BRICS RS-03-38 Böttger et al.: Contracts for Cooperation between Programmers and Designers

BRICS

Basic Research in Computer Science

Contracts for Cooperation between Web Service Programmers and HTML Designers

Henning Böttger
Anders Møller
Michael I. Schwartzbach

BRICS Report Series **RS-03-38**

ISSN 0909-0878 **November 2003**

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

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/03/38/`

Contracts for Cooperation between Web Service Programmers and HTML Designers

Henning Böttger, Anders Møller*, and Michael I. Schwartzbach
BRICS†, Department of Computer Science
University of Aarhus, Denmark
{bottger,amoeller,mis}@brics.dk

Abstract

Interactive Web services consist of a mixture of HTML fragments and program code. The fragments, which are maintained by designers, are combined to form HTML pages that are shown to the clients. The code, which is maintained by programmers, is executed on the server to handle the business logic. Current Web service frameworks provide little help in separating these constituents, which complicates cooperation between programmers and HTML designers.

We propose a system based on XML templates and formalized contracts allowing a flexible separation of concerns. The contracts act as interfaces between the programmers and the HTML designers and permit tool support for statically checking that both parties fulfill their obligations. This ensures that (1) programmers and HTML designers work more independently focusing on their own expertises, (2) the Web service implementation is better structured and thus easier to develop and maintain, (3) it is guaranteed that only valid HTML is sent to the clients even though it is constructed dynamically, (4) the programmer uses the XML templates consistently, and (5) the form input fields being sent to the client always match the code receiving those values. Additionally, we describe tools that aid in the construction and management of contracts and XML templates.

1 Introduction

An interactive Web service contains both HTML markup and program code. Ideally, these two constituents are maintained by different experts: HTML designers and programmers. This would be a relatively simple task, if the HTML pages shown to clients could be statically defined and kept separate from the code. However, most pages are dynamically generated and are typically composed by numerous copies of a specific collection of HTML fragments.

*Supported by the Carlsberg Foundation contract number ANS-1069/20.
†Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

Consider for example a Webboard showing a list of current discussions threads. The length and depth of these are not known before runtime, but the entire HTML page is built of fragments that could be called `Wrapper`, `Header`, `Footer`, `ThreadTitle`, `ThreadItem`, and `ThreadStatus`. Other parts of the page could depend of the preferences of the current client and be composed from `MenuHeader`, `MenuItem`, and so on. If the Webboard allows clients to choose among different skins, then each of the above fragments must even exist in several versions.

Consequently, these HTML fragments are intertwined with the code. In a simple approach using plain Servlets [24] or JSP [25], this means that the lines of the source file are alternating Java code and HTML markup and, clearly, it is difficult to develop and maintain such a hybrid document. In typical scenarios, the programmer and the HTML designer fight for control across a spectrum of possible collaborations:

- the designer presents the desired layout as a series of graphical snapshot, which the programmer tries to emulate;
- the programmer ends up designing the pages, with general guidelines from the designer; or
- the programmer and designer work together in a form of *extreme programming* [4].

Neither of these models is satisfactory or optimally productive and, hence, various techniques may be considered to decouple these activities. In the Java world, it is possible to introduce custom tag libraries [21] containing all the detailed HTML markup. The programmer would then use abstract markup tags such as `<Wrapper>`, `<Header>` and `<Footer>`, whose actual contents may reside in separate files and be independently determined by the designer. With such an approach, however, the decoupling has become too complete and several new problems appear:

- since the designer does not know the flow of control, the concrete HTML fragments may not fit together;
- the designer may inadvertently change the markup such that required form fields disappear; and
- it is near impossible to determine at compile time if the service always generates syntactically valid HTML pages.

We propose a solution that introduces a notion of *contracts* between programmers and designers. This supports the principle of *low coupling* by allowing the two parties to work independently of each other [15]. Intuitively, a contract defines the *shapes* of a collection of jigsaw puzzle pieces. Independently of each other, the programmer may build pages from copies of these pieces and the designer may decorate them with HTML markup. Even when the programmer and the designer is the same person, the contract facilitates separation of concerns.

In this presentation, we use HTML [20] and XHTML [19], the XML [6] variant of HTML, interchangeably. The readers are presumably most familiar with HTML; the tools and techniques we present use XHTML, but it is straightforward to convert in either direction.

Contributions

The contributions of this paper are the following:

- A convenient formal language for specifying contracts between programmers and designers;
- tool support for independently checking that both parties fulfill their obligations, which implies that certain potential runtime errors related to the cooperation between them are detected at compile time;
- tool support for automatically inferring a contract from a Web service, which is useful as a starting point for applying our techniques on preexisting Web services;
- tool support for managing XHTML templates for the designer; and
- experimental evidence that the approach is practically useful in aiding the cooperation between programmers and designers and thereby increasing their productivity.

We work in the context of the JWIG system, which is a Java-based language for developing interactive Web services. JWIG has two special features: (1) A higher-order template mechanism for XHTML page construction; and (2) static guarantees of validity of the dynamically generated pages and consistency of the use of form input fields. The static guarantees are provided using a data-flow analysis that models the manipulation of XML values in the program. JWIG is described in detail in an ACM TOPLAS article [8] and the user manual [7].

Overview

We first, in Section 2, discuss related work on the concepts of separation of concerns and component-based Web development, arguing that existing technologies are insufficient. Section 3 gives an overview of the XML template mechanism in JWIG. In Section 4, we define our contract language and show examples of its use and the benefits it provides.

Section 5 explains how to check automatically that both the programmer and the designer fulfills a given contract. Section 6 describes how other tools can help developing and maintaining Web services using contracts. Section 7 describes our implementation and experiments indicating that our approach successfully allows separation of concerns and collaboration.

2 Related Work

There exist numerous technologies and methodologies for developing Web services; due to the limited space we can only mention a few of the most relevant and focus on those that are based on Java. A common theme is *separation of concerns*: Web sites contains program code that interacts with the clients, accesses databases contents, controls sessions of interactions, and builds HTML pages from small HTML fragments and the actual contents. By identifying the different components, different people may focus on developing and maintaining different aspects of the site, which can increase productivity and the stability of the site.

Model2 [22] is a variant of the model-view-controller (MVC) design pattern [13] tailored for structuring Java-based Web applications. A *controller* Servlet [24] receives a HTTP request, updates the *model*, which represents the underlying data, and then invokes the appropriate *view* renderer, a JSP page [25] that produces the HTML page being sent back to the client. In this paper, our main distinction is between the *programming tasks* and the *designer tasks*, rather than the considering the individual programming tasks.

The Apache community is developing a number of popular projects that are related to ours. Apache Struts [16] is a framework for building Web applications with Java based on a further development of the MVC/Model2 approach, providing a generic controller component that can be configured declaratively. The Tiles extension of Struts is a template mechanism reminiscent of the one in JWIG, but without the static guarantees of validity of the dynamically generated pages and consistency of the use of form fields.

Apache Cocoon is an XML-based development framework using a model of pipelined components. It is "built around the concepts of separation of concerns (making sure people can interact and collaborate on a project, without stepping on each other toes) and component-based Web development" [3]. It considers four areas of concern: *management*, *logic*, *content*, and *style*. Again, we focus on programming and design, roughly corresponding to logic and style, respectively.

Apache Velocity is a Java-based template engine, which also follows the MVC approach and aims for separating Java code from HTML templates. With Velocity, "Web page designers can focus solely on creating a site that looks good, and programmers can focus solely on writing top-notch code. Velocity separates Java code from the Web pages, making the web site more maintainable over the long run" [26]. Its core is a specialization of Servlets combined with Velocity Template Language (VTL), a scripting language used by the page designers and content providers as a simple and more managable alternative to JSP. Velocity's design concept is borrowed from the WebMacro system [27]. Yet another alternative to JSP is FreeMarker [12], which uses a more expressive template language than VTL for constructing HTML pages from templates and Java objects. As the previously mentioned systems, these ones do not provide static guarantees that the separation of code and design is consistent, in contrast to the system we propose.

Other related projects are the Barracuda [10] and Apache Turbine [23] Web application frameworks, which also promise separation of HTML design and programming, but without static guarantees of consistency.

When distinguishing between the programming and HTML design tasks, we apply a slightly different viewpoint than some of these projects: We consider the program code that deals with the construction of complete HTML pages from smaller templates as a part of the programmer's tasks; for example in Velocity, this is instead specified by the page designers using the VTL language.

The template mechanism of JWIG that we build upon originates from MAWL [2], which permitted a complete separation of HTML and code, but no contracts or static guarantees of conformance.

Another approach for building Web sites is to use a Web-oriented *content management system* (CMS). This is a tool that enables non-technical staff to manage and publish content on the Web, under the contraints of centralized rules that specify workflows and Web site appearance. A CMS typically supports a strict separation of content and presentation. In contrast, we focus on the development of more specialized Web services, which do not fit into the CMS model of configuring a Web site by combining shrink-wrapped software components.

Although the general issue of separation of concerns in Web service development is being considered extensively by developers of Web service frameworks, no existing technique solves the problems that we are attacking in this paper: that programmers and designers should be able to work independently and also to safely combine the results of their work.

3 Programming with XML Templates

Dynamic XHTML document are often generated by printing to an output stream, as exemplified by Servlets. The MAWL project invented a more structured approach by defining XML *templates* as primitive values in a Web programming language [14, 2]. A template in MAWL is a wellformed XML fragment containing named *gaps* which at runtime may be *plugged* with string values. The `<bigwig>` and JWIG projects [5, 8] extend this concept to *higher-order* templates, in which gaps may also be plugged with other templates. The following is an example of an XML template:

`members/Outer.xml:`

```
<html>
  <head><title><[title]></title></head>
  <body bgcolor=[color]>
    <[body]>
  </body>
</html>
```

It contains two gaps named `title` and `body`, and one *attribute gap* named `color`. Attribute gaps may only be plugged with string values, while ordinary gaps may be plugged with both strings and templates. The **`plug`** operation

```
x.plug(g,y)
```

creates a copy of `x` where all `g` gaps are replaced by copies of `y`. Note that neither `x` nor `y` are modified, that is, XML templates are *immutable*. Both `x` and `y` may contain other gaps, which can be plugged later. Templates are stored in separate files in a directory structure. A given template is referenced in the Java code using syntax such as:

```
[members.Outer]
```

when the template is in the file named `members/Outer.xml`. An XML template is presented to the client using the syntax:

```
show x;
```

which removes all remaining gaps and produces a textual representation of the resulting document. The current Java thread is suspended until the client submits the form that is typically contained. Subsequently, the string valued expression:

```
receive f
```

returns the value of the field named `f` from the last form that was shown to the client. If the field has multiple occurrences, then the variant **`receive`**`[] f` is used which returns an array of strings.

The template operations may cause runtime exceptions: a `TemplateException` is thrown whenever a referenced template is non-existing or not wellformed; a `ShowException` is thrown whenever a document being shown is not valid XHTML 1.0; and a `ReceiveException` is thrown whenever a form field value is not available when it is being received with **`receive`**.

Consider now the task of presenting a list of names stored as an array of strings:

```
String[] names = {"John Doe",
                  "Joe Blow",
                  "John Q. Citizen"};
```

For this, we need two further templates:

`members/List.xml`:

```
<ul>
  <[items]>
</ul>
```

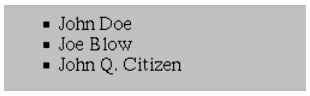
`members/Item.xml`:

```
<li type=[type]>
  <[name]>
</li>
<[items]>
```

The code constructing the desired presentation looks as follows:

```
int i;
XML x = [members.Outer].plug(body,[members.List]);
x = x.plug(title,"Average Guys");
for (i=0; i<names.length; i++)
  x = x.plug(items,[members.Item].plug(name,names[i]));
show x.plug(bgcolor,"silver").plug(type,"square");
```

and yields the result:



Note that plugging also works for embedded gaps and that multiple occurrences of a gap each are plugged with the same value.

A naive implementation of templates would copy the involved templates during a plug operation, yielding an unacceptable time complexity of the above tiny program. However, the data structure underlying the JWIG implementation [8] performs each **`plug`** operation in amortized constant time and each **`show`** operation in linear time in the size of the resulting document. As an added benefit, XML documents are constructed to internally share as many common parts as possible.

JWIG is implemented as a Java framework, an Apache module, and a *desugarer* which transform the special syntax into ordinary Java method invocations.

4 A Contract Language

With the JWIG system, the programmers and the HTML designers can work independently since the code and the HTML templates are completely separated and reside in distinct files. However, without further support, the same problems may arise as with the custom tag library approach and the related approaches mentioned in previous sections: *the decoupling has become too complete*. We need a simple mechanism that enforces the code and the HTML templates to maintain coherency without being too rigid.

We now introduce a formal language for specifying *contracts* between the programmers and the HTML designers. A contract is an interface between code and HTML templates that specifies abstract properties of the HTML templates and their interrelationships. First, we focus on the syntax of contracts and their intuitive meaning, then later we explain the semantics in terms of obligations of the programmers and the HTML designers for fulfilling a contract.

A contract is comprised of **`template`** declarations corresponding to the HTML templates being used. Each **`template`** declaration has a name and contains a description of the gaps, forms, and input fields in the corresponding template:

```
template TemplateName { ... }
```

A **`gap`** descriptor in a template declaration specifies that the corresponding template has at least one gap of that name:

```
gap GapName : GapPresence TemplateNameSet RegExp
```

A *`TemplateNameSet`* is either a single template name or a set of such names enclosed by `{... }`. The *`TemplateNameSet`* of a gap descriptor specifies the possible templates that may be plugged into such gaps. The *`RegExp`* part specifies which strings that may be plugged into the corresponding gaps. A *`RegExp`* is a regular expression over the Unicode alphabet using the syntax from the BRICS Automaton package [18] as summarized in Figure 1. The *`GapPresence`* is either empty or "?": in the former case, the gaps *must* be plugged with one of the allowed values, whereas in the latter case, the gaps *may* additionally be left open.

Operation	Meaning
"s"	the string s
.	any character
rs	concatenation
r|s	union
r&s	intersection
~r	complement
r?	optional
r*	zero or more occurrences
r+	one or more occurrences
r{n}	n occurrences
r{n,}	n or more occurrences
r{n,m}	n to m occurrences
[c]	character class (e.g, [A-Z])
[^c]	negated character class
(r)	grouping
<id>	named regular language

Figure 1: Syntax for regular expressions in contracts.

The part *RegExp* may be omitted from a gap descriptor, in which case no strings may be plugged in. Conversely, if the *TemplateNameSet* is omitted, no templates may be plugged in. The latter case is particularly useful for describing attribute gaps, which are not allowed to contain markup. The *TemplateNameSet* and *RegExp* may both be omitted, which is useful in combination with the **page** construct explained below.

Commonly used regular expressions can be named by **regexp** declarations:

```
regexp RegexpName : RegExp
```

The name *RegexpName* can then be used in other regular expressions using the notation <*RegexpName*>. For example, a "length" in HTML 4.01 [20] can be expressed as follows:

```
regexp LENGTH: [0-9]+"%"?
```

The syntax of URLs is another example that fits into this formalism. The name `AnyString` is predefined to denote the regular expression "`.*`". A **form** descriptor denotes an HTML form:

```
form FormName { ... }
```

which may contain **gap** and **field** descriptors. The *FormName*, which is optional, specifies the name of the form (that is, the `name` attribute). A **field** descriptor describes a family of form fields:

```
field FieldName : FieldCardinality
```

where *FieldCardinality* is either empty, meaning *one occurrence*, "*" meaning *any number*, or "#", which represents a radio button that has the special operational meaning in HTML that a form may contain multiple radio buttons of the same name but only one name/value pair is produced. (Thus, *FieldCardinality* corresponds to the field occurrence lattice in the JWIG article [8], Section 5.2.) The order of **gap**, **field**, and **form** descriptors within a **template** rule is insignificant. A **page** declaration describes an HTML page:

```
page PageName : TemplateNameSet { ... }
```

Such declarations act as entry points into the contract. The *TemplateNameSet* specifies the possible "outermost" templates. The contents of `{...}` are additional gap descriptors that are applicable only to this specific page and extend the existing ones. Each gap name of such an additional gap descriptor must be prefixed with the name of the relevant template. The additional gap descriptors are implicitly merged with the existing ones by taking the unions of the respective template name sets and regular expressions. This mechanism allows different pages of a Web service to be built from the same constituents but with different structures.

Declarations are organized in a hierarchical structure, typically a file system, similarly to Java source files. The root of the hierarchy is called the *contract root directory*. Each file occurring below the contract root directory must contain a **package** declaration:

```
package Package
```

where *Package* is the path to the directory containing the file using "`.`" to separate the components of the path. A file placed in a directory `shop/inventory` should then begin with:

```
package shop.inventory
```

The names of the contract files are insignificant, and it makes no difference whether a set of declarations from the same package are split into multiple files or combined in one file.

All references to **template**, **regexp**, and **page** declarations placed in another package must be qualified according to the package names. For example, if a template named `T` is declared in a directory `a/b/c` relative to the contract root directory, then this template can be referred to as `a.b.c.T`. Similarly to Java, **import** declarations can be used to allow shorter references to such names: If a file contains:

```
import Package
```

then declarations in *Package* can be referred to from this file without the package qualification. The package corresponding to the directory containing the file is always implicitly imported. Ambiguous references are not allowed.

Additionally, Java-style comments (`//...` and `/*...*/`) are allowed throughout the contracts.

```
package shop.inventory
template Wrapper { gap title: <AnyString>,
                   gap body }

template Main { gap inventory: Inventory,
                gap buyform: Selection }

template Inventory { gap items: ? Product }

template Product { gap productname: <AnyString>,
                   gap price: <AnyString>,
                   gap items: ? Product }

template Selection { form { field product,
                            field quantity } }

page MainPage: Wrapper { gap Wrapper.body: Main }
```

Figure 2: A tiny contract.

The job of the HTML designer is to concretize the template declarations by constructing an actual XML template for each. These XML templates are placed in a subdirectory of a *template root directory*, which has the same subdirectory structure as the contract root directory. For example, an XML template corresponding to a template declaration `T` associated a package `a.b.c` is placed in the file `a/b/c/T.xml`.

Finally, we extend JWIG such that each **`show`** statement is annotated with an appropriate reference into the contract:

```
show H as P;
```

The argument *P* refers to a **`page`** declaration that describes the required structure of the HTML page *H* being shown. For example, *P* could be `shop.inventory.MainPage`, which is interpreted relative to some contract root directory. At runtime, the program uses the XML templates below the template root directory, which is specified by a compiler parameter.

4.1 An Example Contract

Assume that we as a part of a Web service want to produce an HTML page with an inventory of products and a form where a product and a quantity can be selected. Figure 2 shows a tiny but realistic example of a contract for such a page. The `Wrapper` template is intended as the outermost template and for reuse in other pages, `Main` corresponds to the body of the page, `Inventory` represents the product inventory, `Product` represents a single product, and `Selection` is the form. Note that the `items` loop in `Product` permits an inventory to contain any number of products. Finally, the `MainPage` page declaration combines the templates using `Wrapper` as entry point and adding an edge from its `body` gap to the `Main` template. These declarations are placed in a directory `shop/inventory` below the contract root directory.

The HTML designer can now create the concrete XML templates. The following templates illustrate one possible design:

`shop/inventory/Wrapper.xml`:

```
<html>
 <head><title><[title]></title></head>
 <body><[body]></body>
</html>
```

`shop/inventory/Main.xml`:

```
<h1>Inventory</h1>
<[inventory]> <p/>
<[buyform]>
```

`shop/inventory/Inventory.xml`:

```
<table border="1">
 <tr> <th>Name</th> <th>Price</th> </tr>
 <[items]>
</table>
```

`shop/inventory/Product.xml`:

```
<tr>
 <td><[productname]></td>
 <td align="right">$<[price]></td>
</tr>
<[items]>
```

`shop/inventory/Select.xml`:

```
<form>
 Product: <input name="product"/> <br/>
 Quantity: <input name="quantity"/> <br/>
 <input type="submit" value="Buy!"/>
</form>
```

These files are placed in the directory `shop/inventory` below the template root directory.

In this design, the inventory is shown as a table with a header and one row for each product. The JWIG system will fill in the `action` attribute for the `form` element. The designer can change the design by modifying the templates. For example, the inventory can be shown as an itemized list instead, simply by replacing the `Inventory` and `Product` templates:

`shop/inventory/Inventory.xml`:

```
<ul>
 <[items]>
</ul>
```

`shop/inventory/Product.xml`:

Figure 3: Three designs for the `InventoryPage` example.

```
<li> <[productname]>: $<[price]> </li>
<[items]>
```

Or, perhaps the designer instead wants the table rows to be colored, such that even and odd rows have different background color. This can be achieved by adding a little JavaScript to the `Wrapper` template:

`shop/inventory/Wrapper.xml:`

```
<html><head><title><[title]></title>
  <script type="text/javascript"> <!--
   function fancycolors() {
     var t = document.getElementsByTagName('tr');
     for (var i = 1; i<t.length; i++)
       t[i].style.backgroundColor =
          i%2==0 ? "gray" : "silver";
   } // -->
  </script>
</head>
<body onload="javascript:fancycolors();">
  <[body]>
</body></html>
```

This illustrates that the designer may apply JavaScript programming to obtain full control of the layout. Of course, the designer should only use JavaScript to change stylesheet properties, not to modify the underlying DOM tree.

The three designs can be seen in Figure 3. These examples show one of the benefits of using contracts:

> *The HTML designer has a large degree of freedom in deciding the design of the pages without involving the programmer or worrying that the service will break because of the modifications.*

In fact, modifications of the design may be performed while the service is running, without any downtime or recompilation—the running service automatically uses the newest family of templates.

Independently of the designer, the programmer can develop the code that produces the final pages from the templates, accesses a database, interacts with the client, etc. Due to the limited space, we will not show an example of such code; it should be clear how the example from Section 3 could be used as a starting point for solving this task.

4.2 Obligations of the HTML Designer

The semantics of a contract is divided into two parts corresponding to the obligations of the HTML designer and the programmer, respectively. We first consider the HTML designer:

§1. The designer must build a template directory structure matching the contract directory structure, such that there is one file `a/b/c/T.xml` containing a well-formed XML template for each template `T` with package `a.b.c` in the contract.

§2. Each template must contain the right occurrences of gaps, fields, and forms. This means that a gap named *g* must occur at least once in the template if and only if the corresponding **`template`** declaration contains a **`gap`** descriptor with name *g*. Similarly, input fields must occur according to the **`field`** descriptors and their associated *FieldCardinality*, and the template must contain one `form` for each **`form`** descriptor, such that the *FormName* matches the `name` attribute, if present.

Recall that **`gap`** and **`field`** descriptors are allowed both at the same level as **`form`** descriptors and nested within them. The gaps and fields in the XML templates must adhere to the nesting structure used in the **`template`** declaration.

§3. All pages that can be derived from the contract and the concrete templates must be valid according to the XHTML syntax.

The set of pages $\mathcal{L}(P, C, T)$ that can be *derived* from a page description P of a contract C and concrete templates T is defined as those pages that can be obtained by starting with a root template of P and recursively plugging all possible templates and strings into the gaps according to the contract. For example, if `contractroot` and `templateroot` denote the contract root directory and the template root directory, respectively, using the first design from the example from Section 4.1, then

$$\mathcal{L}(\texttt{shop.inventory.MainPage}, \texttt{contractroot}, \texttt{templateroot})$$

contains the HTML code for the first page in Figure 3, together with the infinitely many other pages that have a `Wrapper` template outermost, a `Main` template plugged into its `body` gap, etc.

To define validity, we use the DSD2 schema language [17] description of XHTML 1.0 Transitional; we could instead use any XML language that can be described by DSD2 or DTD. The DSD2 language has the benefit that it allows more syntactic requirements to be formally captured than DTD or even XML Schema. For example, the DSD2 description of XHTML expresses the facts that `form` elements cannot be nested, that `input` elements must have a `name` attribute unless the value of `type` is `submit` or reset, and that the values of color attributes must be valid color descriptions. The full DSD2 schema is available at http://www.brics.dk/DSD/xhtml1-transitional.dsd.

The templates shown in the example in Section 4.1 can be seen to fulfill the designer obligations induced by the contract in Figure 2. It is generally far from trivial to verify that the validity requirement is satisfied; in Section 5.2 we present an automatic and conservatively approximating approach.

4.3 Obligations of the Programmer

To fulfill a given contract, the programmer is obliged to:

§4. only use the templates that are declared in the contract by **template** declarations;

§5. only show pages whose structure fulfills the contract, that is, for every instruction "**show** *H* **as** *P*", the structure of the page *H* must match the **page** declaration for *P*; and

§6. only receive form fields that are present according to the **field** descriptors in the contract for the latest page that has been shown.

A page *H* that has been built from templates and strings using plug operations *matches* a **page** declaration for *P* if the following conditions are satisfied:

- The outermost template in *H* is permitted as an outermost template according to *P*.
- For each template or string y that has been plugged into a gap g in a template x during the construction of *H*, there is a corresponding **gap** descriptor in the contract. This **gap** descriptor can be located either in the **template** declaration of x or in the **page** declaration of *P*, as described in Section 4.

The requirement §6 only applies to the singleton version of **receive**, since the array variant **receive[]** cannot fail.

Note that the programmer in principle does not need the designer's concrete XHTML templates until deployment of the Web service. From a contract C we can automatically derive a set of *dummy templates* $T(C)$, which represent a naive but valid design. For every **template** declaration, an XHTML template is created containing a table with a row for each **gap** descriptor, a form for each **form** descriptor, and an input field for each **field** descriptor. The templates that occur as roots in some **page** declaration are enclosed by:

```
<html><head><title/></head><body>...</body></html>
```

As an example, the dummy template for shop.inventory.Main is the following:

```
<table border="1">
  <tr><th colspan="2">shop.inventory.Main</th></tr>
  <tr><td>inventory:</td><td><[inventory]></td></tr>
  <tr><td>buyform:</td><td><[buyform]></td></tr>
</table>
```

As for the designer's obligations mentioned in the previous section, it is also not trivial to verify that the programmer fulfills his parts of the contract. Again, we suggest a fully automatic conservative approximation, which we explain in Section 5.3.

4.4 Renegotiating Contracts

As the example in Section 4.1 indicates, the designer does have a significant degree of freedom in his work without being required to consider the program code. Similarly, the programmer is free to restructure the program code, as long as the structure of the pages being built from the templates fulfills the contract.

Of course, the designer may want to perform more profound changes in the page structures that inevitably involve modifying the program code also. Or, conversely, the programmer may want to modify the functionality of the service, which, for example, may require adding new gaps or form fields to the designer's templates. This leads to another benefit of using contracts:

> *Contracts make it explicit exactly when it is required that the HTML designer and the programmer talk together, in other words, renegotiate the contract.*

In many typical cases where the contract inevitably needs to be changed, doing so is not problematic, because the existing contract can help pinpointing where the changes are required.

4.5 Soundness

If the contract is fulfilled by both parties, then some important guarantees can be issued about the dynamic behavior of the service: Neither `TemplateExceptions`, `ReceiveExceptions`, nor `ShowExceptions` can occur. The proof is trivial: a `TemplateException` occurs if the programmer attempts to use a nonexisting or not wellformed template, but this is made impossible by §1 and §4; a `ReceiveException` occurs if the programmer tries to **`receive`** a nonexisting form field, which is avoided by the combination of §2, §5, and §6; and a `ShowException` occurs if invalid XHTML 1.0 appears at a **`show`** statement, but this is avoided by §2, §3, and §5. This shows yet another benefit of the contracts:

> *The contract defines the interface between HTML design and programming; if both the designer and the programmer fulfill their respective parts of the contract, then their work can safely be combined.*

5 Checking Contracts Automatically

Building on the JWIG system, we are in a unique position to automatically check that the obligations required by a contract are fulfilled by both the designer and the programmer. This allows us to obtain strong guarantees about the behavior of the running

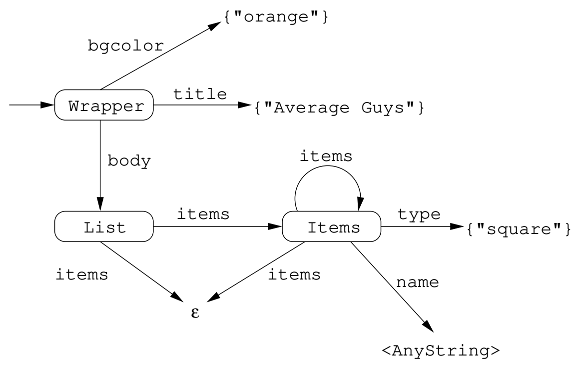


Figure 4: A summary graph.

Web service even before it is launched. Furthermore, the designer and the programmer may independently check their own work, since the contract serves as the crucial interface that combines the indvidual checks to provide a global guarantee.

5.1 Summary Graphs in JWIG

The JWIG system performs static analysis of the class files that implement Web services. For each occurrence of an XML expressions, the analyzer determines the set of XML documents that may result from evaluation at runtime. Such a set is described as a *summary graph*, which is a finite representation of a possibly infinite set of documents. Since the problem of finding the exact solution is clearly undecidable, the analysis is conservatively approximate, so the true set of XML documents may be smaller than that corresponding to the inferred summary graph.

The notion of summary graphs is formally defined in [8]. Informally, the *nodes* of a summary graph are templates being used in the program. The *edges* of the graph corresponds to possible plug operations that have been performed. There is an edge labeled g from a node t_1 to a node t_2 if the template t_1 contains a gap named g which may have been plugged with an XML document whose outermost template is t_2. If gaps are plugged with strings, then the corresponding edges lead to regular expressions describing the possible string values. These regular expressions are inferred by a separate static analysis [9]. Some of the nodes in a summary graph are identified as *roots*, meaning that they correspond to the outermost templates of the documents described. Finally, a summary graph notes for every template whether a given gap is possibly still open or has definitely been plugged with one of the templates corresponding to the given edges.

The *language* $\mathcal{L}(G)$ of a summary graph G is the set of XML documents that is represents. Intuitively it contains those documents that are obtained by starting at a root and following all possible choices of plug operations permitted by the edges.

For the small example in Section 3, the summary graph for the XML expression being shown is inferred to be the one in Figure 4, where an edge to "ε" means that the gap may remain open. For realistic applications, a summary graph may contain hundreds of nodes and edges.

Using the algorithms presented in [8], the JWIG system is able to analyze a summary graph and determine if all documents contained in its language are valid according to a given DSD2 schema, in particular the one describing XHTML 1.0. The analyzer is also able to determine that any field occurring in a **`receive`** expression is guaranteed to be present in all documents that may have been sent to the client at the immediately preceding **`show`** statement. In the present work, we heavily exploit these earlier results. The analyses in JWIG are technically challenging; they handle all aspects of the Java language, and have been carefully engineered to achieve high efficiency and precision.

5.2 Checking the Designer's XHTML

When the designer has created a template directory T, it may be checked against a given contract directory C. First, some simple *local* checks are performed:

- corresponding to §1, the template directory must have the same structure as the contract directory, and each template file must contain a wellformed XML document; and
- corresponding to §2, each template file must contain the gaps, forms, and fields that are mandated by the contract.

What remains is the *global* check that all pages P conform to the XHTML 1.0 specification, corresponding to §3. This is done by constructing a summary graph $G_{P,C,T}$ such that

$$\mathcal{L}(G_{P,C,T}) = \mathcal{L}(P, C, T)$$

and using JWIG to check validity against the appropriate DSD2 schema. The construction is straightforward, since contracts are essentially abstract summary graphs. We simply use the designer's templates as summary graph nodes, the edges and roots are read directly from the contract, and gaps that are marked by "?" in the contract become potentially open in the summary graph.

5.3 Checking the Programmer's Code

For a given program, we first check that each template referred from the program is also mentioned in the contract directory, corresponding to §4. The remaining checks require the construction of summary graphs for each expression in **`show`** statements. These are obtained by constructing the dummy template directory $T(C)$, as described in Section 4.3, and then running the JWIG analyzer on this structure. For every statement of the form:

`show` H **`as`** P;

this provides a summary graph G_H. To check §5, we inspect that P occurs as a page in the contract, construct a summary graph $G_{P,C,T(C)}$ as in Section 5.2 but using the dummy templates, and then check that

$$G_H \subseteq G_{P,C,T(C)}$$

Inclusion on summary graphs is determined on the set of edges, where potentially open gaps are viewed as edges to a special "ε" template, and nodes representing string sets are compared by set inclusion.

Finally, the check of fields in **`receive`** expressions, §6, is simply performed by the JWIG analyzer using the algorithm described in [8].

5.4 Soundness

When the checks described above have been performed, then we are guaranteed that the soundness requirements described in Section 4.5 are satisfied, since our analyses are conservative. Ultimately, this works to ensure that the exceptions thrown by the template operations will never occur. Importantly, the two checks of templates and code are *independent*, which means that the designer and the programmer are free to work on their own, only bound by the limitations of the contract.

Our analyses are of course approximative, which means that they may unfairly reject programs for which no exceptions would actually be thrown during runtime. However, experiences from the JWIG project [8, 9] indicate that the precision is sufficient for practical use. The analyses are also efficient, handling large programs in mere seconds.

6 Additional Tool Support

There are several other opportunities for providing tool support for both the programmer and the designer.

6.1 Viewing Contracts Graphically

Contracts are defined as a directory structure of text files. While this is a concise formalism, it may at times be easier to browse through a graphical representation. This is easily obtained by depicting what is essentially the corresponding summary graph. Using the AT&T Graphviz `dot` tool [11], the contract from Section 4.1 is presented as seen in Figure 5. The edges labeled "MainPage" are applicable only to the `shop.inventory.MainPage` page, whereas the others are also applicable to other pages, as defined in the contract.

6.2 A Dummy Designer

We have earlier shown how a contract C gives rise to a dummy template directory $T(C)$. For a programmer starting out with an already negotiated contract, this approach can be exploited to construct a collection of primitive but functional templates that are guaranteed to satisfy the contract.

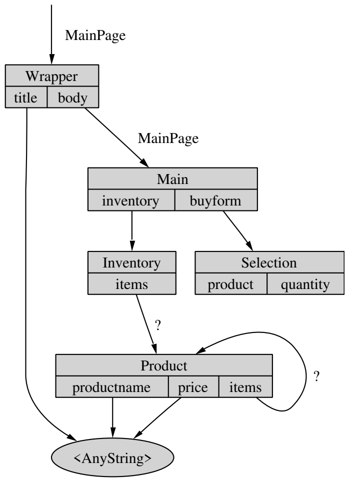


Figure 5: Graphical view of the example contract.

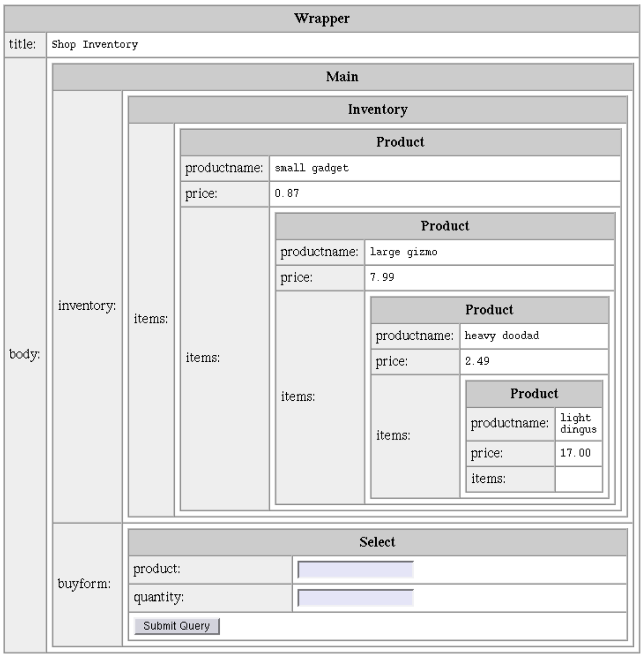


Figure 6: X-ray view of `MainPage`.

Starting out with the contract from Section 4.1, the generated dummy templates present an *X-ray view* of the `MainPage` as shown in Figure 6. For the programmer, this allows an early implementation suitable for testing the business logic. For the designer, it offers an explicit view at the template structure of the generated pages.

6.3 A Dummy Programmer

When the designer has produced a draft set of templates, it is possible to view complete sample documents, even when the programmer has not delivered the code yet. This is done by generating random elements from the set $\mathcal{L}(P, C, T)$ for a given page P. In case of loops in the contract, some care must be taken to generate only finite documents. Our strategy uses a depth-first search of the contract graph to identify the *back edges* [1]. The designer specifies a *scale factor* k, and when unfoldings of the graph are generated, a given back edge is allowed to be traversed at most k times. If a gap g does not have an outgoing edge, then the text `<[`g`]>` is generated. For strings, random elements of the specified regular language are chosen.

6.4 Inferring Contracts

If an existing JWIG service is to be refactored into using contracts, then substantial automatic support is also available. It is possible to *infer* a contract that is satisfied by the given code and XHTML. The template directory structure is copied to form the contract directory. Each template is described in the contract with respect to the existing gaps, forms, and fields. The rest of the contract is inferred from the summary graphs computed by the analyzer. Each **`show`** statement is represented by a **`page`** in the contract. The edges in all summary graphs are distributed into either the templates or the pages in the contract. If an edge is present in all the summary graphs corresponding to **`show`** statements, then it is represented directly in that template, otherwise it is represented in only the pages corresponding to the relevant **`show`** statements. For the `shop.inventory` code, the inferred contract is identical to the given one, which indicates that the accuracy of this inference algorithm is high.

6.5 Managing Templates

For a realistic application, the designer may have to consider a great many templates. Again, automatic tool support may help. In a special mode, the JWIG service may generate XHTML pages enriched with JavaScript such that a mouse click on a point in a generated page will identify the templates involved in that part of the presentation. A screenshot of this feature is presented in Figure 7.

7 Implementation and Experiments

The contract mechanism we have described has been implemented in a prototype extension of the JWIG system. This primarily consists of a syntax checker for contracts and tools for checking that the obligations of the contract are fulfilled by the designer

Figure 7: Identifying the `shop.inventory.Product` template.

and the programmer, as described in Section 5. Most of the tools described in Section 6 have also been prototyped. The implementations have been fairly straightforward, since the most complicated parts wholly rely on the exisiting JWIG analyses.

Checking contracts is fairly efficient. We are in the process of converting existing JWIG services to use the contract system. A large example is the JAOO 2002 conference administration system, which consists of 3,923 lines of pure Java code and 198 XHTML templates totalling 259K data. Checking the obligations of the designer or the programmer takes less than 30 seconds for each.

To test the contract methodology, we have developed a small service from scratch. The example is a typical Web shop, called *The Plant Store*, selling house plants and tracking the buying histories of customers. Although this is just a single case study, we learned some valuable lessons:

- it makes sense for both the programmer and the designer to take the lead when drafting the contract;
- benign renegotiations are frequent in the early stages of development;
- the use of dummy templates and the graphical view of contracts are useful supplements to the text of the contract;
- the X-ray view of generated pages is invaluable for the programmer, allowing early prototyping without design distractions;
- it is a simple task to generate alternate presentations of the pages once they have been designed, which is particularly useful for internationalization and customization.

In all, the contract system seems to hit a sweet spot between complete decoupling and rigidity.

8 Conclusion

By introducing our contract language, we improve collaboration between HTML designers and programmers jointly developing Web services. A contract formalizes the interface between the two parties and pinpoint the dependencies between design and code.

We work in the context of the JWIG system, which already provides a well-structured and flexible mechanism for building Web pages from XML pages together with unique program analyses for ensuring that the templates are used consistently. The obligations incurred by the contract can be automatically checked by tools that exploit these analyses. On top of the contract lanaguage, we offer a host of tools facilitating the use of XML templates and contracts.

Our experiments demonstrate that the contract system can be implemented efficiently and is practically useful.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, January 1983.

[2] David Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. Mawl: a domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, May/June 1999.

[3] Nicola Ken Barozzi et al. Cocoon, 2003. `http://cocoon.apache.org/`.

[4] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, October 1999.

[5] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (second edition), October 2000. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.

[7] Aske Simon Christensen and Anders Møller. *JWIG User Manual*. BRICS, Department of Computer Science, University of Aarhus, June 2002. Notes Series NS-02-6. Available from `http://www.brics.dk/JWIG/manual/`.

[8] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.

[9] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[10] Christian Cryder et al. Barracuda, 2003. `http://barracudamvc.org/`.

[11] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with *dot*, February 2002. Available from `http://www.research.att.com/sw/tools/graphviz/`.

[12] Benjamin Geer, Mike Bayer, et al. FreeMarker, 2003. `http://freemarker.sourceforge.net/`.

[13] G. Krasner and S. Pope. A cookbook for using the model view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

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

[15] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Pearson Education, July 2001.

[16] Craig R. McClanahan et al. Struts, 2002. `http://jakarta.apache.org/struts/`.

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

[18] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2001. `http://www.brics.dk/automaton/`.

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

[20] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification, December 1999. W3C Recommendation. `http://www.w3.org/TR/html4/`.

[21] Gal Schachor, Adam Chace, and Magnus Rydin. *JSP Tag Libraries*. Manning Publications, May 2001.

[22] Govind Seshadri. Understanding JavaServer Pages Model 2 architecture, December 1999. Available from `http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html`.

[23] Jon S. Stevens et al. Turbine, 2003. `http://jakarta.apache.org/turbine/`.

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

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

[26] Jason van Zyl et al. Velocity, 2003. `http://jakarta.apache.org/velocity`.

[27] Justin Wells et al. WebMacro, 2003. `http://www.webmacro.org/`.

Recent BRICS Report Series Publications

RS-03-38 Henning Böttger, Anders Møller, and Michael I. Schwartzbach. ***Contracts for Cooperation between Web Service Programmers and HTML Designers*****. November 2003. 23 pp.**

RS-03-37 Claude Crépeau, Paul Dumais, Dominic Mayers, and Louis Salvail. ***Computational Collapse of Quantum State with Application to Oblivious Transfer*****. November 2003.**

RS-03-36 Ivan B. Damgård, Serge Fehr, Kirill Morozov, and Louis Salvail. ***Unfair Noisy Channels and Oblivious Transfer*****. November 2003.**

RS-03-35 Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. ***A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*****. November 2003. 31 pp.**

RS-03-34 Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Bas Luttik. ***CCS with Hennessy's Merge has no Finite Equational Axiomatization*****. November 2003. 37 pp.**

RS-03-33 Olivier Danvy. ***A Rational Deconstruction of Landin's SECD Machine*****. October 2003. 32 pp. This report supersedes the earlier BRICS report RS-02-53.**

RS-03-32 Philipp Gerhardy and Ulrich Kohlenbach. ***Extracting Herbrand Disjunctions by Functional Interpretation*****. October 2003. 17 pp.**

RS-03-31 Stephen Lack and Paweł Sobociński. ***Adhesive Categories*****. October 2003. 25 pp.**

RS-03-30 Jesper Makholm Byskov, Bolette Ammitzbøll Madsen, and Bjarke Skjernaa. ***New Algorithms for Exact Satisfiability*****. October 2003. 31 pp.**

RS-03-29 Aske Simon Christensen, Christian Kirkegaard, and Anders Møller. ***A Runtime System for XML Transformations in Java*****. October 2003. 15 pp.**

RS-03-28 Zoltán Ésik and Kim G. Larsen. ***Regular Languages Definable by Lindström Quantifiers*****. August 2003. 82 pp. This report supersedes the earlier BRICS report RS-02-20.**