt{"} & \text{if } attr = name\texttt{=[}g\texttt{]} \text{ and } b \in STRING \\ name\texttt{=[}h\texttt{]} & \text{if } attr = name\texttt{=[}h\texttt{]} \\ & \quad \text{and } (h \neq g \vee b \in XML) \\ \overline{\pi}(attr_1, g, b)\ \overline{\pi}(attr_2, g, b) & \text{if } attr = attr_1\ attr_2 \end{cases}$$

This defines plug as a substitution operation where template gaps may contain both strings and templates and string gaps may contain only strings.

For each flow edge from ℓ to ℓ' labeled with a variable x we add the following constraint

$$\mathcal{E}_{\ell}(x) \subseteq \mathcal{E}_{\ell'}(x)$$

to model that the value of x may flow from ℓ to ℓ'.

We now define the semantics of the flow graph as the least solution to the constraint system. This is well-defined because all the constraints are continuous. Note that the environment lattice is not finite, but we do not need to actually compute the solution for any concrete flow graph.

In the following section we specify a translation from JWIG programs into flow graphs. In this translation, each `show` statement, plug expression, and `receive` expression occurring in the JWIG program has a corresponding node in the flow graph. Also, each operand of a `show` or plug operation has a corresponding variable in the flow graph. Correctness of such a translation is expressed as two requirements: (1) let *env* be the least solution to the flow graph constraint system. If we observe the store of a JWIG program at either a `show` or a plug operation during some execution, then the value of each operand is contained in $env_{\ell}(x)$ where ℓ is the node corresponding to the JWIG operation and x is the variable corresponding to the operand. (2) If some session thread of an execution of the JWIG program passes a `show` statement and later a `receive` expression without passing any other `show` statement in between, then the flow graph contains a receive edge from the node corresponding to the `receive` expression to the node corresponding to the `show` statement.

3.3 From JWIG Programs to Flow Graphs

The flow graph must capture the flow of string and XML values in the original JWIG program. Compared to `<bigwig>` and the flow graph construction in [Brabrand et al. 2001] this is substantially more involved due to the many language features of Java. We divide this data flow into three categories: (1) per-method flow of data in local variables, (2) data flow to and from field variables, and (3) flow of argument and return values for method invocations. Since local variables are guaranteed to be private to each method invocation, we model the first kind of data flow in a control-flow-sensitive manner. With field variables, this cannot be done because they may be accessed by other concurrently running session threads, and because we are not able to distinguish between different instantiations of the same class. The second kind of data flow is therefore modeled in a control-flow-insensitive manner.

The translation ignores variables whose type is not `String`, `XML`, or an array of any dimension of these two. For each of the two analyzed types, a unique flow graph *pool variable* is created for representing all the values of that type that cannot be tracked by the analysis. Pooled values include those assigned to and from `Object` variables and arrays, and arguments and results of methods outside the application classes. We add an assignment of `anystring` to the pool variable of type STRING to be maximally pessimistic about the string operations in the nonapplication classes. Something similar is not done for the XML type since we have assumed that XML values are produced only inside the analyzed classes.

In addition to capturing data flow, the flow graph must contain receive edges that reflect the correspondence between show and receive operations in the JWIG program. This requires knowledge of the control-flow in addition to the data flow.

Before the actual translation into flow graphs begin, each code gap is converted to a template gap with a unique name, and the code inside the gap is moved to a new method in the service class.

The whole translation of JWIG programs into flow graphs proceeds through a sequence of phases, as described in the following subsections. Since JWIG includes the entire Java language we attempt to give a brief overview of the translation rather than explain all its details. We claim that this translation is correct according to the definition in the previous section; however it is beyond the scope of this article to state the proof.

1. Individual methods. In the first phase, each method in the application classes is translated individually into a flow graph. Each statement produces one or more nodes, and edges are added to reflect the control-flow inside the method. Each edge is labeled with all local variables of the method. Nested expressions are flattened using fresh local variables and assignments. The special JWIG operations are translated into the corresponding flow graph statement or expression, and all irrelevant operations are modeled with `nop` nodes. Each `receive` expression is translated into two nodes: a `receive` node and an assignment of `anystring`, since we need to model the locations of these operations but have no knowledge of the values being received. The control structures, `if`, `switch`, `for`, etc., are modeled with `nop` nodes and flow edges, while ignoring the results of the branch conditions. XML casts are translated into XML template constants. This is sound since we have assumed that all casts are valid. Figure 8 shows the flow graph for the `main` method of the example JWIG program in Section 2.7.

2. Code gaps. As mentioned, each code gap has been converted into a template gap whose name uniquely identifies the method containing its code. Before every

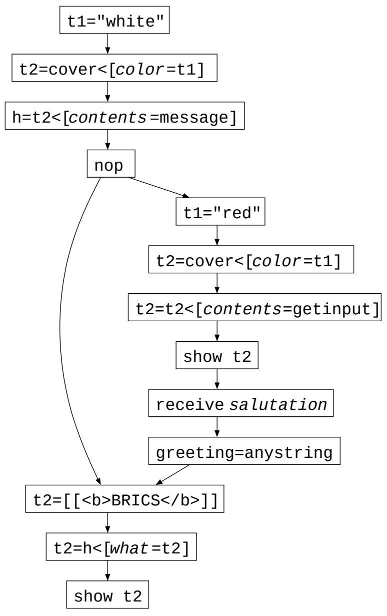


Fig. 8. Flow graph for the `main` method after Phase 1. All edges are here implicitly labeled with the set of variables {`h`,`t1`,`t2`}.

`show` statement, a sequence of method calls and plug operations is inserted to ensure that all code gaps that occur in the program are executed and that their results are inserted. To handle code gaps that generate templates that themselves contain code gaps, an extra flow edge is added from the end of the plug sequence to the start. The analysis used in `<bigwig>` [Brabrand et al. 2001] does not support code gaps.

3. Method invocations. The methods are combined *monovariantly*: each method is represented only once in the flow graph for the whole program. This means that the subsequent analyses that build on the flow graphs also are monovariant. To estimate which methods can be called at each invocation site, a call graph of the JWIG program is constructed using a *class hierarchy analysis* (CHA) [Dean et al. 1995; Sundaresan et al. 2000]. This gives us for each invocation site a set of possible target methods. Of course, other call graph analyses could be applied instead, but CHA has proven to be fast and sufficiently precise for these purposes. This implies that XML document construction usually is programmed in a simple style that does not rely on method overriding. Call graph analyses always work under a closed-world assumption, which is given by the well-formedness assumptions in Section 2.6

For each method invocation, we need to transfer the arguments and the return value, and to add flow edges to and from the possible target methods. The caller method uses a set of special local variables for collecting the arguments and the return value. We first insert a chain of assignments to these caller argument variables. Then we branch for each possible target method and add a chain of assignments from the caller argument variables to the target method parameters, followed by a

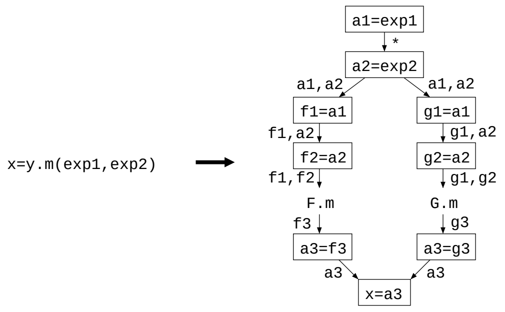


Fig. 9. Modeling method invocations. Assuming that the invocation of `m` in the expression on the left may lead to the classes `F` or `G`, the flow graph on the right is generated where `F.m` and `G.m` are the flow graphs for the target methods. First, the actual parameters are evaluated, then they are passed to the formal parameters for each method and the method bodies are processed, and finally, the return value is collected and the flow is merged. The `*` label denotes all local variables in the caller method.

flow edge to the target method entry point. Similarly, we add flow edges from the method exit points and transfer the return value via a caller result variable. For target methods in nonapplication classes, we use the pool variables in place of the argument and return value variables of the target method.

Figure 9 shows an example of a flow graph for a method invocation where the CHA has determined that there are two possible targets.

4. Exceptions. For every `try-catch-finally` construct, we add edges from all nodes corresponding to statements in the `try` block to the entry node of the `catch` block. These edges are labeled with all local variables of the method. This ensures that the data flow for local variables is modeled correctly. Adding edges from all nodes of the `try` blocks may seem to cause imprecision. A more complex analysis would only add edges from the nodes that correspond to statements that actually may throw exceptions that are caught by the `catch` block. However, our simple approach appears to be sufficiently precise in practice. The reason is that we only consider the flow of XML values and that exceptions are not generally used as a control structure for this purpose.

In order to be able to set up the receive edges in a later phase, we also need to capture the interprocedural control-flow of exceptions. For this purpose, we add a special *drain node* for each method. For each statement, we add a flow edge with an empty label to the drain node of its method. This represents the control-flow for uncaught exceptions within each method. This flow may subsequently lead either to drain nodes for caller methods or to `catch` blocks. To model this, we use the CHA information: for each target method of an invocation site, an edge is added from the drain node of the target method to the drain node of the method containing the invocation site. If the invocation site is covered by an exception handler within the method, an extra edge is added from the drain node of the target method to the entry of the handler.

5. Show and receive operations. The preceding phases have set up flow edges representing all possible control-flow in the program. This means that we at this point have sufficient information to create the receive edges, based on the correspondence between `show` and `receive` nodes.

For each `receive` statement, we want to infer from which `show` statements the control-flow may reach this `receive` without passing another `show` statement in between. We treat the entry points of `main` methods of session classes as if they were `show` statements that show a document with a single form containing no input fields. This models the fact that no input fields can be read with `receive` until a document has been shown.

A simple graph reachability computation on the flow graph would be too imprecise for this purpose. Instead we employ a reachability analysis that takes into account the correct balancing of method calls and returns. A path in the flow graph corresponding to possible control-flow must have no mismatched call/return pairs, that is, a call from one invocation site matched by a return to another invocation site. It may, however, have unmatched returns before the first call and unmatched calls after the last return. This requirement can be precisely captured using context-free reachability, as described in [Reps 1998].

We first decorate each edge in the flow graph with an *reachability tag*, describing the reachability properties of the edge. These tags are put as follows:

E	(Entry)	on all outgoing edges from an entry node of a method;
S	(Show)	on all outgoing edges from a `show` node;
C_n	(Call)	on the edge to the target method entry point ending the argument assignment chain of a method call (the subscript n is a unique identifier of the invocation site);
R_n	(Return)	on every return edge to the invocation site identified by n, and on every edge from the drain node of a method callable by this invocation site to an exception handler in, or the drain node of, the method containing this invocation site; and
N	(Normal)	on all other edges.

Note that edges from drain nodes can have multiple tags, if some method can be called from multiple invocation sites within the same method. These are treated as separate edges, each with one tag, in the following.

The reachability algorithm continuously adds edges to the graph according to a set of rules, corresponding to the grammar productions in the context-free reachability formulation. All rules have the form of a two-edge configuration that causes a shortcut edge to be added. These are shown in the following list:

$$
\begin{array}{lcl}
a \xrightarrow{E} b \xrightarrow{N} c & \triangleright & a \xrightarrow{E} c \\
a \xrightarrow{E} b \xrightarrow{R_n} c & \triangleright & a \xrightarrow{R_n} c \text{ (for any } n) \\
a \xrightarrow{C_n} b \xrightarrow{R_m} c & \triangleright & a \xrightarrow{N} c \text{ (if } n = m) \\
a \xrightarrow{S} b \xrightarrow{N} c & \triangleright & a \xrightarrow{S} c \\
a \xrightarrow{S} b \xrightarrow{R_n} c & \triangleright & a \xrightarrow{S} c \text{ (for any } n) \\
a \xrightarrow{S} b \xrightarrow{C_n} c & \triangleright & a \xrightarrow{T} c \text{ (for any } n) \\
a \xrightarrow{T} b \xrightarrow{C_n} c & \triangleright & a \xrightarrow{T} c \text{ (for any } n) \\
a \xrightarrow{T} b \xrightarrow{E} c & \triangleright & a \xrightarrow{T} c
\end{array}
$$

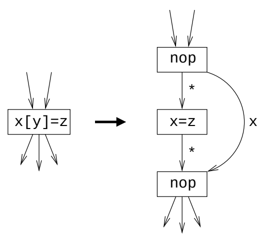


Fig. 10. Shortcutting array updates. Assignments into arrays are modeled using weak update where all entries are merged.

All of the above rules preserve the following invariants of the reachability tags:

$a \xrightarrow{N} b$	a is directly followed by b or is connected to b through a matching C_n/R_n pair.
$a \xrightarrow{E} b$	b is reachable from the method entry at a with only balanced calls/returns.
$a \xrightarrow{C_n} b$	$a \to b$ is a call edge from invocation site n.
$a \xrightarrow{R_n} b$	The return to invocation site n at b or matching exception arriving at b is reachable from a with only balanced calls/returns.
$a \xrightarrow{S} b$	b is reachable from the show statement at a with no unmatched calls.
$a \xrightarrow{T} b$	b is reachable from the show statement at a with one or more unmatched calls after the last return.

The algorithm terminates when no more shortcut edges can be added. Finally, for every `show` node a and `receive` node b, a receive edge is added from b to a if there exists an edge $a \xrightarrow{S} b$ or $a \xrightarrow{T} b$.

We claim that this analysis is sound; that is, for any `show` node a and `receive` node b, if the `receive` statement at b can be reached from the `show` statement at a without passing any other `show` statements, then we have added a receive edge from a to b. It is beyond the scope of this article to state the proof.

6. Arrays. Array variables are translated into variables of their base type. An array is treated like a single entity whose possible values are the union of of its entries. Construction of arrays using `new` is modeled with `null` values to reflect that they are initially empty.

An assignment to an array entry is modeled using weak updating [Chase et al. 1990] where the existing values of the array are merged with the new value. This is done by inserting two `nop` nodes around the assignment and adding an edge bypassing it labeled by the updated variable. This process is shown in Figure 10.

When one array variable is assigned to another, these variables become aliases. Such aliased variables are joined into one variable. This variable will be treated as a field variable and handled as described below, if at least one of its original variables was a field variable. This joining is similar to the technique used in [Sundaresan et al. 2000].

7. Field variables. As mentioned, we model the use of field variables in a flow-insensitive manner where all instances of a given class are merged. This is done for each field simply by adding flow edges labeled by the name of the field from all its definitions to all its uses. To avoid constructing a quadratic number of edges, we add a dummy "x=x" node to collect the definitions and the uses for each variable x.

In `<bigwig>`, a simpler and more restrictive approach was chosen: all global string variables were modeled with `anystring`, and for the global HTML variables, which correspond to the XML field variables in JWIG, the initializer expressions would dictate the types [Brabrand et al. 2001].

8. Graph simplification. Finally, we perform some reductions of the flow graph. This is not necessary for correctness or precision of the subsequent analyses, but as we show in Section 6.2, it substantially decreases the time and space requirements.

First, we remove all code that is unreachable from session entry points according to the flow edges. We ignore edges that originate from method returns since these edges do not contribute to the reachability.

Using a standard reaching definitions analysis on the flow graph [Aho et al. 1986; Nielson et al. 1999], we then find for each assignment all possible uses of that definition. This gives us a set of pairs of nodes where the first node is an assignment to some variable and the second node contains an expression which uses that variable. Once this information is obtained, we remove every flow edge and `nop` node in the graph, and then add new flow edges corresponding to the definition-use pairs. Each new edge is labeled with the single variable of the pair. Finally, a copy propagation optimization is performed to compress chains of copying statements [Aho et al. 1986].

These transformations all preserve the data flow. In the resulting flow graphs, there are no `nop` nodes and all edges are labeled with a single variable, which is crucial for the performance of the subsequent analyses.

This construction of flow graphs for JWIG programs is correct in the sense defined in Section 3.2, both with and without the simplification phase.

3.4 Complexity

During the construction of the flow graph, we have performed two forward data-flow analyses on the intermediate graphs: one for setting up the receive edges in Phase 5 and the other for the reaching definitions analysis in Phase 8. In the following sections, we will describe two more forward data-flow analyses on flow graphs. To bound the worst-case time requirements for all these analyses, we make some general observations. By implementing the analyses using a standard work-list algorithm rather than the chaotic iteration algorithm [Nielson et al. 1999], the time can be bound by

$$O\left(t \cdot \sum_{m \in nodes} |var(m)| \cdot h \cdot \sum_{m' \in succ(m)} |var(m')|\right)$$

where

—t is the maximum cost of computing one binary least-upper-bound operation or one transfer function for a single variable;

—*nodes* is the set of flow-graph nodes;

—$var(m)$ denotes the union of the labels of edges incident to the node m;

—h is the height of the lattice for a single variable; and

—$succ(m)$ is the set of successor nodes of m.

For each node m, an environment associates two lattice elements to each variable v in $var(m)$, one for the entry label and one for the exit label. Each can change at most h times. Because of the work-list, each change for an exit label can result in at most $\sum_{m' \in succ(m)} |var(m')|$ binary least-upper-bound operations and transfer function computations, that is, one for each variable in each successor node, without any other environment changes for exit labels.

Phases 1–4 create at most $O(n^2)$ flow-graph nodes and $O(n^2)$ edges where n is the textual size of the program. The reason for the quadratic increase in the number of nodes is the encoding of argument transferring for method invocations in Phase 3 and the encoding of code gap execution in Phase 2.

All edges created by the receive edge analysis in Phase 5 originate from either a method entry, an invocation site, or a `show` statement, of which there are totally $O(n)$. R_n edges with different n will always go to different nodes, so only a constant number of edges can be created between any two nodes. Thus, at most $O(n^3)$ edges are created. The algorithm is worklist based, so each new edge incurs at most $O(n^2)$ edge comparisons. Therefore, this analysis runs in time $O(n^5)$.

After Phase 5, the control-flow information is no longer needed, so all edges with empty label can be removed, since they have no influence on the subsequent phases. This includes all edges added in Phase 5. Furthermore, Phases 6 and 7 only produce $O(n)$ edges, so the final number of edges after Phases 1–7 is $O(n^2)$.

The complexity of the reaching definitions analysis in Phase 8 can be bound by the formula above. There are $O(n)$ variables, the lattice height is $O(n)$, and the time t is $O(n)$. Again, since there are $O(n^2)$ edges, $\sum_{m \in nodes} |succ(m)|$ is $O(n^2)$. Together, we get that this analysis runs in time $O(n^6)$. Since the other phases of the flow-graph construction are linear in the size of the flow graph, the total construction of the flow graph of a given JWIG program requires worst-case $O(n^6)$ time. As shown in Section 6.2, this bound is rarely encountered in practice.

After the simplification phase, an extra property is satisfied: since all flow edges are definition-use edges, $|var(m)|$ is $O(1)$ for all nodes m. Since the flow graph still contains only $O(n^2)$ nodes and edges, the formula above then reduces to

$$O(n^2 \cdot h \cdot t)$$

This will be used later to estimate the complexities of the remaining analyses.

3.5 Flow Graph for the Example

Figure 11 shows the flow graph that is generated for the JWIG program from Section 2.7. The left part of the graph corresponds to the `main` method, the top right part is the initialization of the field variables, and the bottom right part corresponds to the code in the code gap. The thin edges are flow edges, and the single thick edge is a receive edge. The `show` node in the top left corresponds

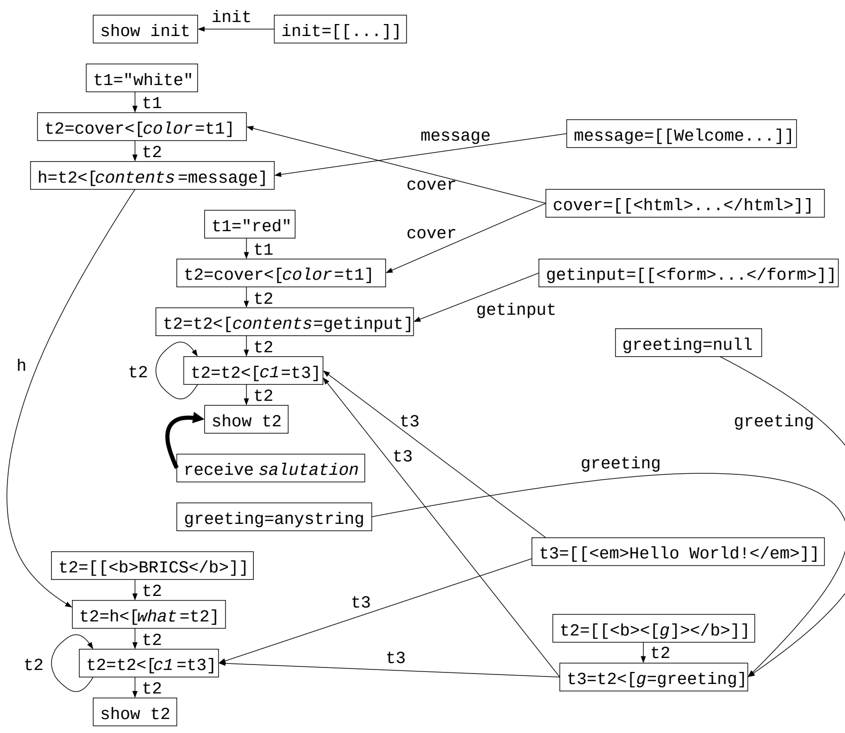


Fig. 11. Flow graph for the JWIG example program.

to the entry point of the session. The `init` template is a simple valid XHTML document with a form that contains no fields. This models the fact that no input values are receivable when session threads are initiated. Note that edges in the part corresponding to the `main` method have changed compared to Figure 8 because of the graph simplification phase.

4. SUMMARY GRAPH ANALYSIS

To statically verify that a given JWIG program will never throw any of the special JWIG exceptions, we perform a *summary graph analysis* based on the flow graph which contains all the information we need from the original program. Summary graphs model how templates are constructed and used at runtime. This analysis depends on a preliminary *string analysis* that for each string expression finds a regular language that approximates the set of strings it may evaluate to. For each analysis we define a lattice expressing an abstraction of the data values along with a corresponding abstract semantics of the expressions and statements, and then apply standard data-flow analysis techniques to find the least solutions. We choose to provide the technical details for two reasons: first, the development of summary graphs was the main challenge in obtaining a useful tool and, second, we believe that summary graphs by themselves may be useful for analyzing XML manipulations in other contexts.

4.1 String Analysis

Given a flow graph of a JWIG program, we must statically model which strings are constructed and used at runtime. In [Brabrand et al. 2001] the corresponding analysis is mixed into the summary graph analysis. Separating these analyses leads to a simpler specification and implementation without damaging the analysis precision. We describe here a rather simple analysis which is adequate for all our benchmarks. However, it should be clear that a more precise string analysis capturing relevant string operations easily could be applied instead, as explained in Section 7.

We first define a *string environment* lattice:

$$SE = Y \rightarrow REG$$

where Y is the set of string variables that occur in the program and REG is the family of regular languages over the Unicode alphabet. We choose regular languages for modeling string sets because they fit elegantly into the validity analysis algorithm in Section 5.5. The ordering on REG is language inclusion and SE inherits this ordering pointwise. We compute an element of this lattice for every program point with a forward data-flow analysis using standard techniques, such as the monotone frameworks of [Nielson et al. 1999; Kam and Ullman 1977]: For every statement that can appear in the flow graph we define a monotone transfer function $SE \rightarrow SE$ and then compute the least fixed point by iteration. First, for each flow-graph string expression we define its abstract denotation by extending every environment map $\Sigma \in SE$ from variables to string expressions, $\widehat{\Sigma} : EXP_{STRING} \rightarrow REG$:

$$\widehat{\Sigma}(exp) = \begin{cases} \Sigma(x) & \text{if } exp = x, \\ \{str\} & \text{if } exp = \texttt{"}str\texttt{"} \\ U^* & \text{if } exp = \textbf{\texttt{anystring}} \\ \emptyset & \text{if } exp = \textbf{\texttt{null}} \end{cases}$$

where U denotes the Unicode alphabet.

For every string assignment statement x `=` exp`;` the transfer function is defined by

$$\Sigma \mapsto \Sigma[x \mapsto \widehat{\Sigma}(exp)]$$

that is, the string environment is updated for x to the environment value of exp. Clearly, this is a monotone operation. For all other statements the transfer function is the identity function, since they do not modify any string variables. The lattice is not finite, but by observing that the only languages that occur are either total or subsets of the finitely many string constants that appear in the program, termination of the fixed point iteration is ensured.

The worst-case complexity of this analysis can be estimated by the formula from the previous section. By the observation above, we only use a part of the lattice. This part has height $O(n)$ since there are at most $O(n)$ string constants in the program. The time t for performing a least-upper-bound or transfer function computation is $O(n)$. Thus, this particularly simple string analysis runs in worst-case time $O(n^4)$ where n is the size of the original JWIG program.

The result of this analysis is for each program point ℓ a map: $string_\ell : Y \to REG$. This analysis is correct in the following sense: for any execution of the program, any program point ℓ, and any string variable x, the regular language $string_\ell(x)$ will always contain the value of x at ℓ. That is, the analysis result is a conservative upper approximation of the string flow.

We are currently working on a more advanced string analysis that models concatenations and other string operations more precisely. This cannot be expressed by a straight application of the monotone framework since the lattice of regular languages is not finite. Instead, we model each variable by a nonterminal in a context-free grammar, which is subsequently coarsened into a regular language [Christensen et al. 2003].

4.2 Summary Graphs

As the string analysis, the summary graph analysis fits into standard data-flow frameworks, but it uses a significantly more complex lattice which we define in the following. Let X, G, and N be, respectively, the sets of template variables, gap names, and template labels that occur in the program. A *summary graph* SG is a finite representation of a set of XML documents defined as

$$SG = (R, T, S, P)$$

where

$R \subseteq N$ is a set of *root nodes*,
$T \subseteq N \times G \times N$ is a set of *template edges*,
$S : N \times G \to REG$ is a *string edge* map, and
$P : G \to 2^N \times \Gamma \times \Gamma$ is a *gap presence* map.

Here $\Gamma = 2^{\{\mathsf{OPEN},\mathsf{CLOSED}\}}$ is the *gap presence lattice* whose ordering is set inclusion. Intuitively, the language $\mathcal{L}(SG)$ of a summary graph SG is the set of XML documents that can be obtained by unfolding its templates, starting from a root node and plugging templates and strings into gaps according to the edges. Assume that $t : N \to xml$ maps every template label to the associated template constant. The presence of a template edge $(n_1, g, n_2) \in T$ informally means that the $t(n_2)$ template may be plugged into the g gaps in $t(n_1)$, and a string edge $S(n, g) = L$ means that every string in the regular language L may be plugged into the g gaps in $t(n)$.

The gap presence map, P, specifies for each gap name g which template constants may contain open g gaps reachable from a root and whether g gaps may or must appear somewhere in the unfolding of the graph, either as template gaps or as attribute gaps. The first component of $P(g)$ denotes the set of template constants with potentially open g gaps, and the second and third components describe the presence of template gaps and attribute gaps, respectively. Given such a triple, $P(g)$, we let $nodes(P(g))$ denote the first component. For the other components, the value OPEN means that the gaps may be present, and CLOSED means that they may be absent. Recall from Section 2.3 that, at runtime, if a document is shown with open template gaps, these are treated as empty strings. For open attribute gaps, the entire attribute is removed. We need the gap presence information in the summary graphs to (1) determine where edges should be added when modeling plug operations, (2) model the removal of gaps that remain open when a document is shown, and (3) detect that plug operations may fail because the specified gaps have already been closed.

This unfolding of summary graphs is explained more precisely with the following formalization:

$$\mathit{unfold}(SG) = \{d \in \mathit{XML} \mid \exists r \in R : SG, r \vdash t(r) \Rightarrow d \;\text{ where } SG = (R, T, S, P)\}$$

The *unfolding relation*, $\Rightarrow$, is defined by induction in the structure of the summary graph. We use inference rules as a convenient notation for expressing these mutually dependent relations. For the parts that do not involve gaps the definition is a simple recursive traversal:

$$\frac{}{SG, n \vdash \mathit{str} \Rightarrow \mathit{str}}$$

$$\frac{SG, n \vdash \mathit{xml}_1 \Rightarrow \mathit{xml}_1' \quad SG, n \vdash \mathit{xml}_2 \Rightarrow \mathit{xml}_2'}{SG, n \vdash \mathit{xml}_1\ \mathit{xml}_2 \Rightarrow \mathit{xml}_1'\ \mathit{xml}_2'}$$

$$\frac{SG, n \vdash \mathit{atts} \Rightarrow \mathit{atts}' \quad SG, n \vdash \mathit{xml} \Rightarrow \mathit{xml}'}{SG, n \vdash \texttt{<}\mathit{name}\ \mathit{atts}\texttt{>}\ \mathit{xml}\ \texttt{</}\mathit{name}\texttt{>} \Rightarrow \texttt{<}\mathit{name}\ \mathit{atts}'\texttt{>}\ \mathit{xml}'\ \texttt{</}\mathit{name}\texttt{>}}$$

$$\frac{}{SG, n \vdash \epsilon \Rightarrow \epsilon}$$

$$\frac{}{SG, n \vdash \mathit{name}\texttt{="}\mathit{str}\texttt{"} \Rightarrow \mathit{name}\texttt{="}\mathit{str}\texttt{"}}$$

$$\frac{SG, n \vdash \mathit{atts}_1 \Rightarrow \mathit{atts}_1' \quad SG, n \vdash \mathit{atts}_2 \Rightarrow \mathit{atts}_2'}{SG, n \vdash \mathit{atts}_1\ \mathit{atts}_2 \Rightarrow \mathit{atts}_1'\ \mathit{atts}_2'}$$

There is no unfolding for code gaps since they have already been reduced to template gaps in the flow graph construction. For template gaps we unfold according to the string edges and template edges and check whether the gaps may be open:

$$\frac{\mathit{str} \in S(n, g)}{(R, T, S, P), n \vdash \texttt{<[}g\texttt{]>} \Rightarrow \mathit{str}}$$

$$\frac{(n, g, m) \in T \quad (R, T, S, P), m \vdash t(m) \Rightarrow \mathit{xml}}{(R, T, S, P), n \vdash \texttt{<[}g\texttt{]>} \Rightarrow \mathit{xml}}$$

$$\frac{n \in \mathit{nodes}(P(g))}{(R, T, S, P), n \vdash \texttt{<[}g\texttt{]>} \Rightarrow \texttt{<[}g\texttt{]>}}$$

For attribute gaps we unfold according to the string edges, and check whether the gaps may be open:

$$\frac{\mathit{str} \in S(n, g)}{(R, T, S, P), n \vdash \mathit{name}\texttt{=[}g\texttt{]} \Rightarrow \mathit{name}\texttt{="}\mathit{str}\texttt{"}}$$

$$\frac{n \in \mathit{nodes}(P(g))}{(R, T, S, P), n \vdash \mathit{name}\texttt{=[}g\texttt{]} \Rightarrow \mathit{name}\texttt{=[}g\texttt{]}}$$

The following function, *close*, is used on the unfolded templates to plug the empty string into remaining template gaps and remove all attributes with gap values:

$$close(xml) = \begin{cases} \texttt{<}name\ close(atts)\texttt{>} & \\ \quad close(xml')\ \texttt{</}name\texttt{>} & \text{if } xml = \texttt{<}name\ atts\texttt{>}\ xml'\ \texttt{</}name\texttt{>} \\ \epsilon & \text{if } xml = \texttt{<[}g\texttt{]>} \\ close(xml_1)\ close(xml_2) & \text{if } xml = xml_1\ xml_2 \\ xml & \text{otherwise} \end{cases}$$

$$close(atts) = \begin{cases} close(atts_1)\ close(atts_2) & \text{if } atts = atts_1\ atts_2 \\ \epsilon & \text{if } atts = name\texttt{=[}g\texttt{]} \\ atts & \text{otherwise} \end{cases}$$

We now define the language of a summary graph by

$$\mathcal{L}(SG) = \{close(d) \in XML \mid d \in unfold(SG)\}$$

Let $\mathcal{G}$ be the set of all summary graphs. This set is a lattice where the ordering is defined as one would expect:

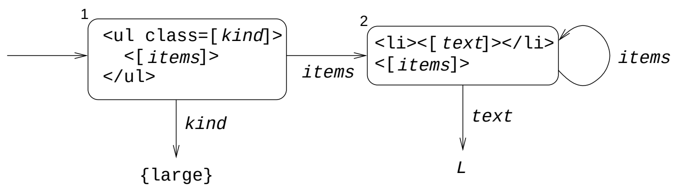
$$\begin{gathered}(R_1, T_1, S_1, P_1) \sqsubseteq (R_2, T_2, S_2, P_2) \quad \Leftrightarrow \\ R_1 \subseteq R_2 \ \wedge \ T_1 \subseteq T_2 \ \wedge \\ \forall n \in N, g \in G : S_1(n, g) \subseteq S_2(n, g) \ \wedge \ P_1(g) \sqsubseteq P_2(g)\end{gathered}$$

where the ordering on gap presence maps is defined by componentwise set inclusion. This ordering respects language inclusion: if $SG_1 \sqsubseteq SG_2$, then $\mathcal{L}(SG_1) \subseteq \mathcal{L}(SG_2)$.

Compared to the summary graphs in [Brabrand et al. 2001] this definition differs in the following ways: first of all, the gap presence map is added. The old algorithm worked under the assumption that all incoming branches to join points in the flow graph would agree on which gaps were open. This was achieved using a simple preliminary program transformation that would convert the "implicit ϵ-plugs" of `<bigwig>` [Brabrand 2000] into explicit ones using the information from the DynDoc type system [Sandholm and Schwartzbach 2000]. Since JWIG does not inherit this implicit-plug feature from the `<bigwig>` design nor uses a DynDoc-like type system we have added the gap presence map. This map contains the information from the "gap track analysis" in [Brabrand et al. 2001], but in addition to finding gaps that *may* be open it also tracks *must* information which we need to verify the use of plug operations later.

Second, the present definition is more flexible in that it allows strings to be plugged into template gaps. In [Brabrand et al. 2001], template gaps were completely separated from attribute gaps. Third, we generalize the flat string lattice to full regular languages, allowing us to potentially capture many string operations.

Figure 12 shows an example summary graph consisting of two nodes, a single template edge, and two string edges. The language of this summary graph is the set of XML documents that consist of `ul` elements with a `class="large"` attribute and zero or more `li` items containing some text from the language L. Note that the *`items`* gap in the root template may be OPEN according to the gap presence map, so the empty template may be plugged in here, corresponding to the case where the list is empty.



Gap presence:		
kind	↦	(Ø,{CLOSED},{CLOSED})
items	↦	({1,2},{OPEN}, {CLOSED})
text	↦	{Ø,{CLOSED},{CLOSED})

Fig. 12. A summary graph whose language is a set of XML documents, each containing a `ul` list with zero or more text items and a `class` attribute. The node on the left is a root, and L denotes some set of strings.

Note that our analysis is *monovariant*, in the sense that each template constant is only represented once. It is possible to perform a more expensive analysis that duplicates summary graph nodes according to some criteria, but we have not yet encountered the need. On the other hand, our analysis is *polyvariant* in XML element constructors, since these are analyzed separately for each occurrence in the templates.

The summary graph abstraction has evolved through experiments during our previous work on `<bigwig>`. We claim that it is in a finely tuned balance between expressibility and complexity. In [Christensen et al. 2002], we give a constructive proof that summary graphs have essentially the same expressive power as the regular expression types of XDuce [Hosoya and Pierce 2000], in the sense that they characterize the same family of XML languages—if disregarding restrictions on character data and attributes, which are not supported by XDuce. However, summary graphs contain extra structure, for instance, by the gap presence maps, which is required during analysis to model the gap plugging mechanism. Summary graphs contain the structure and expressiveness to capture the intricacies of normal control-flow in programs and are yet sufficiently tractable to allow efficient analysis.

4.3 Constructing Summary Graphs

At every program point ℓ in the flow graph, each template variable $x \in X$ is associated a summary graph, as modeled by the *summary graph environment* lattice:

$$SGE = X \to \mathcal{G}$$

which inherits its structure pointwise from $\mathcal{G}$. We compute an element of the lattice for every program point using yet another forward data-flow analysis. Let $[\![\ xml\ ]\!]_n$ mean the template constant labeled n, and let $tgaps(n)$ and $agaps(n)$ be the sets of template gap names and attribute gap names, respectively, that occur in the template constant labeled n. Given an environment lattice element $\Delta \in SGE$

we define an abstract denotation for template expressions, $\widehat{\Delta} : \mathit{EXP}_{\mathit{XML}} \rightarrow \mathcal{G}$:

$$\widehat{\Delta}(\mathit{exp}) = \begin{cases} \Delta(x) & \text{if } \mathit{exp} = x \\ \mathit{const}(\mathit{tgaps}(n), \mathit{agaps}(n), n) & \text{if } \mathit{exp} = \texttt{[[} \mathit{xml} \texttt{]]}_n \\ \mathit{tplug}(\Delta(x), g, \Delta(y)) & \text{if } \mathit{exp} = x \texttt{ <[} g \texttt{ = } y \texttt{]} \\ & \quad \text{and } y \text{ has type } \mathsf{XML} \\ \mathit{splug}(\Delta(x), g, \mathit{string}_\ell(y)) & \text{if } \mathit{exp} = x \texttt{ <[} g \texttt{ = } y \texttt{]} \\ & \quad \text{and } y \text{ has type } \mathsf{STRING} \\ (\emptyset, \emptyset, \lambda(m,h).\emptyset, \lambda h.(\emptyset,\emptyset,\emptyset)) & \text{if } \mathit{exp} = \texttt{null} \end{cases}$$

where the auxiliary functions are

$$\begin{aligned} \mathit{const}(A,B,n) = (&\{n\}, \emptyset, \lambda(m,h).\emptyset, \\ &\lambda h.(\mathsf{if}\ h \in A \cup B\ \mathsf{then}\ \{n\}\ \mathsf{else}\ \emptyset, \\ &\quad \mathsf{if}\ h \in A\ \mathsf{then}\ \{\mathsf{OPEN}\}\ \mathsf{else}\ \{\mathsf{CLOSED}\}, \\ &\quad \mathsf{if}\ h \in B\ \mathsf{then}\ \{\mathsf{OPEN}\}\ \mathsf{else}\ \{\mathsf{CLOSED}\})) \end{aligned}$$

$$\begin{aligned} &\mathit{tplug}((R_1,T_1,S_1,P_1), g, (R_2,T_2,S_2,P_2)) = \\ &\quad (R_1, \\ &\quad T_1 \cup T_2 \cup \{(n,g,m) \mid n \in \mathit{nodes}(P_1(g)) \ \wedge\ m \in R_2\}, \\ &\quad \lambda(m,h).S_1(m,h) \cup S_2(m,h), \\ &\quad \lambda h.\mathsf{if}\ h{=}g\ \mathsf{then}\ P_2(h)\ \mathsf{else}\ (p_1 \cup p_2, \mathit{merge}(t_1,t_2), \mathit{merge}(a_1,a_2))) \\ &\qquad\qquad\qquad \text{where } P_1(h) = (p_1,t_1,a_1) \text{ and } P_2(h) = (p_2,t_2,a_2) \end{aligned}$$

$$\mathit{merge}(\gamma_1,\gamma_2) = \mathsf{if}\ \gamma_1{=}\{\mathsf{OPEN}\} \vee \gamma_2{=}\{\mathsf{OPEN}\}\ \mathsf{then}\ \{\mathsf{OPEN}\}\ \mathsf{else}\ \gamma_1 \cup \gamma_2$$

$$\begin{aligned} &\mathit{splug}((R,T,S,P), g, L) = \\ &\quad (R, \\ &\quad T, \\ &\quad \lambda(m,h).\mathsf{if}\ h{=}g \wedge m \in \mathit{nodes}(P(h))\ \mathsf{then}\ S(m,h) \cup L\ \mathsf{else}\ S(m,h), \\ &\quad \lambda h.\mathsf{if}\ h{=}g\ \mathsf{then}\ (\emptyset, \{\mathsf{CLOSED}\}, \{\mathsf{CLOSED}\})\ \mathsf{else}\ P(h)) \end{aligned}$$

For template constants, we look up the set of gaps that appear and construct a simple summary graph with one root and no edges. The *tplug* function models plug operations where the second operand is a template expression. It finds the summary graphs for the two subexpressions and combines them as follows: The roots are those of the first graph since it represents the outermost template. The template edges become the union of those in the two graphs plus a new edge from each node that may have open gaps of the given name to each root in the second graph. The string edge sets are simply joined without adding new information. For the gaps that are plugged into, we take the gap presence information from the second graph. For the other gaps we use the *merge* function to mark gaps as "definitely open" if they are so in one of the graphs and otherwise take the least upper bound. The *splug* function models plug operations where the second operand is a string expression. It adds the set of strings obtained by the string analysis for the string expression to the appropriate string edge, and then marks the designated gaps as "definitely closed". The `null` constant is modeled by the empty set of documents. Attempts to plug or show a null template yield null dereference exceptions at runtime, and we do not wish to perform a specific null analysis.

Having defined the analysis of expressions we can now define transfer functions for the statements. As for the other data-flow analysis, only assignments are interesting. For every XML assignment statement x `=` exp`;` the transfer function is defined by

$$\Delta \mapsto \Delta[x \mapsto \widehat{\Delta}(exp)]$$

and for all other statements the transfer function is the identity function.

By inspecting the *tplug*, *merge*, and *splug* functions it is clear that the transfer function is always monotone. The lattice *SGE* is not finite, but analysis termination is ensured by the following observation: for any program, all summary graph components are finite, except *REG*. However, the string analysis produces only a finite number of regular languages, and we here use at most all possible unions of these. So, only a finite part of *SGE* is ever used.

The result of this analysis is for each program point ℓ a map:

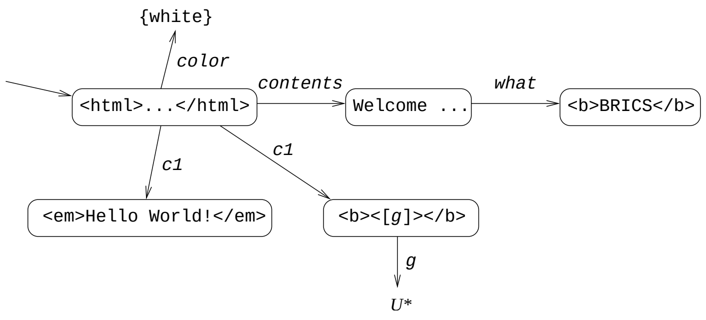
$$summary_\ell : EXP_{XML} \to \mathcal{G}$$

This analysis is conservative as the string analyses; that is, it is sound but not complete: for any execution of the program, any program point ℓ, and any XML expression *exp*, the set of XML documents $\mathcal{L}(summary_\ell(exp))$ will always contain the value of *exp* at ℓ.

The worst-case complexity of this analysis can also be estimated using the formula from Section 3.4. The lattice height is the sum of the heights of the four summary graph components. The node set N and the gap name set G both have size $O(n)$, again where n is the size of the original JWIG program. The height of the root node component is thus $O(n)$. For each template edge (n, g, n') which is created during the analysis, (n, g) determines a specific gap in a specific template in the original JWIG program. Since there can be at most $O(n)$ of these, we can at most construct $O(n^2)$ template edges. Similarly, for the string edge map, all but $O(n)$ pairs of elements from N and G are mapped to a fixed element. For the string analysis, we have argued that the height of the used part of the string lattice is $O(n)$, so the string edge component has height $O(n^2)$. Both the domain and the codomain of the gap presence map have size $O(n)$, so this component also has height $O(n^2)$. In total, the height h of the summary graph lattice is $O(n^2)$. For the same reasons, the sizes of the summary graphs that are constructed are also at most $O(n^2)$. All operations on summary graphs are linear in their sizes, so the time t for computing a summary graph operation is $O(n^2)$. Inserting this in the formula gives that the summary graph construction runs in time $O(n^6)$ in the size of the program. Note that without the flow-graph simplification phase, the formula would have given $O(n^8)$ instead of $O(n^6)$.

4.4 Summary Graphs for the Example

For the JWIG example program from Section 2.7, the following summary graph is generated for the `exit` statement in the `main` method:



Implicitly in this illustration, the gap presence map maps everything to ($\emptyset$, {CLOSED}, {CLOSED}), and the string edge map maps to the empty language by default. Because of the simple flow in the example program, the language of this summary graph is precisely the set of XML documents that may occur at runtime. In general, the summary graphs are conservative since they they may denote languages that are too large. This means that the subsequent analyses can be sound but not complete.

5. PROVIDING STATIC GUARANTEES

The remaining analyses are independent of both the original JWIG program and the flow graphs. All the relevant information is at this stage contained in the inferred summary graphs. This is a modular approach where the "front-end" and "back-end" analyses may be improved independently of each other. Also, summary graphs provide a good context for giving intuitive error messages.

5.1 Plug Analysis

We first validate *plug consistency* of the program, meaning that gaps are always present when subjected to the plug operation and that XML templates are never plugged into attribute gaps. This information is extracted from the summary graph of the template being plugged into.

In earlier work [Sandholm and Schwartzbach 2000] a similar check was performed directly on the flow graphs. Our new approach has the same precision, even though it relies exclusively on the summary graphs. Furthermore, we no longer require the flow graph to agree on the gap information for all incoming branches in the join points, as mentioned in Section 4.2.

For a specific plug operation x **<[** g **=** y **]** at a program point ℓ, consider the summary graph $\mathit{summary}_\ell(x) = (R, T, S, P)$ given by the data-flow analysis described in the previous section. Let $(p, t, a) = P(g)$. We now check consistency of the plug operation simply by inspecting that the following condition is satisfied:

$$\begin{array}{ll} t = \{\mathsf{OPEN}\} \vee a = \{\mathsf{OPEN}\} & \text{if } y \text{ has type } \mathsf{STRING}\text{, and} \\ t = \{\mathsf{OPEN}\} \wedge a = \{\mathsf{CLOSED}\} & \text{if } y \text{ has type } \mathsf{XML}. \end{array}$$

This captures the requirement that string plug operations are allowed on all gaps that are present, while template plug operations only are possible for template gaps. If a violation is detected, a precise error message can be generated: for instance, if y has type XML, t = {OPEN}, and a = {OPEN,CLOSED}, we report that, although there definitely are open template gaps of the given name, there may also be open attribute gaps, which could result in a `PlugException` at runtime.

As mentioned, the summary graphs that are constructed are conservative with respect to the actual values that appear at runtime. However, the plug analysis clearly introduces no new imprecision, that is, this analysis is both sound and complete with respect to the summary graphs: it determines that a given plug operation cannot fail if and only if for every value in $unfold(summary_\ell(x))$, the plug operation does not fail. If the plug analysis detects no errors, it is guaranteed that no `PlugException` will ever be thrown when running the program. Since the analysis merely inspects the gap presence map component of each summary graph that is associated with a plug operation, this analysis takes time $O(n)$.

5.2 Receive Analysis

We now validate *receive consistency*, meaning that `receive` and `receive[]` operations always succeed. For the single-string variant, `receive`, it must be the case that for all program executions, the last document being shown before the receive operation contained exactly one field of the given name. Also, there must have been at least one show operation between the initiation of the session thread and the receive operation. If these properties are satisfied, it is guaranteed that no `ReceiveException` will ever be thrown when running the program.

The array variant, `receive[]`, always succeeds, so technically, we do not have to analyze those operations. However, we choose to consider it as an error if we are able to detect that for a given `receive[]` operation, there are no possibility of ever receiving other that the empty array. This is to follow the spirit of Java where, for instance, it is considered a compile-time error to specify a cast operation that is guaranteed to fail for all executions.

In case the name of the field is either `submit`, `submit.x`, or `submit.y`, then we know that it comes from a submit button or image. As described in Section 2.3, exactly one value is then always generated. That is, in these cases, both `receive` and `receive[]` always succeed. For the remainder of this section, we thus assume that the field name is not one among those three.

Given a receive operation, we need to count the number of occurrences of input fields of the given name that may appear in every document sent to the client in an associated `show` operation. For a concrete XHTML/HTML document, this information is defined by Section 17.13.2 in [Raggett et al. 1999]. For a running JWIG program, a conservative approximation of the information can be extracted from the receive edges in the flow graph and the summary graphs of the associated `show` operations.

Compared to the field analysis in `<bigwig>` [Sandholm and Schwartzbach 2000], this situation differs in a number of ways: (1) the present analysis works on summary graphs rather than on flow graphs. (2) In the old analysis, the plug and receive analyses were combined. We separate them into two independent analyses without losing any precision. (3) In `<bigwig>`, a `form` is always inserted automatically around the entire document body. That precludes documents from having other forms for submitting input to other services. As described in Section 2.2, JWIG instead allows multiple forms by identifying those relevant to the session by the absence of an `action` attribute in the `form` element. (4) The notions of tuples and relations in [Sandholm and Schwartzbach 2000] are in JWIG replaced by arrays and `receive[]` operations.

Again, we will define a constraint system for computing the desired information. This information is represented by a value of the following lattice, C:

$$\begin{tikzcd}[column sep=small]
 & \ast \arrow[dl, dash] \arrow[dr, dash] & \\
0 \arrow[ddr, dash] & & 1 \arrow[d, dash] \\
 & & \diamond \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

The element 0 means that there are always zero occurrences of the field, 1 means that there is always exactly one occurrence, $*$ means that the number varies depending on the unfolding or that it is greater than one, $\diamond$ represents one or more radio buttons, and $\perp$ represents an unknown number. The constraint system applies two special monotone operators on C: $\oplus$ for addition and $\otimes$ for multiplication. These are defined as follows:

$\oplus$	$\perp$	0	$\diamond$	1	$*$
$\perp$	$\perp$	0	$\diamond$	1	$*$
0	0	0	$\diamond$	1	$*$
$\diamond$	$\diamond$	$\diamond$	$\diamond$	$*$	$*$
1	1	1	$*$	$*$	$*$
$*$	$*$	$*$	$*$	$*$	$*$

$\otimes$	$\perp$	0	$\diamond$	1	$*$
$\perp$	$\perp$	0	$\perp$	$\perp$	$\perp$
0	0	0	0	0	0
$\diamond$	$\perp$	0	$\diamond$	$\diamond$	$*$
1	$\perp$	0	$\diamond$	1	$*$
$*$	$\perp$	0	$*$	$*$	$*$

Assume that we are given a summary graph (R, T, S, P) corresponding to a specific `show` statement. Two special functions are used for extracting information about fields and gaps for an individual node in the summary graph:

$$\mathit{count} : N \to \mathit{GFP}$$

$$\mathit{allforms} : N \to 2^{\mathit{GFP}}$$

where $\mathit{GFP} = (F \to C) \times (G \to C)$ shows the number of occurrences of fields and gaps in a specific form. The *allforms* function returns a set of such values, corresponding to the various forms that may appear, and *count* counts disregarding the `form` elements:

$$\mathit{count}(n) = (\mathit{fcount}(n, t(n)), \mathit{gcount}(n, t(n)))$$

$$\mathit{allforms}(n) = \bigcup_{k \in \mathit{forms}(t(n))} \{(\mathit{fcount}(n, k), \mathit{gcount}(n, k))\}$$

where

$$
\mathit{forms}(\mathit{xml}) = \begin{cases}
\{\mathit{xml}\} & \text{if } \mathit{xml} = \texttt{<form } \mathit{atts}\texttt{>}\ \mathit{xml}'\ \texttt{</form>} \\
 & \quad \text{and } \mathit{atts} \text{ does not contain } \texttt{action} \\
\mathit{forms}(\mathit{xml}') & \text{if } \mathit{xml} = \texttt{<}\mathit{name}\ \mathit{atts}\texttt{>}\ \mathit{xml}'\ \texttt{</}\mathit{name}\texttt{>} \\
 & \quad \text{and } \mathit{name} \neq \texttt{form} \\
 & \quad \text{or } \mathit{atts} \text{ contains } \texttt{action} \\
\mathit{forms}(\mathit{xml}_1) \cup \mathit{forms}(\mathit{xml}_2) & \text{if } \mathit{xml} = \mathit{xml}_1\ \mathit{xml}_2 \\
\emptyset & \text{otherwise}
\end{cases}
$$

$$
\mathit{gcount}(n, \mathit{xml})(g) = \begin{cases}
\mathit{gcount}(n, \mathit{xml}_1)(g) & \\
\quad \oplus\, \mathit{gcount}(n, \mathit{xml}_2)(g) & \text{if } \mathit{xml} = \mathit{xml}_1\ \mathit{xml}_2 \\
\mathit{gcount}(n, \mathit{xml}')(g) & \\
\quad \oplus\, \mathit{gcount}(n, \mathit{atts})(g) & \text{if } \mathit{xml} = \texttt{<}\mathit{name}\ \mathit{atts}\texttt{>}\ \mathit{xml}'\ \texttt{</}\mathit{name}\texttt{>} \\
1 & \text{if } \mathit{xml} = \texttt{<[}g\texttt{]>} \\
 & \quad \text{and } n \notin \mathit{nodes}(P(g)) \\
* & \text{if } \mathit{xml} = \texttt{<[}g\texttt{]>} \\
 & \quad \text{and } n \in \mathit{nodes}(P(g)) \\
0 & \text{otherwise}
\end{cases}
$$

$$
\mathit{gcount}(n, \mathit{atts})(g) = \begin{cases}
\mathit{gcount}(n, \mathit{atts}_1)(g) & \\
\quad \oplus\, \mathit{gcount}(n, \mathit{atts}_2)(g) & \text{if } \mathit{atts} = \mathit{atts}_1\ \mathit{atts}_2 \\
1 & \text{if } \mathit{atts} = \texttt{<[}g\texttt{]>} \\
 & \quad \text{and } n \notin \mathit{nodes}(P(g)) \\
* & \text{if } \mathit{atts} = \texttt{<[}g\texttt{]>} \\
 & \quad \text{and } n \in \mathit{nodes}(P(g)) \\
0 & \text{otherwise}
\end{cases}
$$

$$
\mathit{fcount}(n, \mathit{xml})(f) = \begin{cases}
\mathit{fcount}(n, \mathit{xml}_1)(f) & \\
\quad \oplus\, \mathit{fcount}(n, \mathit{xml}_2)(f) & \text{if } \mathit{xml} = \mathit{xml}_1\ \mathit{xml}_2 \\
\mathit{fcount}(n, \mathit{xml}')(f) & \text{if } \mathit{xml} = \texttt{<}\mathit{name}\ \mathit{atts}\texttt{>}\ \mathit{xml}'\ \texttt{</}\mathit{name}\texttt{>} \\
 & \quad \text{and } \mathit{name} \notin \text{FIELDS} \\
\mathit{fc}(n, \mathit{atts}) & \text{if } \mathit{xml} = \texttt{<}\mathit{name}\ \mathit{atts}\texttt{>}\ \mathit{xml}'\ \texttt{</}\mathit{name}\texttt{>} \\
 & \quad \text{and } \mathit{name} \in \text{FIELDS} \\
 & \quad \text{and } \mathit{atts} \text{ contains } \texttt{name="}f\texttt{"} \\
0 & \text{otherwise}
\end{cases}
$$

The *forms* function finds the relevant `form` elements in the given template, *gcount* counts the number of occurrences of a given gap name, and *fcount* counts the number of occurrences of a given field name. Note that the latter two functions need to consider the gap presence map of the summary graph. For the field count we can assume that only valid XHTML is shown because of the show analysis presented in the next section, and we can exploit the restrictions about input field elements described in Section 2.3. The set FIELDS $= \{$`input`, `button`, `select`, `textarea`, `object`$\}$ contains all names of elements that define input fields. The function $fc(n, atts)$ counts the number of name-value pairs that may be produced: if *atts* contains `type="radio"`, then it returns $\diamond$; otherwise, if *atts* contains a `type` attribute with value `reset`, `submit`, or `image`, or an attribute with name `disabled` or `declare`, it returns 0; otherwise, if it contains `type="checkbox"` or an attribute named `multiple`, it returns $*$, and otherwise it returns 1. In order to detect whether `disabled` or `declare` occur, the gap presence map and the string edges need to be consulted in case of attribute gaps.

With these auxiliary functions in place, we can now define the value $fp \in C$ representing the number of occurrences of f in the possible unfoldings of the summary graph:

$$fp = \bigsqcup_{r \in R} \Phi(r)$$

If for every root, the number of occurrences is always 0, always $\diamond$, or always 1, the final result is 0, $\diamond$, or 1, respectively; if it sometimes is $\diamond$ and sometimes 1, the result is 1; and otherwise it is $*$. The function Φ traverses the nodes in the summary graph, looking for applicable forms:

$$\Phi(n) = \bigsqcup_{(\mathit{ff},gg) \in allforms(n)} infields(n, (\mathit{ff}, gg)) \sqcup \bigsqcup_{(n,h,m) \in T,\ h \in tgaps(n)} \Phi(m)$$

The left-hand factor counts the field occurrences for each `form` element that occurs directly in the template of n, while the right-hand factor looks at the templates that may be plugged into gaps in n.

$$infields(n, (\mathit{ff}, gg)) = \mathit{ff}(f) \oplus \bigoplus_{h \in G} (gg(h) \otimes infollow(n, h))$$

$$infollow(n, h) = \bigsqcup_{(n,h,m) \in T} infields(m, count(m))$$

Given a current node n and an element (ff, gg) of GFP representing the fields and gaps directly available in a particular form, the *infields* function sums the field occurrences according to ff and those that may occur due to plug operations. For the latter part, we iterate through the possible gaps and multiply each count with the gap multiplicity. The *infollow* function follows the template edges and recursively finds the number of field occurrences in the same way as *outfollow* but now assuming that we are inside a form.

As usual, we can compute the least fixed point by iteration because the lattice is finite and all operations are monotone. Since the *count* and *allforms* functions never return $\perp$, the result, fp, is always in the set $\{0, \diamond, 1, *\}$. The desired properties can now be verified by inspecting that

$fp \in \{1, \diamond\}$ for `receive` operations, and
$fp \neq 0$ for `receive[]` operations

for every summary graph computed for some `show` operation that is connected by a receive edge to the receive operation in the flow graph.

As the plug analysis, this receive analysis is both sound and complete with respect to the summary graphs and the receive edges—assuming that only valid XHTML is ever shown: for a `receive` f operation, the analysis determines that it cannot fail if and only if for every label ℓ of a `show` node which has an edge to the `receive` node, it is the case that in every XML document in $\mathcal{L}(summary_\ell(x))$, each form without an `action` attribute produces exactly one f field value. A similar property holds for `receive[]` operations.

For each `receive` and `receive[]` operation, we calculate *fp* for every summary graph of an associated `show` operation. Thus, *fp* is calculated $O(n^2)$ times, where n is the size of the original JWIG program. The auxiliary functions *count* and *allforms* can be precomputed in time $O(n)$. Each argument to *infields* denotes a specific form element in a template constant. Since there are $O(n)$ template nodes and $O(n)$ form elements in the program, both Φ and *infields* are given at most $O(n)$ different values as arguments. Since the lattice has constant height, we therefore iterate through the summary graph $O(n)$ times. Each iteration performs a single traversal of the summary graph which takes time $O(n^2)$. In total, the receive analysis runs in time $O(n^5)$ in the size of the original JWIG program.

5.3 Show Analysis

For every `show` statement in the JWIG program, we have computed a summary graph that describes how the XML templates are combined in the program and which XML documents may be shown to the client at that point. This gives us an opportunity to verify that all documents being shown are *valid* with respect to some document type. In particular, we wish to ensure that the documents are valid XHTML 1.0 [Pemberton et al. 2000], which is the most commonly used XML language for interactive Web services. XHTML 1.0 is the official XML version of the popular HTML 4.01. It is relatively easy to translate between the two, so in principle our technique works for HTML as well.

Validity of an XML document means that it is well-formed and in addition satisfies some requirements given by a schema for the particular language. The first part, well-formedness, essentially means that the document directly corresponds to a tree structure whose internal nodes are elements by requiring element tags to balance and nest properly. This part comes for free in JWIG, since all XML templates are syntactically required to be well-formed. The remaining validity requirements specify which attributes a given element may have and which text and subelements that may appear immediately below the element in the XML tree. Such properties are specified using a *schema language*. In XHTML, the requirements are given by a DTD (Document Type Definition) schema plus some extra restrictions that cannot be formalized in the DTD language.

Our validation technique is parameterized by the schema description. Thereby we expect that it will be straightforward to support, for instance, WML or VoiceXML which are used for alternative interaction methods, in place of XHTML. Rather that using DTD, we apply a novel schema language, *Document Structure Description 2.0* (DSD2) [Møller 2002]. This schema language is more expressive than DTD, so more validity requirements can be formalized. The expressive power of DSD2 is comparable to that of W3C's XML Schema [Thompson et al. 2001], but DSD2 is significantly simpler, which is indicated below and substantiated in more detail in [Klarlund et al. 2002]. Because of its expressiveness and simplicity, DSD2 is an ideal choice for the present application.

Recall from Section 2.4 that the PowerForms language for expressing high-level form field validation constraints is integrated into JWIG by means of the `show` *X* `powerforms` *P* operation. Since we have summary graphs corresponding to the *P* expression, we can also verify that values of those expressions at runtime always are valid PowerForms documents. In the following, we focus on the validation of the XHTML documents; checking validity of the PowerForms documents is merely a matter of using a schema for the PowerForms language instead of the XHTML schema as parameter for the validation algorithm that we present in the following.

5.4 The Document Structure Description 2.0 Language

The DSD2 language is designed as a successor to the schema language described in [Klarlund et al. 2000; 2002]. A DSD2 schema description of an XML language is itself an XML document. A *DSD2 processor* is a tool that takes as input a DSD2 schema and an XML document called the *instance document*. First it normalizes the instance document according to the schema, for instance, by inserting default attributes and contents. Then it checks whether the normalized instance document is *valid*, that is, that all validity requirements about the attributes and contents of the elements are satisfied.

The following description of DSD2 is intended to give a brief overview—not to define the language exhaustively. A normative specification document for DSD2 is currently under development [Møller 2002].

Conceptually, a DSD2 schema consists of a list of *rules*. A rule is either a *conditional*, a *declaration*, or a *requirement*. Furthermore, there are notions of *uniqueness* and *pointers* which we can ignore here. To simplify the presentation, we omit a description of the normalization phase and focus on the two most central phases: declaration checking and requirement checking. Each of these two phases consist of a traversal of the instance document tree where each element is processed in turn. The *current element* is the one currently being processed.

A conditional rule contains a list of rules whose applicability is guarded by a boolean expression. Only if the boolean expression evaluates to true for the current element, the rules within are considered. Boolean expressions are built of the usual boolean operators, together with `element` expressions which probe the name of the current element, `attribute` expressions which probe the presence and values of attributes, and `parent`, `ancestor`, `child`, and `descendant` operators which probe whether certain properties, which are themselves specified as boolean expressions, are satisfied for the elements above or below the current element in the instance document tree.

In the *declaration checking phase*, it is checked that all present attributes and contents in the current element are allowed, meaning that they are declared by applicable declaration rules. A declaration rule contains a list of *attribute declarations* and *contents declarations*. An attribute declaration specifies that an attribute with a given name is allowed in the current element provided that the value matches a given regular expression. A contents declaration is a regular expression over individual characters and elements that specifies a requirement for the presence, ordering, and number of occurrences of subelements and character data. Elements are described using boolean expressions, as described above. A contents declaration only looks at elements that are mentioned in the regular expression. This subsequence of the contents is called the *mentioned contents*. If the expression contains any character subexpressions, all character data in the contents is included in the mentioned contents. Checking a contents declaration succeeds if the mentioned contents matches the regular expression. This technique allows the contents of an element to be described by multiple regular expressions, each considering only a subsequence of the contents. All attributes and contents that have been matched by an applicable declaration are considered to be *declared*.

In the *requirement checking phase*, the requirement rules are considered. A requirement rule contains boolean expressions that must evaluate to true for the current element, provided that the rule is applicable for that element.

As indicated in the above description, the language is essentially built from boolean logic and regular expressions. For convenience, specifications can be grouped and named for modularity and reuse. Furthermore, the DSD2 schema can restrict the name of the root element: for example, in XHTML, it must be `html`. DSD2 has full support for Namespaces [Bray et al. 1999]; for XHTML, the namespace `http://www.w3.org/1999/xhtml` is used.

As an example, the following snippet of the DSD2 description of XHTML 1.0 describes `dl` elements:

```
<if><element name="h:dl"/>
  <declare>
    <attribute name="compact"><string value="compact"/></attribute>
    <contents>
      <repeat min="1"><union>
        <element name="h:dt"/><element name="h:dd"/>
      </union></repeat>
    </contents>
  </declare>
  <rule ref="h:ATTRS"/>
</if>
```

These rules show that a `dl` element from the XHTML namespace, which is recognized by the `h` prefix, may contain a `compact` attribute, provided that its value is `compact`, and that the contents must contain at least one `dt` or `dd` element. Additionally, `ATTRS`, which is defined elsewhere, describes some additional common attributes that may occur.

The following example (abbreviated with "...") describes `a` elements:

```
<if><element name="h:a"/>
  <declare>
    <attribute name="name"><stringtype ref="h:NMTOKEN"/></attribute>
    <attribute name="shape">
      <stringtype ref="h:SHAPE"/>
      <default value="rect"/>
    </attribute>
    ...
    <contents>
      <repeat><union>
        <string/>
        <boolexp ref="h:PHRASE"/>
        ...
      </union></repeat>
    </contents>
  </declare>
  <rule ref="h:HREFLANG"/>
  ...
  <require>
    <not><ancestor><element name="h:a"/></ancestor></not>
  </require>
</if>
```

This reads: If the current element is named "a", then the subrules are applicable. First, the attributes `name`, `shape`, etc. are declared. The `stringtype` constructs are references to regular expressions defining the valid attribute values. For `shape` attributes, a default is specified. Then, a contents declaration states that all text is allowed as contents together with some contents expressions defined elsewhere. After that, there are some references to rule definitions, and finally, there is a requirement stating that `a` elements cannot be nested. The latter rule is an example of a validity requirement that cannot be expressed by DTD or XML Schema.

As a final example, the following requirement can be found in the description of `input` elements:

```
<require>
  <or>
    <attribute name="type">
      <union><string value="submit"/><string value="reset"/></union>
    </attribute>
    <attribute name="name"/>
  </or>
</require>
```

This states that there must be a `type` attribute with value `submit` or `reset` or a `name` attribute. This is another validity requirement that cannot be expressed concisely in most other schema languages. The whole DSD2 schema for XHTML 1.0 can be found at `http://www.brics.dk/DSD/xhtml1-transitional.dsd`.

5.5 Show Analysis Using DSD2

We show below how the DSD2 processing algorithm explained in the previous section generalizes from concrete XML documents to summary graphs. In fact, the DSD2 language has been designed with summary graph validation in mind. Since the DSD2 language is a generalization of the DTD language, the following algorithm could be adapted to DTD. One benefit of using DSD2 is that every validity requirement that merely appear as comments in the DTD schema for XHTML can be formalized in DSD2, as exemplified in the previous section. Of course, there still are syntactic requirements that even DSD2 cannot express. For instance, in tables, the `thead`, `tfoot`, and `tbody` sections must contain the same number of columns, and the `name` attributes of `a` elements must have unique values. Summary graphs clearly do not have the power to count such numbers of occurrences, so we do not attempt to also check these requirements. Still, our approach based on DSD2 captures more syntactic errors than possible with DTD or XML Schema. Only the uniqueness requirements specified by ID and IDREF attributes are not checked, but they do not play a central role in the schema for XHTML anyway.

Recall from Section 2.3 that we make a few modification of the documents at runtime just before they are sent to the clients. It is trivial to modify the XHTML schema to take these modifications into account. For instance, rather than requiring one or more entries in all lists, the analysis can permit any number since lists with zero entries are always removed.

Given a DSD2 schema and a summary graph SG associated to some `show` statement, we must check that every XML document in $\mathcal{L}(SG)$ is valid according to the schema. In contrast to the analyses described in the previous sections, we will describe this one in a less formal manner because of the many technical details involved. Rather than showing all the complex underlying equation systems or describing the entire algorithm in detailed pseudocode, we attempt to present a concise overview of its structure. Again, we focus on the declaration checking phase and the requirement checking phase. Since neither has side-effects on the instance document, they can be combined into a single traversal. The main part of the algorithm is then structured as two phases:

(1) Every boolean formula (including the subformulas) occurring in the schema is assigned a truth value for every element that occurs in an XML template in SG. This is done inductively in the structure of the formulas.

(2) For every element that occurs in an XML template in SG, the following steps are performed:
   (a) By considering the conditional rules in the schema, the declaration and requirement rules that are applicable to the current element are found;
   (b) declaration checking is performed; and
   (c) requirement checking is performed.

If no violations are detected in the second phase, SG is valid, which implies that all documents in $\mathcal{L}(SG)$ are valid. In a concrete implementation, the first phase is performed lazily, by-need of the second phase. The evaluation of boolean expressions and the declaration checking require further explanation:

Boolean expressions. Evaluation of boolean expressions is nontrivial (1) because we now have to consider all the possible unfoldings of the summary graph and (2) because of the context operators `ancestor`, `descendant`, etc. We apply a *four-valued* logic for that purpose. Evaluating a boolean expression relative to an element results in one of the following values:

true – if evaluating the expression on every possible occurrence of the element in every possible unfolding of the summary graph would result in *true*;

false – if they all would result in *false*;

some – if some occurrence of the element in some unfolding would result in *true* and others in *false*;

don't-know – if the processor is unable to determine any of the above.

All boolean operators extend naturally to these values. The value *don't-know* is for instance produced by the conjunction of *some* and *some*. To evaluate `ancestor` and `descendant` operations, additional traversals of the summary graph may be required. In general, the number of traversals can be bounded by the maximal alternation depth of `ancestor` and `descendant` operators. For the XHTML schema, a total of two traversals always suffice.

When evaluation of expressions in requirement rules results in *false* or *some*, it means that invalid documents may be generated in the JWIG program, so an appropriate warning message is produced. When evaluation of those expressions results in *don't-know* and also when evaluation of the guard expressions of conditional schema rules result in *some* or *don't-know*, we terminate with a "don't know" message. However, for our concrete XHTML schema, that can in fact never happen since all the guard expressions only test the name of the current element.

Declaration checking. In the declaration checking step, we check that all attributes and contents of the current element are declared by applicable declarations. An attribute is declared by an attribute declaration if the attribute value matches the regular expression of the declaration. In case the attribute is an attribute gap, this amounts to checking inclusion of one regular language of Unicode strings in another. However, it is possible that one attribute declaration specifies that a given attribute may have one set of values and another declaration specifies that the same attribute may also have another set of values. Therefore, in general, we check that all values that are possible according to the string edges match some declaration. If some attribute is not declared, the summary graph is not valid.

As explained in Section 5.4, a contents sequence matches a contents declaration if the mentioned contents is in the language of the regular expression of the declaration. In case there are no gaps in the contents, this is a simple check of string inclusion in a regular language. If there are gaps, the situation is more involved: the template edges from the gaps may lead to templates which at the top-level themselves contain gaps. (The *top-level* of a template is the sequence of elements, characters, and gaps that are not enclosed by other elements.) In turn, this may cause loops of template edges. Therefore, in general, the set of possible contents sequences forms a context-free language, which we represent by a context-free grammar. Without such loops, the language is regular. The problem of deciding inclusion of a context-free language in a regular language is decidable [Hopcroft and Ullman 1979], but computationally expensive. For that reason, we approximate the context-free language by a regular one: whenever a loop is found in the context-free grammar, we replace it by the regular language A^* where A consists of the individual characters and elements occurring in the loop. This allows us to apply a simpler regular language inclusion algorithm. Although loops in summary graphs often occur, our experiments show that it is rare that this approximation actually causes any imprecision. The reason for this is that summary graph loops result from iteratively or recursively constructed XML values in the JWIG program, and for such values, the XHTML schema typically does not impose any ordering requirements on the contents sequences. More precise approximation techniques could of course also be used [Mohri and Nederhof 2001]. In addition to checking that all contents declarations are satisfied, we check that all parts of the contents have been declared, that is, matched by some declaration. If not, the summary graph is not valid. Again, if any declaration contains a character subexpression, all character data is considered declared.

Compared with the technique described in [Brabrand et al. 2001], we have now moved from an abstract version of DTD to the more powerful DSD2 schema language. Furthermore, by the introduction of four-valued logic for evaluation of boolean expressions, we have repaired a defect that in rare cases caused the old algorithm to fail.

The whole algorithm runs in linear time in the size of the XML templates. We show experimental results in the next section. The algorithm is sound; that is, if it validates a given summary graph it is certain that all unfoldings into concrete XML documents are also valid. Because of the possibility of returning "don't know" and of the approximation of context-free languages by regular ones, the algorithm is generally not complete. An alternative to our approach of "giving up" when these situations occur would be to branch out on all individual unfoldings and use classical two-valued logic. This indicates that the problem is decidable. However, the complexity of the algorithm would then increase significantly. Anyway, for the XHTML schema, false errors can only occur in the rare cases where we actually need to approximate context-free languages by regular ones, as mentioned above.

6. IMPLEMENTATION AND EVALUATION

To make experiments for evaluating the JWIG language design and the performance of the program analyses, we have made a prototype implementation. It consists of the following components, roughly corresponding to the structure in Figures 6 and 7:

—a simple desugarer, made with JFlex [Klein 2001], for translating JWIG programs into Java code;

—the `<bigwig>` low-level runtime system [Brabrand et al. 1999] which in its newest version [Møller 2001a] is based on a module for the Apache Web Server [Behlendorf et al. 2002] and extended with Java support;

—a Java-based runtime system for representing and manipulating XML templates;

—a Java implementation of the PowerForms tool [Brabrand et al. 2000];

—a part of the Soot optimization framework for Java [Vallee-Rai et al. 1999; Sundaresan et al. 2000], which converts Java bytecode to a more convenient three-address instruction code language, called Jimple;

—a flow graph package, which operates on the Jimple code generated by Soot and also uses Soot's CHA implementation;

—a finite-state automaton package with UTF16 Unicode alphabet and support for all standard regular operations [Møller 2001b];

—a summary graph construction package which also performs the string analysis;

—a plug and receive analyzer which performs the checks described in Section 5.1 and 5.2; and

—a DSD2 validity checker with summary graph support and schemas for XHTML 1.0 and PowerForms.

All parts are written in Java, except the low-level runtime system which is written in C. The Java part of the runtime system amounts to 3000 lines, and the analysis framework is 12,500 lines. A user manual is available online [Christensen and Møller 2002].

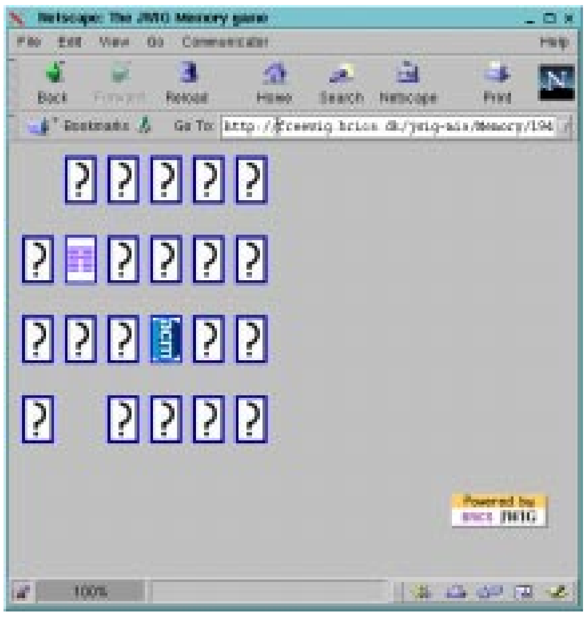

Netscape: The JWIG Memory game
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Go To: http://freewig.brics.dk/jwig-mis/Memory/194
Powered by BRICS JWIG
100%

Fig. 13. A snapshot of the Memory Game being played.

6.1 Example: The Memory Game

To give a more complete example of a JWIG service, we present the well-known Memory Game, where the player must match up pairs of cards lying face down. First, the number of pairs is chosen, next the game itself proceeds iteratively, and finally the player is congratulated. A snapshot of the game in progress is seen in Figure 13.

The main session, presented in Figure 14, looks just like a corresponding sequential Java program. The templates being used are presented in Figure 15. The construction of a grid of cards is performed by the `makeCardTable` method presented in Figure 16. The class representing individual cards is seen in Figure 17. In all, the Memory Game is written in 163 lines of JWIG.

By itself, the session concept and the XML templates simplify the program compared to solutions in JSP or Servlets. Furthermore, since the example is analyzed without errors, we know that no JWIG exceptions will be thrown while the game is being played. In particular, we are guaranteed that all documents being shown are valid XHTML according to the strict standard imposed by the DSD2 schema.

The JWIG runtime system, which is also used in the `<bigwig>` project, is tailor-made for session-based Web services. Each session thread is associated a unique URL which refers to a file on the server. This file at all times contains the most recent page shown to the client. The session code runs as a JVM thread that lives for the entire duration of the session. In contrast, sessions in Servlet and JSP services run as short-lived threads where the session identity is encoded using cookies or hidden input fields, as described in Section 1.3. This precludes sessions from being bookmarked, such that the client cannot suspend and later resume a

```
public class Game extends Session {
    public void main() {
        // ask for number of cards
        int howmany;
        do {
            show wrap <[ body = welcome <[ atmost = images.length ]]);
            howmany = Integer.parseInt(receive howmany);
        } while (howmany < 1 || howmany > images.length);

        // generate random permutation of cards
        Card[] cards = new Card[howmany*2];
        Random random = new Random();
        for (int i = 0 ; i < howmany ; i++) {
            for (int c = 0 ; c < 2 ; c++) {
                int index;
                do {
                    index = random.nextInt(howmany*2);
                } while (cards[index] != null);
                cards[index] = new Card(i);
            }
        }

        // play the game
        int pairsleft = howmany;
        int moves = 0;
        show makeCardTable(cards);
        while (pairsleft > 0) {
            // first card picked
            int firstcard = Integer.parseInt(receive submit);
            cards[firstcard].status = 1;
            show makeCardTable(cards);

            // second card picked
            int secondcard = Integer.parseInt(receive submit);
            cards[secondcard].status = 1;
            moves++;

            // check match
            if (cards[firstcard].value == cards[secondcard].value) {
                cards[firstcard].status = 2;
                cards[secondcard].status = 2;
                if (--pairsleft > 0)
                    show makeCardTable(cards);
            } else {
                show makeCardTable(cards);
                cards[firstcard].status = 0;
                cards[secondcard].status = 0;
            }
        }

        // done, show result
        exit farewell <[ howmany = howmany, moves = moves ];
    }
}
```

Fig. 14. The main session of the Memory Game.

```
private static final XML wrap = [[
  <html>
    <head><title>The JWIG Memory game</title></head>
    <body><form><[body]></form></body>
  </html>
]];

private static final XML welcome = [[
  <h3>Welcome to the JWIG Memory game!</h3>
  <p>How many pairs of cards do you want (from 1 to <[atmost]>)?</p>
  <input type="text" name="howmany"/>
]];

private static final XML farewell = wrap <[ body = [[
  <h3>Thank you for playing this game!</h3>
  <p>You found all <[howmany]> pairs using <[moves]> moves.</p>
]] ];
```

Fig. 15. Templates from the Memory Game.

```
private XML makeCardTable(Card[] cards) {
    XML table = [[ <table><[row]></table> ]];
    for (int y=0; y < (cards.length+COLS-1)/COLS; y++) {
        XML row = [[ <tr><[elem]></tr><[row]> ]];
        for (int x=0; x < COLS; x++) {
            XML elem = [[ <td><[contents]></td><[elem]> ]];
            int index = y*COLS+x;
            if (index < cards.length) {
                elem = elem <[ contents = cards[index].makeCard(index) ];
            }
            row = row <[ elem = elem ];
        }
        table = table <[ row = row ];
    }
    return wrap <[ body = table ];
}
```

Fig. 16. Generating a grid of cards in the Memory Game.

session, and the history buffer in the browser typically gets cluttered with references to obsolete pages. In our solution, the session URL functions as an identity of the session, which avoids all these problems. These aspects are described in more detail in [Brabrand et al. 2002].

If we introduce an error in the program, by forgetting the `name` attribute in the `input` field in the `welcome` template, then the JWIG analyzer produces the following output:

```
*** Field ‘howmany’ is never available on line 68
*** Invalid XHTML at line 49
--- element ’input’: requirement not satisfied:
<or xmlns="http://www.brics.dk/DSD/2.0/error">
 <attribute name="type">
  <union>
```

```
private class Card {
    public int status;
    public int value;

    public Card(int value) {
        this.status = 0;
        this.value = value;
    }

    public XML makeCard(int index) {
        switch(status) {
        case 0:
            return [[ <input type="image" alt="card"
                             src=[image] name=[index] /> ]]
                   <[ image = back_image, index = index ];
        case 1:
            return [[ <img src=[image] alt=[num] /> ]]
                   <[ image = images[value], num = value ];
        case 2:
            return [[ <img src=[image] alt="" /> ]]
                   <[ image = blank_image ];
        default:
            return null;
        }
    }
}
```

Fig. 17. Representing a card in the Memory Game.

```
    <string value="submit" />
    <string value="reset" />
   </union>
  </attribute>
  <attribute name="name" />
</or>
```

In the first line, the receive analysis complains that the *howmany* field is never available from the client. The remainder of the error message is from the show analysis, which notices that the `input` element violates the quoted requirement from the XHTML schema. This particular validity error is not caught by DTD validation of the generated document. If the involved element contained gaps, the error message would include a print of all relevant template and string edges and values of the gap presence map, which shows the relevant plug operations. Clearly, such diagnostics are useful for debugging.

6.2 Performance

The JWIG implementation may be evaluated with respect to compile-time, analysis-time, or runtime. The compile-time performance is not an issue, since JWIG programs are simply piped through a JFlex desugarer and compiled using a standard Java compiler. The JWIG runtime performance is not particularly interesting, since we reuse the `<bigwig>` runtime system and the standard J2SE JVM. The runtime implementation of the XML values is quite mature. We have optimal runtime com-

Name	Lines	Templates	Shows/Exits	Total Time
Chat	79	4	3	9.718
Guess	97	8	7	11.126
Calendar	132	6	2	10.032
Memory	163	9	6	15.136
TempMan	216	13	3	11.016
WebBoard	766	32	24	13.507
Bachelor	1078	88	14	131.320
Jaoo	3923	198	9	39.929

Fig. 18. The benchmark services.

Name	Load	Construct	Size Before	Simplify	Size After
Chat	5.893	1.762	246/399	0.274	107/99
Guess	6.380	1.798	326/506	0.255	124/100
Calendar	6.172	1.957	396/685	1.957	124/127
Memory	6.270	1.997	469/789	0.293	144/129
TempMan	6.084	2.370	792/1440	0.965	205/195
WebBoard	5.973	2.660	990/1330	0.906	422/287
Bachelor	5.777	4.496	2311/3488	22.037	1059/914
Jaoo	6.138	5.588	3078/4484	14.208	1406/1008

Fig. 19. Flow graph construction.

plexities, in the sense that a plug operation always takes constant time and the printing of a document is linear in the size of the output. The critical component in our system is the extensive collection of static analyses that we perform on the generated class files.

As shown in Figure 7, the static analysis is a combination of many components, which we in the following quantify separately. Our benchmark suite, shown in Figure 18, is a collection of small- to medium-sized JWIG services, most of which have been converted from corresponding `<bigwig>` applications [Brabrand et al. 2001]. The right-most column shows the total time in seconds for the entire suite of program analyses. All experiments are performed on a 1 GHz Pentium III with 1 GB RAM running Linux. For all benchmarks, at most 150 MB memory is used.

The four larger ones are an XML template manager where templates can be uploaded and edited (`TempMan`), an interactive Web board for on-line discussions (`WebBoard`), a system for study administration (`Bachelor`), and a system for management of the JAOO 2001 conference (`Jaoo`).

Figure 19 shows the resources involved in computing the flow graphs. For each benchmark we show the time in seconds used by Soot, the time in seconds used by Phases 1 through 7 described in Section 3, the size of the resulting flow graph, the time in seconds used by the simplifying Phase 8, and the size of the simplified flow graph. The flow-graph sizes are shown as number of nodes and number of flow edges. The loading time is dominated by initialization of Soot. Phases 1 through 7 of the flow-graph construction are seen to be linear in the program size. The time for the simplification phase strongly depends on the complexity of the document constructions, which explains the relatively large number for the `Bachelor` service. For all benchmarks, the simplification phase substantially reduces the flow-graph

Name	Time	Largest Size
Chat	0.136	2/1/5
Guess	0.114	3/2/2
Calendar	0.347	5/9/5
Memory	2.193	7/8/5
TempMan	0.275	8/9/5
WebBoard	1.473	9/11/11
Bachelor	38.478	47/83/24
Jaoo	5.260	33/45/48

Fig. 20. Summary graph construction.

Name	Plug	Receive	Show
Chat	0.004	0.014	1.645
Guess	0.003	0.011	2.565
Calendar	0.003	0.006	1.353
Memory	0.003	0.008	4.372
TempMan	0.003	0.012	1.370
WebBoard	0.005	0.013	2.477
Bachelor	0.007	0.016	60.509
Jaoo	0.018	0.007	8.710

Fig. 21. Summary graph analysis.

size. Furthermore, recall that before the simplification phase, flow edges may have multiple variables, while after simplification, they all have exactly one variable.

Figure 20 quantifies the computation of summary graphs, including the string analysis. For each benchmark we show the total time in seconds and the size of the largest summary graph, in terms of nodes, template edges, and nontrivial string edges. The relatively large numbers for the `Bachelor` example correctly reflects that it constructs complicated documents. Without graph simplification, the total time for the `Memory` example blows up to more than 15 minutes, while the `Jaoo` example was aborted after 90 minutes. We conclude that summary graph construction appears to be inexpensive in practice and that graph simplification is worth the effort.

Figure 21 deals with the subsequent analysis of all the computed summary graphs. For each benchmark we show the total time in seconds for each of the three analyses and the total number of false errors generated by the conservative analyses.

We conclude that the JWIG prototype implementation is certainly feasible to use, but that there is room for performance improvements for the implementation.

6.3 Precision

Since our analyses are sound, we know that accepted programs will never contain errors. Thus, the evaluation of precision will focus on our ability to accept safe programs and the frequency of false errors. Among our benchmark programs, none contained plug or receive errors and all were accepted by the corresponding analyses. The larger benchmarks were flagged by the show analysis, but, by careful inspection, all complaints were seen to correctly identify actual XHTML validity errors. Thus, the analysis appears to be precise enough to serve as a real help to the programmer.

All in all, we encountered no false errors. It is, of course, easy to construct programs that will be unfairly rejected, but those do not seem to occur naturally.

7. PLANS AND IDEAS FOR FUTURE DEVELOPMENT

Our current system can be extended and improved in many different directions which we plan to investigate in future work. These can be divided into language design, program analysis, and implementation issues, and are briefly described in the following.

7.1 Language design

So far, the design of JWIG has focused on two topics that are central to the development of interactive Web services: sessions and dynamic construction of Web documents. However, there are many more topics that could benefit from high-level language-based solutions, as shown in [Brabrand et al. 2002].

The current XML cast operation in JWIG is somewhat unsatisfactory for two reasons: (1) if a cast fails due to invalid XHTML, an exception is not thrown immediately since it is not detected until a subsequent show operation; and (2) its expressiveness is limited—for instance, unions of templates cannot be expressed. One solution to this may be to use DSD2 descriptions instead of constant templates in the cast operations. However, to generalize the analyses correspondingly, a technique for transforming a DSD2 description of an XML language into a summary graph is needed. We believe that this is theoretically possible—further investigation will show whether it is also practically feasible.

Another idea is to broaden the view from interactive Web services to whole Web sites comprising many services and documents. The Strudel system [Fernandez et al. 1999] has been designed to support generation and maintenance of Web sites according to the design principle that the underlying data, the site structure, and the visual presentation should be separated. A notion of data graphs allows the underlying data to be described, a specialized query language is used for defining the site structure, and an HTML template language that resembles the XML template mechanism in JWIG defines the presentation. We believe that the development of interactive services can be integrated into such a process. For sites that comprise both complex interactive session-based services and more static individual pages, the concepts in the `Service.Session` and `Service.Page` classes could be applied. JWIG could also benefit from a mechanism for specifying dependencies between the pages or sessions and the data, for instance, such that pages are automatically cached and only recomputed when certain variables or databases are modified.

We have shown that our template mechanism is suitable for constructing XHTML documents intended for presentation. If the underlying data of a Web service is represented with XML, as suggested by Strudel, we will need a general mechanism for extracting and transforming XML values. Currently, we only provide the plug operation for combining XML templates—a converse "unplug" operation would be required for deconstructing XML values. Preliminary results suggest that our notion of summary graphs and our analyses generalize to such general XML transformations [Christensen et al. 2002]. XDuce [Hosoya and Pierce 2000] is a related research language designed to make type-safe XML transformations. In XDuce, types are simplified DTDs where we instead use the more powerful DSD2 notation.

Furthermore, XDuce is a tiny functional language while JWIG contains the entire Java language. Instead of relying on a type system for ensuring that the various XML values are valid according to some schema definition, we perform data-flow analyses based on summary graphs. Based on these ideas, a current project aims to make JWIG a general and safe XML transformation language.

The XML template mechanism in JWIG can be separated from the session-based model. Thereby, the XML template mechanism could be integrated into, for instance, Servlets, replacing its output stream approach for constructing XHTML documents, and the plug and show analyses could then be applied to obtain the static guarantees. However, the receive analysis depends on the session-based model, which is not available in Servlets. Alternatively, our language design and program analysis ideas are readily applicable to other languages that construct XML values—all that is needed is a means for obtaining flow graphs similar to ours.

7.2 Program analysis

The experiments indicate that the notion of summary graphs is suitable for modeling the XML template mechanism and that the analysis precision is adequate. However, the preliminary string analysis described in Section 4.1 can be improved. The modular design of the analyses makes it possible to replace this simple string analysis by a more precise one, such as [Christensen et al. 2003]. For example, string concatenation operations can be modeled more precisely by exploiting the fact that regular languages are closed under finite concatenation. Because of loops in the flow graphs, this will in general produce context-free languages so a suitable technique for approximating these by regular languages is needed. That essentially amounts to applying widening for ensuring termination. Other operations, such as the substring methods, can also easily be modeled more precisely than with `anystring`. An advanced version of such an analysis would apply flow-sensitivity, such that, for example, `if` statements that branch according to the value of a string variable would be taken into account, and instead of modeling `receive` by `anystring`, the regular languages provided by PowerForms specifications could be applied. A natural extension to these ideas would be to add a "regular expression cast" operator to the JWIG language. As with the other cast operations, that would provide a back-door to the analysis, which occasionally can be convenient no matter how precise the analysis may be.

The current program analysis is based on the assumption that the medium used for communication with the clients is XHTML. However, since the show analysis is parameterized by a DSD2 description, validity with respect to any XML language describable by DSD2 can be checked instead. Two obvious alternatives are WML, *Wireless Markup Language* [WAP Forum 2001], which is used for mobile WAP devices with limited network bandwidth and display capabilities, and VoiceXML, *Voice Extensible Markup Language* [Boyer et al. 2000], for audio-based dialogs. Such languages can be described precisely with DSD2. Only the receive analysis requires modification since it needs to identify the forms and fields, or whatever the equivalent notions are in other formalisms.

7.3 Implementation

Our current implementation is a prototype developed to experiment with the design and test the performance. This means that there are plenty of ways to improve the performance of the analysis and the runtime system.

We plan to apply the `metafront` syntax macros [Brabrand et al. 2003] in a future version to improve the quality of the parsing error messages. This will also allow us to experiment with syntax macros as a means for developing highly domain-specific languages in the context of Java-based interactive Web services.

Finally, we believe that it is possible to significantly improve the runtime performance for JWIG services by integrating the JWIG runtime system with a Java Enterprise Edition server. For instance, this allows `Service.Page` to become essentially as efficient as JSP code by exploiting that the threads are never suspended by `show` statements. JRockit [Appeal Virtual Machines 2002] is a commercial JVM implementation which is tuned for Web servers with high loads. In particular, it supports light-weight threads which will significantly reduce the overhead induced by our session model.

8. CONCLUSION

We have defined JWIG as an extension of the Java language with explicit high-level support for two central aspects of interactive Web services: (1) sessions consisting of sequences of client interactions and (2) dynamic construction of Web pages. Compared to other Web service programming languages, these extensions can improve the structure of the service code. In addition to being convenient during development and maintenance of Web services, this allows us to perform specialized program analyses that check at compile-time whether or not runtime errors may occur due to the construction of Web pages or communication with the clients via input forms. The program analyses are based on a unique notion of *summary graphs* which model the flow of document fragments and text strings through the program. These summary graphs prove to contain exactly the information needed to provide all the desired static guarantees of the program behavior.

This article can be viewed as a case study in program analysis. In contains a total of seven analyses operating on different abstractions of the source program: one for making receive edges during flow-graph construction, the reaching definitions analysis in the flow-graph simplification phase, the string analysis, the summary graph construction, and the plug, receive, and show analyses. The whole suite of analyses is modular in the sense that each of them easily can be replaced by a more precise or efficient one, if the need should arise. If, for example, future experience shows that the control-flow information in the flow graphs is too imprecise, one could apply a *variable-type analysis* [Sundaresan et al. 2000] instead of CHA. Or, if the string analysis should turn out to be inadequate for developing, for example, WML services, it could be replaced by another. Analysis correctness is given by the correctness of each phase. For instance, the flow graphs conservatively approximate the behavior of the original JWIG programs, the summary graphs conservatively model the template constructions with respect to the flow graphs, and the validity results given by the show analysis are conservative with respect to the summary graphs.

The language extensions permit efficient implementation, and despite the theoretical worst-case complexities of the program analyses, they are sufficiently precise and fast for practical use. For our benchmark suite we have encountered no false errors and the overhead of analysis is small enough to fit into a normal development cycle.

All source code for our JWIG implementation, including API specifications and the DSD2 schema for XHTML 1.0, is available online from `http://www.jwig.org/`.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley.

Appeal Virtual Machines. 2002. JRockit – the faster server JVM. `http://www.jrockit.com/`.

Atkinson, L. 2000. *Core PHP Programming*, 2nd ed. Prentice Hall.

Behlendorf, B. et al. 2002. The Apache HTTP server project. `http://httpd.apache.org/`.

Box, D. et al. 2000. Simple object access protocol (SOAP) 1.1. W3C Note. `http://www.w3.org/TR/SOAP/`.

Boyer, L., Danielsen, P., Ferrans, J., Karam, G., Ladd, D., Lucas, B., and Rehor, K. 2000. *Voice eXtensible Markup Language, Version 1.0.* W3C. W3C Note, `http://www.w3.org/TR/voicexml/`.

Brabrand, C. 2000. *<bigwig> Version 1.3 – Reference Manual.* BRICS, Department of Computer Science, University of Aarhus. Notes Series NS-00-4.

Brabrand, C., Møller, A., Ricky, M., and Schwartzbach, M. I. 2000. PowerForms: Declarative client-side form field validation. *World Wide Web Journal 3,* 4 (December), 205–314. Kluwer.

Brabrand, C., Møller, A., Sandholm, A., and Schwartzbach, M. I. 1999. A runtime system for interactive Web services. *Computer Networks 31,* 11-16 (May), 1391–1401. Elsevier. Also in Proc. 8th International World Wide Web Conference, WWW8.

Brabrand, C., Møller, A., and Schwartzbach, M. I. 2001. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01.* 221–231.

Brabrand, C., Møller, A., and Schwartzbach, M. I. 2002. The `<bigwig>` project. *ACM Transactions on Internet Technology 2,* 2, 79–114.

Brabrand, C., Schwartzbach, M. I., and Vanggaard, M. 2003. The metafront system: Extensible parsing and transformation. In *Proc. 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications, LDTA '03.*

Bray, T., Hollander, D., and Layman, A. 1999. Namespaces in XML. W3C Recommendation. `http://www.w3.org/TR/REC-xml-names/`.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. 2000. Extensible Markup Language (XML) 1.0 (second edition). W3C Recommendation. `http://www.w3.org/TR/REC-xml`.

Chase, D. R., Wegman, M., and Zadeck, F. K. 1990. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '90.*

Christensen, A. S. and Møller, A. 2002. *JWIG User Manual.* BRICS, Department of Computer Science, University of Aarhus. Available from `http://www.jwig.org/manual/`.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. 2002. Static analysis for dynamic XML. Tech. Rep. RS-02-24, BRICS. May. Presented at Programming Language Technologies for XML, PLAN-X, October 2002.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. 2003. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03.* LNCS, vol. 2694. Springer-Verlag, 1–18.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. 2001. Web services description language (WSDL) 1.1. W3C Note. `http://www.w3.org/TR/wsdl`.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. 9th European Conference on Object-Oriented Programming, ECOOP '95*. LNCS, vol. 952. Springer-Verlag.

Exolab Group. 2002. Castor. `http://castor.exolab.org/`.

Fernandez, M. F., Suciu, D., and Tatarinov, I. 1999. Declarative specification of data-intensive Web sites. In *Proc. 2nd Conference on Domain-Specific Languages, DSL '99*. USENIX/ACM.

Homer, A., Schenken, J., Gibbs, M., Narkiewicz, J. D., Bell, J., Clark, M., Elmhorst, A., Lee, B., Milner, M., and Rehan, A. 2001. *ASP.NET Programmer's Reference*. Wrox Press.

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Hosoya, H. and Pierce, B. C. 2000. XDuce: A typed XML processing language. In *Proc. 3rd International Workshop on the World Wide Web and Databases, WebDB '00*. LNCS, vol. 1997. Springer-Verlag.

Hunter, J. and McLaughlin, B. 2001. JDOM. `http://jdom.org/`.

Kam, J. B. and Ullman, J. D. 1977. Monotone data flow analysis frameworks. *Acta Informatica 7,* 305–317.

Klarlund, N., Møller, A., and Schwartzbach, M. I. 2000. Document Structure Description 1.0. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-00-7. Available from `http://www.brics.dk/DSD/`.

Klarlund, N., Møller, A., and Schwartzbach, M. I. 2002. The DSD schema language. *Automated Software Engineering 9,* 3, 285–319. Kluwer. Earlier version in Proc. 3rd ACM SIGPLAN-SIGSOFT Workshop on Formal Methods in Software Practice, FMSP '00.

Klein, G. 2001. JFlex – the fast scanner generator for Java. `http://www.jflex.de/`.

Ladd, D. A. and Ramming, J. C. 1996. Programming the Web: An application-oriented language for hypermedia services. *World Wide Web Journal 1,* 1 (January). O'Reilly & Associates. Proc. 4th International World Wide Web Conference, WWW4.

McClanahan, C. R. et al. 2002. Struts. `http://jakarta.apache.org/struts/`.

Møller, A. 2001a. The `<bigwig>` runtime system. `http://www.brics.dk/bigwig/runwig/`.

Møller, A. 2001b. dk.brics.automaton – finite-state automata and regular expressions for Java. `http://www.brics.dk/automaton/`.

Mohri, M. and Nederhof, M.-J. 2001. *Robustness in Language and Speech Technology*. Kluwer Academic Publishers, Chapter 9: Regular Approximation of Context-Free Grammars through Transformation.

Møller, A. 2002. Document Structure Description 2.0. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from `http://www.brics.dk/DSD/`.

Netscape. 1999. Server-side JavaScript. `http://developer.netscape.com/docs/manuals/ssjs.html`.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis*. Springer-Verlag.

Oskoboiny, G. 2001. HTML Validation Service. `http://validator.w3.org/`.

Pemberton, S. et al. 2000. XHTML 1.0: The extensible hypertext markup language. W3C Recommendation. `http://www.w3.org/TR/xhtml1`.

Raggett, D., Hors, A. L., and Jacobs, I. 1999. HTML 4.01 specification. W3C Recommendation. `http://www.w3.org/TR/html4/`.

Reps, T. 1998. Program analysis via graph reachability. *Information and Software Technology 40,* 11-12 (November/December), 701–726.

Sandholm, A. and Schwartzbach, M. I. 2000. A type system for dynamic Web documents. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00*.

Sun Microsystems. 2001a. Java Servlet Specification, Version 2.3. Available from `http://java.sun.com/products/servlet/`.

Sun Microsystems. 2001b. JavaServer Pages Specification, Version 1.2. Available from `http://java.sun.com/products/jsp/`.

SUN MICROSYSTEMS. 2002. JAXB. `http://java.sun.com/xml/jaxb/`.

SUNDARESAN, V., HENDREN, L. J., RAZAFIMAHEFA, C., VALLEE-RAI, R., LAM, P., GAGNON, E., AND GODIN, C. 2000. Practical virtual method call resolution for Java. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00*.

THOMPSON, H. S., BEECH, D., MALONEY, M., AND MENDELSOHN, N. 2001. XML Schema part 1: Structures. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

VALLEE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. 1999. Soot – a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference, CASCON '99*. IBM.

WAP FORUM. 2001. *Wireless Markup Language, Version 2.0.* Wireless Application Protocol Forum. `http://www.wapforum.org/`.