BRICS

Basic Research in Computer Science

BRICS RS-00-43 Brabrand et al.: PowerForms: Declarative Client-Side Form Field Validation

PowerForms: Declarative Client-Side Form Field Validation

Claus Brabrand
Anders Møller
Mikkel Christensen, Ricky
Michael I. Schwartzbach

BRICS Report Series RS-00-43

ISSN 0909-0878 December 2000

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/00/43/`

PowerForms:
Declarative Client-Side Form Field Validation

Claus Brabrand   Anders Møller   Mikkel Ricky
Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark
{brabrand,amoeller,ricky,mis}@brics.dk

Abstract

All uses of HTML forms may benefit from validation of the specified input field values. Simple validation matches individual values against specified formats, while more advanced validation may involve interdependencies of form fields.

There is currently no standard for specifying or implementing such validation. Today, CGI programmers often use Perl libraries for simple server-side validation or program customized JavaScript solutions for client-side validation.

We present PowerForms, which is an add-on to HTML forms that allows a purely declarative specification of input formats and sophisticated interdependencies of form fields. While our work may be seen as inspiration for a future extension of HTML, it is also available for CGI programmers today through a preprocessor that translates a PowerForms document into a combination of standard HTML and JavaScript that works on all combinations of platforms and browsers.

The definitions of PowerForms formats are syntactically disjoint from the form itself, which allows a modular development where the form is perhaps automatically generated by other tools and the formats and interdependencies are added separately.

PowerForms has a clean semantics defined through a fixed-point process that resolves the interdependencies between all field values. Text fields are equipped with status icons (by default traffic lights) that continuously reflect the validity of the text that has been entered so far, thus providing immediate feed-back for the user. For other GUI components the available options are dynamically filtered to present only the allowed values.

PowerForms are integrated into the `<bigwig>` system for generating interactive Web services, but is also freely available in an Open Source distribution as a stand-alone package.

1 Introduction

We briefly review some relevant aspects of HTML forms. The CGI protocol enables Web services to receive input from clients through forms embedded in HTML pages. An HTML form is comprised of a number of input fields each prompting the client for information.

The visual rendering of an input field and how to enter the information it requests is determined by its type. The most widely used fields range from expecting lines of textual input to providing choices between a number of fixed options that were determined at the time the page was constructed. Many of the fields only differ in appearance and are indistinguishable to the server in the sense that they return the same kind of information. Fields of type `text` and `password`, although rendered differently, each expect one line of textual input from the client. Multiple lines of textual input can be handled through the `textarea` field. The fields of types `radio` and `select` both require exactly one choice between a number of static options, whereas an arbitrary number of choices are permitted by the `checkbox` and `select` (`multiple`) fields. Individual `radio` and `checkbox` fields with common name may be distributed about the form and constitute a group for which the selection requirements apply. The options of a `select` field, on the other hand, are grouped together in one place in the form. In addition, there are the more specialized fields, `image`, `file`, `button`, and `hidden`, which we shall not treat in detail. Finally, two fields control the behavior of the entire form, namely `reset` and `submit`, which respectively resets the form to its initial state and submits its contents to the server.

Input validation

Textual input fields could possibly hold anything. Usually, the client is expected to enter data of a particular form, for instance a number, a name, a ZIP-code, or an e-mail address. The most frequent solution is to determine on the server whether the submitted data has the required form, which is known as *server-side input validation*. If some data are invalid, then those parts are presented once again along with suitable error messages, allowing the client to make the necessary corrections. This process is repeated until all fields contain appropriate data. This solution is simple, but it has three well-known drawbacks:

- it takes time;
- it causes excess network traffic; and
- it requires explicit server-side programming.

Note that these drawbacks affect all parties involved. The client is clearly annoyed by the extra time incurred by the round-trip to the server for validation, the server by the extra network traffic and "wasted" cycles, and the programmer by the explicit programming necessary for implementing the actual validation and re-showing of the pages. An obvious solution to the first two drawbacks is to move the validation from the server to the client, yielding *client-side input*

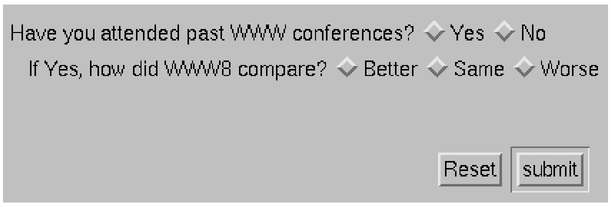


Figure 1: Conference questionnaire.

validation. The third drawback, however, is only partially alleviated. All the details of re-showing pages are no longer required, but the actual validation still needs to be programmed.

The move from server-side to client-side also opens for another important benefit, namely the possibility of performing the validation *incrementally*. The client no longer needs to click the submit button before getting the validation report. This allows errors to be be signalled as they occur, which clearly eases the task of correctly filling out the form.

Field interdependencies

Another aspect of validation involves interdependent fields. Many forms contain fields whose values may be constrained by values entered in other fields. Figure 1 exhibits a simple questionnaire from a conference, in which participants were invited to state whether they have attended past conferences and if so, how this one compared. The second question clearly depends on the first, since it may only be answered if the first answer was positive. Conversely, an answer to the second question may be required if the first answer was "Yes".

Such interdependencies are almost always handled on the server, even if the rest of the validation is addressed on the client-side. The reason is presumably that interdependencies require some tedious and delicate JavaScript code. This kind of validation is explicitly requested in the W3C working draft on extending forms [13]. One could easily imagine more advanced dependencies. Also, it would be useful if illegal selections could somehow automatically be deselected.

JavaScript programming

Traditionally, client-side input validation is implemented in JavaScript. We will argue that this may not be the best choice for most Web authors.

First of all, using a general-purpose programming language for a relatively specific purpose exposes the programmer to many unnecessary details and choices. A small high-level domain-specific language dedicated to input validation would involve only relevant concepts and thus be potentially easier to learn and use. Many assisting libraries exist [9], but must still be used in the context of a full programming language.

Secondly, JavaScript code has an operational form, forcing the programmer to think about the order in which the fields and their contents are validated. However, the simplicity of the input validation task permits the use of a purely *declarative* approach. A declarative specification abstracts away operational details, making programs easier to read, write, and maintain. Also, such an approach is closer to composing HTML than writing JavaScript, making input validation available to more people. As stated in the W3C working draft on extending forms:

> "It should be possible to define a rich form, including validations, dependencies, and basic calculations without the use of a scripting language."

Our solution will precisely include such mechanisms for validations and dependencies.

Finally, the traditional implementation task is further complicated by diverging JavaScript implementations in various browsers. This forces the programmer to stay within the subset of JavaScript that is supported by all browsers—a subset that may be hard to identify. In fact, a number of sites and FAQs are dedicated to identifying this subset [15, 8]. A domain-specific language could be compiled into this common subset of JavaScript, implying that only the compiler writer will be concerned with this issue.

Our solution: PowerForms

As argued above, our solution is to introduce a high-level *declarative* and *domain-specific* language, called PowerForms, designed for incremental input validation.

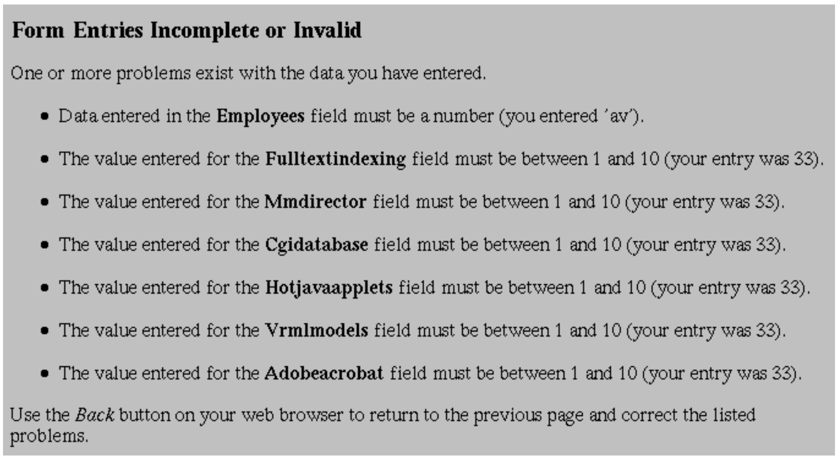
Section 2 presents our solution for simple validation; Section 3 extends this to handle field interdependencies; Section 4 exhibits how other common uses of JavaScript also can be handled through declarative specification; Section 5 presents the overall strategy of the translation to JavaScript; and Section 6 describes the availability of the PowerForms packages.

Related work

Authoring systems like Cold Fusion [6] can automate server-side verification of some simple formats, but even so the result is unsatisfactory. A typical response to invalid data is shown in Figure 2. It refers to the internal names of input fields which are unknown to the client, and the required corrections must be remembered when the form is displayed again.

Active Forms [14] is based on a special browser supporting Form Applets programmed as Tcl scripts. It does not offer high-level abstractions or integration with HTML.

Web Dynamic Forms [7] offer an ambitious and complex solution. They propose a completely new form model that is technically unrelated to HTML and exists entirely within a Java applet. Inside this applet, they allow complicated interaction patterns controlled through an event-based programming model in



Form Entries Incomplete or Invalid

One or more problems exist with the data you have entered.

- Data entered in the **Employees** field must be a number (you entered 'av').
- The value entered for the **Fulltextindexing** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Mmdirector** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Cgidatabase** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Hotjavaapplets** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Vrmlmodels** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Adobeacrobat** field must be between 1 and 10 (your entry was 33).

Use the *Back* button on your web browser to return to the previous page and correct the listed problems.

Figure 2: Typical server-side validation.

which common actions are provided directly and others may be programmed in Java. When a form is submitted, the data are extracted from the applet and treated as ordinary HTML form data. The intervening years have shown that Web authors prefer to use standard HTML forms instead and then program advanced behavior in JavaScript. Thus, our simpler approach of automatically generating this JavaScript code remains relevant. An important reason to stay exclusively with HTML input fields is that they can be integrated into HTML tables to control their layout.

The XHTML-FML language [12] also provides a means for client-side input validation by adding an attribute called `ctype` to textual input fields. However, this attribute is restricted to a (large) set of predefined input validation types and there is no support for field inderdependency.

Our PowerForms notation is totally declarative and requires no programming skills. Furthermore, it is modular in the sense that validation can be added to an input field in an existing HTML form without knowing anything but its name. The validation markup being completely separate from the form markup allows the layout of a form to be redesigned at any time in any HTML editor.

2 Validation of Input Formats

The language is based on regular expressions embedded in HTML that is subsequently translated into a combination of standard HTML and JavaScript. This approach benefits from an efficient implementation through the use of finite-state automata which are interpreted by JavaScript code.

Named formats may be associated to fields whose values are then required to belong to the corresponding regular sets. The client is continuously receiving feedback, and the form can only be submitted when all formats are satisfied. The server should of course perform a double-check, since the JavaScript code is open to tampering.

Regular expressions denoting sets of strings are a simple and familiar formalism for specifying the allowed values of form fields. As we will demonstrate, all reasonable input formats can be captured in this manner. Also, the underlying technology of finite-state automata gives a simple and efficient implementation strategy.

Syntax

We define a rich XML syntax [5] for regular expressions on strings:

```
regexp  →  <const value=stringconst/> |
           <empty/> |
           <anychar/> |
           <anything/> |
           <charset value=stringconst/> |
           <fix low=intconst high=intconst/> |
           <relax low=intconst high=intconst/> |
           <range low=charconst high=charconst/> |
           <intersection> regexp* </intersection> |
           <concat> regexp* </concat> |
           <union> regexp* </union> |
           <star> regexp </star> |
           <plus> regexp </plus> |
           <optional> regexp </optional> |
           <repeat count=intconst> regexp </repeat>
           <repeat low=intconst high=intconst> regexp </repeat>
           <complement> regexp </complement> |
           <regexp exp=stringconst/> |
           <regexp id=stringconst> regexp </regexp> |
           <regexp idref=stringconst/> |
           <regexp uri=stringconst/> |
           <include uri=stringconst/>
```

Here, *regexp*∗ denotes zero or more repetitions of *regexp*. The nonterminals *stringconst*, *intconst*, and *charconst* have the usual meanings.

Note that the verbose XML syntax also allows standard Perl syntax for regular expressions through the construct <regexp exp=*stringconst*/>. Our full syntax is however more general, since it includes intersection, general complementation, import mechanisms, and a richer set of primitive expressions.

A regular expression is associated with a form field through a declaration:

```
formatdecl  →  <format name=stringconst
                       help=stringconst
                       error=stringconst>
                 regexp
               </format>
```

The value of the optional help attribute will appear in the status line of the browser when the field has focus; similarly, the value of the optional error attribute will appear if the field contains invalid data.

The format takes effect for a form field of type type text, password, select, radio, or checkbox whose name is the value of the name attribute. The need for input formats is perhaps only apparent for text and password fields, but we need the full generality later in Section 3.

Semantics of regular expressions

Each regular expression denotes an inductively defined set of strings. The const element denotes the singleton set containing its value. The empty element denotes the empty set. The anychar element denotes the set of all characters. The anything element denotes the set of all strings. The charset denotes the set of characters in its value. The fix element denotes the set of numerals from low to high all padded with leading zeros to have the same length as high. The relax element denotes the set of numerals from low to high. The range element denotes the set of singleton strings obtained from the characters low to high. The intersection element denotes the intersection of the sets denoted by its children. The concat element denotes the concatenation of the sets denoted by its children. The union element denotes the union of the sets denoted by its children. The star element denotes zero or more concatenations of the set denoted by its child. The plus element denotes one or more concatenations of the set denoted by its child. The optional element denotes the union of the set containing the empty string and the set denoted by its child. The repeat element with attribute count denotes a fixed power of the set denoted by its child. The repeat element with attributes low and high denotes the corresponding interval of powers of the set denoted by its child, where low defaults to zero and high to infinity. The complement element denotes the complement of the set denoted by its child. The regexp element with attribute exp denotes the set denotes by its attribute value interpreted as a standard Perl regular expression. The regexp element with attribute id denotes the same set as its child, but in addition names it by the value of id. The regexp element with attribute idref denotes the same set as the regular expression whose name is the value of idref. It is required that each id value is unique throughout the document and that each idref value matches some id value. The regexp element with attribute uri denotes the set recognized by a precompiled automaton. The include element performs a textual insertion of the document denoted by its url attribute.

Semantics of format declarations

The effect on a form field of a regular expression denoting the set S is defined as follows. For a text or password field, the effect is to decorate the field with one of four annotations:

- *green light*, if the current value is a member of S;

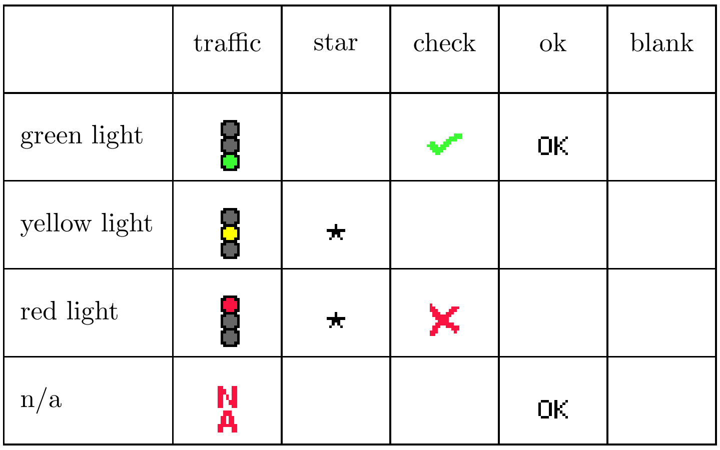


Figure 3: Different styles of status icons.

- *yellow light*, if the current value is a proper prefix of a member of S;
- *red light*, if the current value is not a prefix of a member of a non-empty S; or
- *n/a*, if S is the empty set.

The form cannot be submitted if it has a yellow or red light. The default annotations, which are placed immediately to the right of the field, are tiny icons inspired by traffic lights, but they can be customized with arbitrary images to obtain a different look and feel as indicated in Figure 3. Other annotations, like colorings of the input fields, would also seem reasonable, but current limitations in technology make this impossible.

For a select field, the effect is to filter the option elements, allowing only those whose values are members of S. There is a slight deficiency in the design of a singular select, since it in some browser implementations will always show one selected element. To account for the situation where no option is allowed, we introduce an extension of standard HTML, namely <option value="foo" error> which is legal irrespective of the format. The form cannot be submitted if the error option is selected, unless S is the empty set.

For a radio field, the effect is that the button can only be depressed if its value is a member of S; if S is not the empty set, then the form cannot be submitted unless one button is depressed. Note that the analogue of the error option is the case where no button is depressed.

For a checkbox field, the effect is that the button can only be depressed if its value is a member of S.

Using our mechanism, it is possible to create a *deadlocked* form that cannot be submitted. The simplest example is the following, assuming the input field below is the only one in the radio button group named foo:

```
<input type="radio" name="foo" value="aaa">
```

```
<format name="foo"><const value="bbb"></format>
```

Regardless of whether the radio button `foo` is depressed or not, `foo` will never satisfy its requirements. Thus, the form can never be submitted. This behavior exposes a flaw in the design of the form, rather than an inherent problem with our mechanisms.

Examples

All reasonable data formats can be expressed as regular expressions, some more complicated than others. A simple example is the password format for user ID registration, seen in Figure 4, which is five or more characters not all alphabetic:

```
<regexp id="pwd">
  <intersection>
     <repeat low="5"><anychar/></repeat>
     <complement>
       <star>
         <union>
           <range low="a" high="z"/>
           <range low="A" high="Z"/>
         </union>
       </star>
     </complement>
  </intersection>
</regexp>
```

or alternatively using the Perl syntax where possible:

```
<regexp id="pwd">
  <intersection>
     <regexp exp=".{5,}"/>
     <complement>
       <regexp exp="[a-zA-Z]*"/>
     </complement>
  </intersection>
</regexp>
```

To enforce this format on the existing form, we just add the declarations:

```
<format name="Password1"><regexp idref="pwd"/></format>
<format name="Password2"><regexp idref="pwd"/></format>
```

At our Web site we show more advanced examples, such as legal dates including leap days, URIs, and time of day. As a final example, consider a simple format for ISBN numbers:

```
<regexp id="isbn">
  <concat>
    <repeat count="9">
       <concat>
         <range low="0" high="9"/>
```

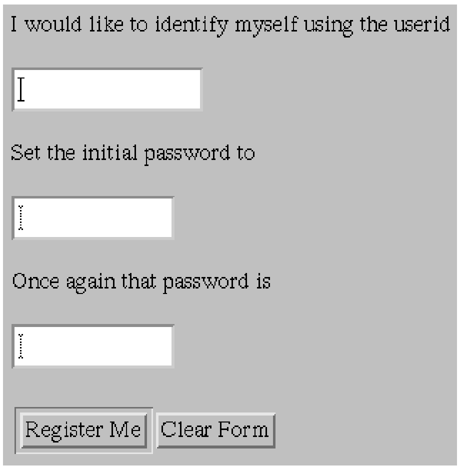


Figure 4: User ID registration.

```
      <optional><charset value=" -"/></optional>
    </concat>
  </repeat>
  <charset value="0123456789X"/>
 </concat>
</regexp>
```

or more succinctly:

```
<regexp id="isbn">
  <regexp exp="([0-9]([ -]?)){9}[0-9X]"/>
</regexp>
```

An input field that exploits this format is:

```
Enter ISBN number: <input type=text name="isbn" size=20>
<format name="isbn"
        help="Enter an ISBN number"
        error="Illegal ISBN format">
  <regexp idref="isbn"/>
</format>
```

Initially, the field has a yellow light. This status persists, as seen in Figure 5, while we enter the text `"0-444-50264-"` which is a legal prefix of an ISBN number. Entering another `"-"` yields a red light. Deleting this character and entering `5` will finally give a legal value and a green light.

While the input field has focus, the `help` string appears in the status line of the browser. If the client attempts to submit the form with invalid data in this field, then the `error` text appears in an alert box.

An ISBN format that includes checksums can be described as a complex regular expression that yields a 201-state automaton. This full format would

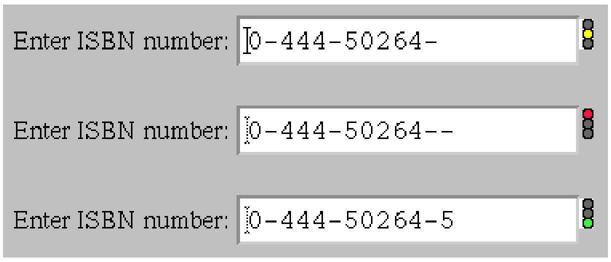


Figure 5: Checking ISBN numbers.

only accept `5` as the last digit, since that is the correct checksum. Such a regular expression could hardly be written by hand; in fact, we generated it using a C program. But as precompiled automata may be saved and provided as formats, this shows that our technology also allows us to construct and publish a collection of advanced default formats, similarly to the datatypes employed in XML Schema [2] and the predefined `ctype` formats suggested in [12].

3 Interdependencies of Form Fields

We present a simple, yet general mechanism for expressing interdependencies. We have strived to develop a purely declarative notation that requires no programming skills. Our proposal is based on dynamically evolving formats that are settled through a fixed-point process.

Syntax

We extend the syntax for formats as follows:

```
formatdecl → <format name=stringconst> format </format>

format     → regexp |
             <if> boolexp
                <then> format </then>
                <else> format </else>
             </if> |
             <format id=stringconst> format </format> |
             <format idref=stringconst/>

boolexp    → <match name=stringconst> regexp </match> |
             <equal name=stringconst value=stringconst/> |
             <and> boolexp* </and> |
             <or> boolexp* </or> |
             <not> boolexp* </not>
```

Now, the format that applies to a given field is dependent on the values of other fields. The specification is a binary decision tree, whose leaves are regular expressions and whose internal nodes are boolean expressions. Each boolean expression is a propositional combination of the primitive `match` and `equal` elements that each test the field indicated by `name`. Even this simple language is more advanced than required for most uses.

Semantics of boolean expressions

A boolean expression evaluates to true or false. For a `text` or `password` field, `equal` is true iff its current value equals `value`; `match` is true iff its current value is a member of the set denoted by `regexp`. For a `select` field, `equal` is true iff the value of a currently selected option equals `value`; `match` is true iff the value of a currently selected option is a member of the set denoted by `regexp`. For a collection of `radio` or `checkbox` fields, `equal` is true iff a button whose value equals `value` is currently depressed; `match` is true iff a button whose value is a member of the set denoted by `regexp` is currently depressed.

For the boolean operators, `and` is true iff all of its children are true, `or` is true if one of its children is true, and `not` is true if all of its children are false.

Semantics of interdependencies

Given a collection of form fields $F_1,\ldots,F_n$ with associated formats and values, we define an *iteration* which in order does the following for each F_i:

- evaluate the current format based on the current values of all form fields;
- update the field based on the new current format.

The updating varies with the type of the form field:

- for a `text` field, the status light is changed to reflect the relationship between the current value and the current format;
- for a `select` field, the options are filtered by the new format, and the selected options that are no longer allowed by the format are unselected; if the current selection of a singular `select` is disallowed, the `error` option is selected;
- for a `radio` or `checkbox` field, a depressed button is released if its value is no longer allowed by the format.

An iteration is *monotonic*, which intuitively means that it can only delete user data. Technically, an iteration is a monotonic function on a specific lattice of form status descriptions. It follows that repeated iteration will eventually reach a fixed-point. In fact, if b is the total number of `radio` and `checkbox` buttons, p is the total number of `select` options, and s is the number of singular `select`s, then at most $b + p + s + 1$ iterations are required. Usually, however, the fixed-point will stabilize after very few iterations; also, a compile-time dependency analysis can keep this number down. Only complex forms with a high degree of interdependency will require many iterations.

The behavior of a PowerForm is to iterate to a new fixed-point whenever the client changes an input field; furthermore, the form data can only be submitted when all the form fields are in a status that allows this.

Note that the fixed-point we obtain is dependent on the order in which the form fields are updated: permuting the fields may result in a different fixed-point. We choose to update the fields in the textual order in which they appear in the document. This is typically the order in which the client is supposed to consider them, and the resulting fixed-point appears to coincide with the intuitively expected behavior. For simpler forms, the order is usually not significant.

With form interdependency it is not only possible to create a deadlocked form that can never be submitted, but also to create buttons that can never be depressed. Consider again the example from Section 2. Since the value `aaa` is different from `bbb`, the `foo` button will instantly be released whenever it is depressed. Such behavior can of course also stem from more complicated interdependent behavior.

The possible behaviors of PowerForms can in principle be analyzed statically. Define the size $|R|$ of a regular expression to be the number of states in the corresponding minimal, deterministic finite-state automaton, and the size $|F|$ of an input field to be the product of the sizes of all regular expressions that it may be tested against. Then a collection of input fields $F_1, \ldots, F_n$ determines a finite transition system with $|F_1||F_2| \cdots |F_n|$ states for which the reachability problem is decidable but hardly feasible in practice. We therefore leave it to the Web author to avoid aberrant behavior.

Examples

As a first example, we will redo the questionnaire from Figure 1:

```
Have you attended past WWW conferences?
<input type="radio" name="past" value="yes">Yes
<input type="radio" name="past" value="no">No
<br>
 If Yes, how did WWW8 compare?
<input type="radio" name="compare" value="better">Better
<input type="radio" name="compare" value="same">Same
<input type="radio" name="compare" value="worse">Worse
```

To obtain the desired interdependence, we declare the following format:

```
<format name="compare">
   <if><equal name="past" value="yes"/>
     <then><complement><const value=""/></complement></then>
     <else><empty/></else>
   </if>
</format>
```

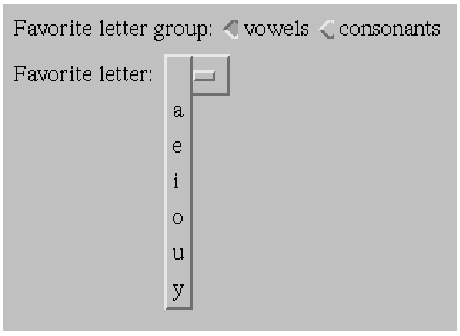


Figure 6: Only vowels are presented.

Only if the first question is answered in the positive, may the second group of radio buttons may be depressed and an answer is also required. A second example shows how radio buttons may filter the options in a selection:

```
Favorite letter group:
<input type="radio" name="group" value="vowel" checked>vowels
<input type="radio" name="group" value="consonant">consonants
<br>
Favorite letter:
<select name="letter">
  <option value="a">a
  <option value="b">b
  <option value="c">c
  ...
  <option value="x">x
  <option value="y">y
  <option value="z">z
</select>
```

The unadorned version of this form allows inconsistent choices such as `group` having value `vowel` and `letter` having value `z`. However, we can add the following format:

```
<format name="letter">
  <if><equal name="group" value="vowel"/>
    <then><charset value="aeiouy"/></then>
    <else><charset value="bcdfghjklmnpqrstvwxz"/></else>
  </if>
</format>
```

Apart from enforcing consistency, the induced behavior will make sure that the client is only presented with consistent options, as shown in Figure 6. Next, consider the form:

```
<b>Personal info</b>
```

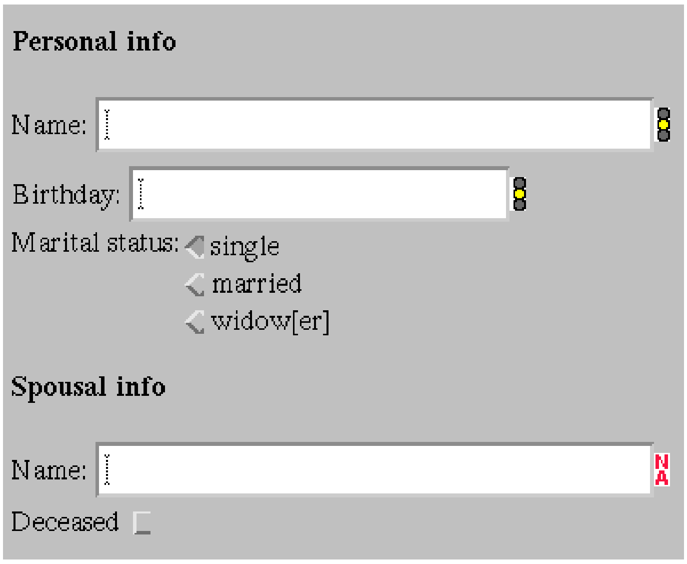


Figure 7: Collecting personal information.

```
<p>
Name: <input type="text" name="name" size="30"><br>
Birthday: <input type="text" name="birthday" size="20"><br>
<table border="0" cellpadding="0" cellspacing="0">
<tr><td valign="top">Marital status:</td>
<td><input type=radio name="marital" value="single" checked>single
<br>
<input type="radio" name="marital" value="married">married
<br>
<input type="radio" name="marital" value="widow">widow[er]
</td>
</tr>
</table>
<p>
<b>Spousal info</b>
<p>
Name: <input type="text" name="spouse" size="30"><br>
Deceased <input type="radio" name="deceased" value="deceased">
```

Several formats can be used here. For the birthday, we select from our standard library a 35-state automaton recognizing legal dates including leap days:

```
<format name="birthday">
  <regexp uri="http://www.brics.dk/bigwig/powerforms/date.dfa"/>
</format>
```

Among the other fields, there are some obvious interdependencies. Spousal info is only relevant if the `marital` status is not `single`, and the spouse can only be

Figure 8: Collecting customer information.

`deceased` if the `marital` status is `widow`:

```
<format name="spouse">
  <if><equal name="marital" value="married"/>
    <then><regexp idref="handle"/></then>
    <else>
      <if><equal name="marital" value="single"/>
        <then><empty/></then>
        <else><regexp idref="handle"/></else>
      </if>
    </else>
  </if>
</format>

<format name="deceased">
  <if><equal name="marital" value="widow"/>
    <then><const value="deceased"/></then>
    <else><empty/></else>
  </if>
</format>
```

Here, `handle` refers to some regular expression for the names of people. Note that if the `marital` status changes from `widow` to `single`, then the `deceased` button will automatically be released. Dually, it seems reasonable that after a change from `single` to `widow`, the `deceased` button should automatically be depressed. However, such action is generally not meaningful, since it may cause the form to oscillate between two settings. In our formalism, this would violate the monotonicity property that guarantees termination of the fixed-point iteration. Still, the form cannot be submitted until the `deceased` button is depressed for a `widow`. The initial form is shown in Figure 7.

An example of a more complex boolean expression involves the form in Figure 8. Here, simple formats determine that the correct style of phone numbers is used for the chosen country. The option of requesting a visit from the NYC office is only open to those customers who live in New York City. This constraint is enforced by the following format:

```
<format name="nyc">
  <if><and><equal name="country" value="US"/>
           <match name="phone">
              <concat>
                <union>
                  <const value="212"/>
                  <const value="347"/>
                  <const value="646"/>
                  <const value="718"/>
                  <const value="917"/>
                </union>
                <anything/>
              </concat>
           </match>
      </and>
   <then><anything/></then>
   <else><empty/></else>
  </if>
</format>
```

Residents from other cities will find that they cannot depress the button.

As a final example of the detailed control that we offer, consider the form in Figure 9 which invites users to request a new version of a product. Until the client has stated whether he has a license or not, it is impossible to choose a version. Once the choice has been made, licensed users can choose between all versions, others are limited to versions 1.1 and 1.2. The format on the last group of radio buttons is:

```
<format name="version">
  <if><equal name="license" value="yes"/>
      <then><anything/></then>
      <else>
        <if><equal name="license" value="no"/>
            <then><union>
                    <const value="1.1"/>
                    <const value="1.2"/>
                  </union>
            </then>
            <else><empty/></else>
        </if>
      </else>
  </if>
</format>
```

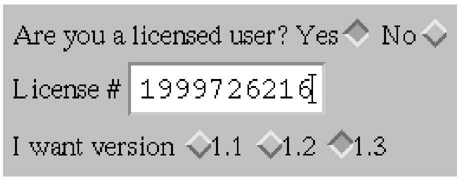


Figure 9: Collecting user information.

4 Applet results

Java applets can be used in conjunction with forms to implement new GUI components that collect data from the client. However, it is not obvious how to extract and validate data from an applet and submit it to the server on equal footing with ordinary form data.

We propose a simple mechanism for achieving this goal. We extend the applet syntax to allow `result` elements in addition to `param` elements. An example is the following:

```
<applet codebase="http://www.brics.dk/bigwig/powerapplets"
        code="slidebar.class">
   <param name="low" value="32">
   <param name="high" value="212">
   <result name="choice">
</applet>
```

When this applet is displayed, it shows a slide bar ranging over the interval [32..212]. When the form is submitted, the applet will be requested to supply a value for the `choice` result. This value is then assigned to a `hidden` form field named `choice` and will now appear with the rest of the form data. If the applet is not ready with the result, then the form cannot be submitted.

This extension only works for applets that are subclasses of the special class `PowerApplet` that we supply. It implements the method `putResult` that is used by the applet programmer to supply results, as well as the methods `resultsReady` and `getResult` that are called by the JavaScript code that implements the form submission.

In relation to PowerForms, applet results play the same role as input fields. Thus, they can have associated formats and be tested in boolean expressions. The value of an optional `error` attribute will appear in the alert box if an attempt is made to submit the form with a missing or invalid applet result.

5 Translation to JavaScript

A PowerForms document is parsed according to a very liberal HTML grammar that explicitly recognizes the special elements such as `format` and `regexp`. The generated HTML document retains most of the original structure, except that it contains the generated JavaScript code. Also, each input field is modified to include `onKeyup`, `onChange`, and `onClick` functions that react to modifications from the client.

A function `update_foo` is defined for each input field name `foo`. This function checks if the current data is valid and reacts accordingly. Another function `update_all` is responsible for computing the global fixed-point.

Each regular expression is by the compiler transformed into a minimal, deterministic finite-state automaton, which is directly represented in a JavaScript data structure. It is a simple matter to use an automaton for checking if a data value is valid. For `text` and `password` fields, the status lights green, yellow, and red correspond to respectively an accept state, a non-accept state, and the crash state. For efficiency, the generated automata are time-stamped and cached locally; thus, they are only recompiled when necessary.

The generated code is quite small, but relies on a 500 line standard library with functions for manipulating automata and the Document Object Model [1].

6 Availability

The PowerForms system is freely available in an open source distribution from our Web site located at `http://www.brics.dk/bigwig/powerforms/`. The package includes documentation, the examples from this paper and many more, and the compiler itself which is written in 4000 lines of C. The generated JavaScript code has been tested for Netscape on Unix and Windows and for Explorer on Windows.

PowerForms are also directly supported by the `<bigwig>` system which is a high-level language for generating interactive Web services [4, 3, 11, 10]. It is likewise available at `http://www.brics.dk/bigwig/`.

7 Conclusion

We have shown how to enrich HTML forms with simple, declarative concepts that capture advanced input validation and field interdependencies. Such forms are subsequently compiled into JavaScript and standard HTML. This allows the design of more complex and interesting forms while avoiding tedious and error-prone JavaScript programming.

We would like to thank the entire `<bigwig>` team for assisting in experiments with PowerForms. Thanks also goes to the PowerForms users, in particular Frederik Esser, for valuable feedback.

References

[1] Vidur Apparao et al. *Document Object Model (DOM) Level 1 Specification.* W3C Recommendation, 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[2] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes.* W3C Working Draft, April 2000. http://www.w3.org/TR/xmlschema-2/.

[3] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. A runtime system for interactive Web services. *Computer Networks*, 31:1391–1401, 1999. Also in proceedings of WWW8.

[4] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. Designing a language for developing interactive Web services, 2000. http://www.brics.dk/bigwig/research/publications/.

[5] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0.* W3C Recommendation, February 1998. http://www.w3.org/TR/REC-xml.

[6] John Desborough. *Cold Fusion 3.0 Intranet Application.* International Thomson Publishing, 1997.

[7] Andreas Girgensohn and Alison Lee. Seamless integration of interactive forms into the web. In *Proceedings of WWW6*, 1997. http://www.scope.gmd.de/info/www6/technical/paper083/paper83.html.

[8] Jukka Korpela. JavaScript and HTML: possibilities and caveats, 2000. http://www.hut.fi/u/jkorpela/forms/javascript.html.

[9] Netscape Corp. JavaScript form validation sample code, 1999. http://developer.netscape.com/docs/examples/javascript/formval/overview.html.

[10] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for Web services. In *Fundamental Approaches to Software Engineering, FASE'98*, LNCS 1382, pages 270–284. Springer-Verlag, March/April 1998.

[11] Anders Sandholm and Michael I. Schwartzbach. A domain specific language for typed dynamic documents. In *Proceedings of POPL'00*, 2000.

[12] Sebastian Schnitzenbaumer, Malte Wedel, and Muditha Gunatilake, editors. *XHTML-FML 1.0: Forms Markup Language.* Stack Overflow AG, 1999. http://www.mozquito.org/documentation/spec_xhtml-fml.html.

[13] Sebastian Schnitzenbaumer, Malte Wedel, and Dave Raggett, editors. *XHTML Extended Forms Requirements.* W3C, 1999. http://www.w3.org/TR/xhtml-forms-req.html.

[14] Paul Thistlewaite and Steve Ball. Active forms. In *Proceedings of WWW5*, 1996. http://www5conf.inria.fr/fich_html/papers/P40/Overview.html.

[15] Martin Webb and Michel Plungjan. JavaScript form FAQ knowledge base, 2000. http://developer.irt.org/script/form.htm.

Recent BRICS Report Series Publications

RS-00-43 Claus Brabrand, Anders Møller, Mikkel Christensen, Ricky, and Michael I. Schwartzbach. *PowerForms: Declarative Client-Side Form Field Validation*. December 2000. 21 pp. To appear in *World Wide Web Journal*, 4(3), 2000.

RS-00-42 Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. *The `<bigwig>` Project*. December 2000. 25 pp.

RS-00-41 Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. *The DSD Schema Language and its Applications*. December 2000. 32 pp. Shorter version appears in Heimdahl, editor, *3rd ACM SIGSOFT Workshop on on Formal Methods in Software Practice*, FMSP '00 Proceedings, 2000, pages 101–111.

RS-00-40 Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. *MONA Implementation Secrets*. December 2000. 19 pp. Shorter version appears in Daley, Eramian and Yu, editors, *Fifth International Conference on Implementation and Application of Automata*, CIAA '00 Pre-Proceedings, 2000, pages 93–102.

RS-00-39 Anders Møller and Michael I. Schwartzbach. *The Pointer Assertion Logic Engine*. December 2000. 23 pp. To appear in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '01 Proceedings, 2001.

RS-00-38 Bertrand Jeannet. *Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Synchronous Programs*. December 2000.

RS-00-37 Thomas S. Hune, Kim G. Larsen, and Paul Pettersson. *Guided Synthesis of Control Programs for a Batch Plant using* UPPAAL. December 2000. 29 pp. Appears in Hsiung, editor, *International Workshop in Distributed Systems Validation and Verification. Held in conjunction with 20th IEEE International Conference on Distributed Computing Systems (ICDCS '2000)*, DSVV '00 Proceedings, 2000.

RS-00-36 Rasmus Pagh. *Dispersing Hash Functions*. December 2000. 18 pp. Preliminary version appeared in Rolim, editor, *4th. International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '00, Proceedings in Informatics 8, 2000, pages 53–67.