



XML Prague 2018

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 8–10, 2018

XML Prague 2018 – Conference Proceedings

Copyright © 2018 Jiří Kosek

ISBN 978-80-906259-4-5 (pdf)

ISBN 978-80-906259-5-2 (ePub)



The Complete Solution for XML Authoring & Development



XML Editor

oXygen XML Editor is a complete XML editing solution for developers and content authors.



XML Author

oXygen XML Author provides a visual interface designed for user-friendly structured authoring.



XML Developer

oXygen XML Developer is an effective and easy-to-use industry-leading XML development tool.



XML Web Author

oXygen XML Web Author is the ultimate tool for editing and reviewing content in browsers on any device.



WebHelp

oXygen XML WebHelp allows you to publish DITA and DocBook in a modern, interactive web-based help system.

www.oxygenxml.com

A comprehensible description of XML documents

The X-definition is an XML document that can be used to describe a set of XML documents.

- Easy to design and to understand
- Can be used to describe and to process JSON data
- The comprehensibility of the X-definition source makes it extremely easy to create a documentation of XML structures.

Validation, processing or construction

The X-definition integrates the validation process of XML data, the processing and the construction of a new XML document - all in the same language.

- An easy way to facilitate the maintenance of large projects
- A generation of a detailed error report
- Tools for a dynamic error processing

The Java environment interconnection

The X-definition allows Java projects interconnections.

- It is possible to execute Java methods and to access Java objects
- The feature "X-Components" enables to generate source Java code representing XML structure and to use it in Java programs (in a similar way as JAXB)

Connection to databases and external data

In the X-definition you can execute database statements (either in relational databases or in XML databases). It is possible also to access external data.

An easy maintenance of large projects

An excessive collection of X-definitions has been successfully used in real projects in which very often large data files needed to be processed (many GB).

Table of Contents

General Information	vii
Sponsors	ix
Preface	xi
Assisted Structured Authoring using Conditional Random Fields – <i>Bert Willems</i>	1
XML Success Story: Creating and Integrating Collaboration Solutions to Improve the Documentation Process – <i>Steven Higgs</i>	13
xqerl: XQuery 3.1 Implementation in Erlang – <i>Zachary N. Dean</i>	23
XML Tree Models for Efficient Copy Operations – <i>Michael Kay</i>	33
Using Maven with XML development projects – <i>Christophe Marchand and Matthieu Ricaud-Dussarget</i>	49
Varieties of XML Merge: Concurrent versus Sequential – <i>Tejas Pradip Barhate and Nigel Whitaker</i>	61
Including XML Markup in the Automated Collation of Literary Text – <i>Elli Bleeker, Bram Buitendijk, Ronald Haentjens Dekker, and Astrid Kulsdom</i>	77
Multi-Layer Content Modelling to the Rescue – <i>Erik Siegel</i>	97
Combining graph and tree – <i>Hans-Juergen Rennau</i>	107
SML – A simpler and shorter representation of XML – <i>Jean-François Larvoire</i> ...	137
Can we create a real world rich Internet application using Saxon-JS? – <i>Pieter Masereeuw</i>	157
Implementing XForms using interactive XSLT 3.0 – <i>O’Neil Delpratt and Debbie Lockett</i>	167
Life, the Universe, and CSS Tests – <i>Tony Graham</i>	181
Form, and Content – <i>Steven Pemberton</i>	213
tokenized-to-tree – <i>Gerrit Imsieke</i>	229

Form, and Content

Data-Driven Forms

Steven Pemberton

CWI, Amsterdam

<steven.pemberton@cwi.nl>

Abstract

Because of the legacy of paper-based forms, modern computer-based forms are often seen as static data-collection applications, with rows of rectangular boxes for collecting specific pieces of data. However, they have far more opportunities for being dynamic, checking data for consistency, leaving out fields for non-relevant data, and changing structure and detail to match the data-filling flow. Furthermore, data is no longer limited to pure textual input, but can be entered using any method that is available on a computer.

While classically it is the form that drives the data produced, this paper examines how forms can be data-driven, for structure, for presentation, and for execution, and proposes that our view of forms have been severely impaired by the paper-based legacy.

Keywords: XML, XForms, Forms, Data-driven

1. Introduction

Throughout history, when new technologies have been introduced, there has been a tendency for them to imitate the old technologies they are replacing before they iterate to their proper embodiment.

For instance, the first printed books looked like hand-written manuscripts, by using a typeface that imitated handwritten text, making them much harder to read than necessary; the first cars looked like horse-drawn carts without the horse, partly because that was what people could make at the time, but also because horse-drawn carts were what they were seen to be replacing — looking back from a modern perspective it is astonishing how long it took for anyone to have the bright idea of actually enclosing the driver in a space protected from the elements; and modern computer applications (such as agendas) often imitate their real-life counterparts in excruciating detail.

Digital forms have in a similar way long been held back by the history of paper-based form-filling. In the early days of the Web, in this author's experience, many managers required their online forms to be identical to their paper-based equivalents, even though this meant not being able to use facilities that would otherwise have made them much easier to use.

As online production and usage matures, so has the online form-filling experience improved. There are now a small number of standardised form development languages. However, form development is often still largely based around the static idea of the form filling experience, for example [2].

It is time to move up a level of abstraction. Forms are about data collection, and traditionally the shape of the form drives the shape of the data. This paper investigates how the data can drive the form, and shows that things that we might not traditionally see as forms can be viewed as data collection.

1.1. XForms

XForms [4], [5] is an XML-based markup language, originally designed only for traditional-style forms, but in later iterations generalised to more widely-applicable usage.

A principle feature of the language is its separation of data and associated data description from the actual controls used to display and enter the data. This separation of concerns can be compared to the separation of content and styling done with style sheets, and has similar advantages, making the data more tractable, and facilitating reuse.

The data in XForms is contained in a *model* that consists of any number of XML instances, which can be loaded from external sources, along with descriptions of properties and relationships that nodes in the data may have.

Controls in the user interface are then bound to data nodes using XPath expressions [3]. For instance:

```
<input ref="cc-number" label="Credit card number"/>
```

An example data property is *relevance*. As a simple example, the credit-card number to be input can be marked in the model as being relevant only if the method of payment is by credit card:

```
<bind ref="cc-number" relevant="../payment-method = 'credit'"/>
```

Controls bound to values that are not relevant, as well as controls bound to elements that are simply not present in the instance data, are *disabled*: they are not visible to the user, and they can't be used for input. As will be seen, this is an essential element of data-driven forms.

XForms is a W3C standard; a new version, 2.0, is in preparation [6], and some facilities of XForms 2.0 are used in the examples that follow.

This paper presents three case studies of the use of data-driven forms: one of data-driven structure, one of data-driven presentation, and the last of data-driven execution.

1.2. What is a Form?

A traditional form is no more than a method of data-collection, taking input from a user and storing it somewhere. Computer-based forms however are becoming steadily more dynamic, for instance, partially filling in an address based on the entry of a postcode, or calculating the final amount based on what you have ordered, the taxes, and the delivery costs. Clearly, modern forms deal with input from a user, but do calculation and output as well. Is a log-in dialogue box a form? Yes, it is. Is a widget asking you to move a pointer on a map to indicate the location you want to share a form? It can be so construed. As forms become more and more dynamic, the distinction between 'form' and 'application' becomes steadily more nebulous. This paper consequently uses a liberal definition of what constitutes a form.

2. Data-driven Structure: A Questionnaire

The first case study is a classic data-collection form. It is based on one produced for an Internet community, for identifying potential improvements for the internet. It has a short introduction, gives the user a choice of three options, and then reveals a small number of questions, based on the choice. (The actual production form was more complex, but for the sake of exposition, it has been compressed here to the essence).

Something like this:

Before making choice	After making choice
<div>Process improvement</div> <p>Please help us discover problems and solutions that would improve our processes.</p> <p>How can you help?</p> <ul style="list-style-type: none"><input type="radio"/> You know a problem that needs to be fixed<input type="radio"/> You know a 'solution' that doesn't work<input type="radio"/> You have a prediction about a future possible failure	<div>Process improvement</div> <p>Please help us discover problems and solutions that would improve our processes.</p> <p>How can you help?</p> <ul style="list-style-type: none"><input checked="" type="radio"/> You know a problem that needs to be fixed<input type="radio"/> You know a 'solution' that doesn't work<input type="radio"/> You have a prediction about a future possible failure <div>You know a problem that needs to be fixed</div> <p>What problem do you see?</p> <div></div> <p>Can you propose a solution?</p> <div></div> <div>Submit</div>

2.1. The Static Version

Since this is a classic style of form, in the most obvious and direct —static— approach, the controls could look like this, where the user makes a choice:

Please help us discover problems and solutions that would improve our processes.

```
<select1 ref="choice">
  <label>How can you help?</label>
  <item><label>You know a problem that needs to be fixed</label>
    <value>problem</value></item>
  <item><label>You know a 'solution' that doesn't work</label>
    <value>failure</value></item>
  <item><label>You have a prediction about a future possible failure</label>
    <value>prediction</value></item>
</select1>
```

and as a result of the choice, one of the options is displayed:

```
<switch ref="choice">

  <case name=""/> <!-- Until a choice is made, nothing is displayed -->

  <case name="problem">
    <group ref="problem">
      <label>You know a problem that needs to be fixed</label>
      <textarea ref="problem"><label>What problem do you see?</label></textarea>
      <textarea ref="solution"><label>Can you propose a solution?</label></textarea>
    </group>
  </case>

  <case name="failure">
    <group ref="failure">
      <label>You know a 'solution' that doesn't work</label>
      <textarea ref="problem">
        <label>What 'solution' will fail or cause trouble?</label>
      </textarea>
      <textarea ref="solution"><label>Can you propose a fix?</label></textarea>
    </group>
  </case>

  <case name="prediction">
    <group ref="prediction">
      <label>You have a prediction about a future possible failure</label>
      <textarea ref="problem"><label>What is your scenario?</label></textarea>
      <select1 ref="likely">
        <label>How likely is this scenario?</label>
        <item><label>High</label><value>1</value></item>
        <item><label>Medium</label><value>2</value></item>
        <item><label>Low</label><value>3</value></item>
      </select1>
      <textarea ref="who"><label>Who should address these issues?</label></textarea>
      <textarea ref="solution"><label>Can you propose a solution?</label></textarea>
    </group>
  </case>
</switch>

<submit><label>Submit</label></submit>
```

This uses the following structure for the data; for any one answer, only the values for the selected case would get filled in:

```
<data>
  <choice/>
  <problem>
```

```
        <problem/>
        <solution/>
    </problem>
    <failure>
        <problem/>
        <solution/>
    </failure>
    <prediction>
        <problem/>
        <likely/>
        <who/>
        <solution/>
    </prediction>
</data>
```

2.2. Making the Form Multi-lingual

One of the early decisions was to make the form multi-lingual. This involved creating an instance that contained all the labels and other texts:

```
<instance id="m">
  <messages xmlns="" lang="en">
    <intro>Please help us discover problems and solutions that would
      improve our processes.</intro>
    ...
  </messages>
</instance>
```

and in the body of the form:

```
<output ref="instance('m')/intro"/>
```

For the select1, the messages:

```
<choice>
  <label>How can you help?</label>
  <item value="problem">You know a problem that needs to be fixed</item>
  <item value="failure">You know a 'solution' that doesn't work</item>
  <item value="prediction">You have a prediction about
    a future possible failure</item>
</choice>
```

with the control now reading:

```
<select1 ref="choice">
  <label ref="instance('m')/choice/label"/>
  <itemset ref="instance('m')/choice/item">
    <label ref="."/>
    <value ref="@value"/>
  </itemset>
</select1>
```

```
</itemset>
</select1>
```

And for the groups within the cases, the messages:

```
<problem>
  <label>You know a problem that needs to be fixed</label>
  <problem>What problem do you see?</problem>
  <solution>Can you propose a solution?</solution>
</problem>
```

And the controls:

```
<group ref="problem">
  <label ref="instance('m')/problem/label"/>
  <textarea ref="problem">
    <label ref="instance('m')/problem/problem"/></textarea>
  <textarea ref="solution">
    <label ref="instance('m')/problem/solution"/></textarea>
</group>
```

and similar for the other two cases.

2.3. Generalising

As a result of this change, it was obvious how similar the three cases were. The third has a little extra detail, but otherwise they are nearly identical. If the data is changed to reflect these similarities, like this:

```
<data>
  <choice/>
  <answer choice="problem">
    <problem/>
    <solution/>
  </answer>
  <answer choice="failure">
    <problem/>
    <solution/>
  </answer>
  <answer choice="prediction">
    <problem/>
    <likely/>
    <who/>
    <solution/>
  </answer>
</data>
```

then the whole switch in the form can be replaced with a single group that is driven by the data. The group selects only the answer whose choice attribute matches that of the value actually chosen, and so covers all three cases:

```
<group ref="answer[@choice=../choice]">
  <label ref="instance('m')/answer[@choice=context()/@choice]/label"/>
  <textarea ref="problem">
    <label ref="instance('m')/answer[@choice=context()/../@choice]/problem"/>
  </textarea>
  <select1 ref="likely">
    <label ref="instance('m')/answer[@choice=context()/../@choice]/likely/label"/>
    <itemset ref="instance('m')/answer[@choice=context()/../@choice]/likely/item">
      <label ref="."/><value ref="@value"/>
    </itemset>
  </select1>
  <textarea ref="who">
    <label ref="instance('m')/answer[@choice=context()/../@choice]/who"/>
  </textarea>
  <textarea ref="solution">
    <label ref="instance('m')/answer[@choice=context()/../@choice]/solution"/>
  </textarea>
</group>
```

Note that:

- Just as before, since initially no answer has a choice with a value that has been selected (since nothing has yet been selected), the ref will select no nodes, and so nothing will be displayed for the answers.
- If the instance data for the selected answer doesn't have a likely or who element, the controls for those elements won't be displayed, since similarly they are not bound to any node.

This change to the data requires a similar change to the structure of the messages document. Note how closely its structure mirrors that of the data:

Data	Messages
<pre> <data> <choice/> <answer choice="problem"> <problem/> <solution/> </answer> <answer choice="failure"> <problem/> <solution/> </answer> <answer choice="prediction"> <problem/> <likely/> <who/> <solution/> </answer> </data> </pre>	<pre> <messages lang="en"> <label>Process improvement</label> <intro>Please help us discover problems and solutions that would improve our processes.</intro> <choice>How can you help?</choice> <answer choice="problem"> <label>You know a problem that needs to be fixed</label> <problem>What problem do you see?</problem> <solution>Can you propose a solution?</solution> </answer> <answer choice="failure"> <label>You know a 'solution' that doesn't work</label> <problem>What 'solutions' will fail or cause trouble?</problem> <solution>Can you propose a solution?</solution> </answer> <answer choice="prediction"> <label>You have a prediction about a future possible failure</label> <problem>What is your scenario?</problem> <likely> <label>How likely is this scenario?</label> <item value="1">High</item> <item value="2">Medium</item> <item value="3">Low</item> </likely> <who>Who should address these issues?</who> <solution>Can you propose a solution?</solution> </answer> </messages> </pre>

The message for the group's label is selected using `answer[@choice=context()/@choice]`. This selects the answer element in the messages, whose choice attribute matches that of the context element. In this case the context element is the answer element in the data that has been selected.

For the first textarea element, the context item is now the element problem, which is a child of answer, so you have to go up one level to get to answer, in order to get its choice attribute: `item[@choice=context()../@choice]`.

2.4. Analysis

The form is now driven from two data files: the template for the data, and the messages. Essentially, the group of controls acts as an interpreter of the data.

To illustrate this, suppose a fourth option were to be added to the form; all that is necessary is to add it to the data template, with a suitable new choice value:

```
<answer choice="solution"><problem/><who/><solution/></answer>
```

and matching messages in the message file:

```

<answer choice="solution">
  <label>You know an existing solution that can be adopted</label>

```

```
<problem>What solution do you know of?</problem>
<who>Who should we approach?</who>
<solution>How could we best adopt the solution?</solution>
</answer>
```

and the form now works *without change* with the new entry.

This makes life much easier for content providers: they can make textual changes to a form without having to ask the programmers to do it, they can add new cases themselves fairly easily. In fact, you could even make a form to simplify the process!

To adapt the form for a different language, you only have to supply a translation of the message document for the new language, and add code to switch between languages by loading in the various message documents.

3. Data-driven Display: A Presentation Manager

The CSS [1] styling language has a special *presentation* mode. If you include a set of rules grouped as *projection* media, like so:

```
@media projection {
  ...
}
```

then when the browser is put into presentation mode, those styling rules apply. The idea is that the browser goes into full-screen mode, and the projection rules will typically increase the font size, and express where 'page' breaks are. As a result, this allows you to avoid using proprietary presentation software, and use HTML, plus CSS with a projection mode, and present from a browser.

This has had several advantages, for instance:

- it makes the content easily repurposable,
- it gives a choice of editing software,
- it is platform independent,
- it gives a lot of control,
- but most important: it guarantees a long life for the content. The use of proprietary software always brings with it the risk of the content no longer being readable after several years.

Unfortunately, and shamefully, only one browser ended up supporting presentation mode, Opera, but now even Opera has discontinued support.

Since the effective demise of presentation mode, many packages of Javascript have emerged to support presentation in combination with HTML5, such as [9], [10], [11], [12], and [13] (and *dozens* more) and although some of them are very cute, they all have some underlying problems:

- They are largely not standardised: each has its own format for slides, and its own package of javascript, so you can't swap between packages;
- If you want to repurpose existing content, you have to edit the content files;
- If support for the package disappears (which has already happened for some packages), you are in trouble: this is comparable to the problem of using proprietary software.

To mitigate these problems, one solution is to use XForms to display the slides. Of course, this is not quite as good as having a standard built into the browser, but the advantages include:

- It is very easy: it is a surprisingly small amount of markup;
- it allows the continued use and repurposing of existing content without change;
- it continues to give the power of XHTML+CSS for styling.

3.1. Slide Deck

Each slide deck is an XHTML document, where, in this example, each slide is a top-level div containing XHTML including images.

The initial slide deck is loaded into an XForms instance like this:

```
<instance id="slides"
  src="http://www.cwi.nl/~steven/Talks/2018/prague/" />
```

(we'll see later how to load different decks).

The central part of the application is then an XForms group that handles a single div:

```
<group ref="h:body/h:div[position()=instance('i')/index]">
  ...
</group>
```

This selector defines how to find a single slide within the instance, so it can be considered cleaner to gather the instance and this definition together:

```
<instance id="slides"
  src="http://www.cwi.nl/~steven/Talks/2018/prague/" />
<bind id="slide" ref="h:body/h:div"/>
```

and then use this for the group. In this way, the controls in the form are independent of the data:

```
<group ref="bind('slide')[position()=instance('i')/index]">
  ...
</group>
```

(The bind function is an XForms 2.0 feature)

Either way, this requires an administration instance to keep track of which slide is visible at any time, initialised to 1:

```
<instance id="i">
  <admin xmlns="">
    <index>1</index>
  </admin>
</instance>
```

Although buttons *could* be added to step through the slides like this:

```
<trigger label="←">
  <setvalue ev:event="DOMActivate"
    ref="instance('i')/index" value=". - 1"/>
</trigger>
<trigger label="→">
  <setvalue ev:event="DOMActivate"
    ref="instance('i')/index" value=". + 1"/>
</trigger>
```

it is preferable to do it via the keyboard, not least because presentation remotes act as if they are keyboards, sending the characters "Page Up" and "Page Down" when the buttons are pressed:

```
<action ev:event="keydown" ev:defaultAction="cancel">
  <setvalue ref="instance('i')/index"
    if="event('key')='PageUp'
      or event('key')='ArrowLeft'" value=". - 1"/>
  <setvalue ref="instance('i')/index"
    if="event('key')='PageDown' or
      event('key')='ArrowRight'" value=". + 1"/>
</action>
```

(It is necessary to cancel the default action of the event, since otherwise the browser would do a page up or down as well.)

3.2. Displaying One Slide

Now that we have the infrastructure to step through each slide, we can define how an individual slide should be presented.

Each slide contains a sequence of XHTML elements. So within the group holding the slide, each of those elements have to be displayed. They are treated one by one within a repeat:

```
<repeat ref="*">
  ...
</repeat>
```

Here are some simple cases:

```
<output class="h1" ref=". [name(.)='h1']"/>
<output class="h2" ref=". [name(.)='h2']"/>
<output class="pre" ref=". [name(.)='pre']"/>
```

The XPath idiom ". [name(.)='h1']" selects the current element only if its name is 'h1'. If its name doesn't match, then no node is selected by the output element, and so the control is *disabled* and is not rendered; if the name matches, then its content is output. By attaching a class, CSS controls how it will be displayed. Clearly at most one of the output elements will be enabled.

In fact these can be combined into one output element by taking advantage of XForms 2.0 attribute value templates:

```
<output class="{name(.)}"
  ref=". [name(.)='h1' or name(.)='h2' or name(.)='pre']"/>
```

More complicated cases are those elements that themselves contain other elements, such as <p> and .

The easier of these two is . Here a similar trick is used, with a repeat over the contained elements, with the advantage that we know they are all elements:

```
<group class="ul" ref=". [name(.)='ul']">
  <repeat ref=". [name(.)='li']">
    <output class="li" ref="."/>
  </repeat>
</group>
```

The <p> elements have a complication that they may contain mixed content. For this, rather than using the selector "*", the selector "node()" is used, which selects all child nodes: text and comments as well as elements:

```
<group class="p" ref=". [name(.)='p']">
  <repeat ref="node()">
    <output class="text" ref=". [name(.)='#text']"/>
    <output class="{name(.)}"
      ref=". [name(.)='em' or name(.)='strong' or
name(.)='code' or name(.)='a']"/>
    <output class="img"
      ref=". [name(.)='img']" value="concat(instance('i')/base, ►
@src)" mediatype="image/*"/>
  </repeat>
</group>
```

Since there is no output element that selects comment nodes, they won't be displayed.

The only interesting case here is for images. The `src` attribute is relative to the original slides, so must be concatenated with the base URL of the slides, which can be stored in the admin instance:

```
<instance id="i">
  <admin xmlns="">
    <index>1</index>
    <base>https://homepages.cwi.nl/~steven/Talks/2018/prague/</base>
  </admin>
</instance>
```

3.3. Loading Other Slide Sets

Having the base stored in the admin instance makes it easy to load another slide set. The user supplies the URL of the new slide set, it gets submitted, and the result is used to replace the slides instance:

```
<input ref="instance('i')/base" label="URL:"/>
<submit submission="change" label="Go"/>
```

where the `<submission>` element looks like this, remembering also to set the index back to 1:

```
<submission id="change" resource="{instance('i')/base}"
  method="get" serialize="none"
  replace="instance" instance="slides">
  <action ev:event="xforms-submit-done">
    <setvalue ref="instance('i')/index" value="1"/>
  </action>
</submission>
```

3.4. Analysis

The input from the user for this form is minimal: it is a URL for a slide set, and a single integer, indicating which slide to display, which is incremented and decremented via keystrokes, or emulated keystrokes from a presentation remote. Although it was presented as an 'administrative' value, it is in fact the central piece of input. The controls, in a similar way to the first example, are an interpreter for the data in the selected slide; the result can be considered a 'presentation' of the integer.

4. Data-driven Control: The XForms 2.0 Test Suite

XForms 1.0 and 1.1 both had test suites that consisted largely of static XForms documents [7], [8]. If you wanted to add more cases to a test, it involved adding to the set of documents, or editing the individual documents.

The test suite for XForms 2.0 now being constructed takes a different approach. While different parts of the test suite have different structures, depending on what is being tested, we consider here the testing of functions.

4.1. Testing Functions

It is required to test that functions like

```
compare('apple', 'orange')
```

return the right result.

To do this, the string is enclosed in an element:

```
<test>compare('apple', 'orange')</test>
```

sub-elements are added to identify the parameters

```
<test>compare('<a>apple</a>', '<b>orange</b>')</test>
```

and attributes added to store the required result, the actual result, and whether the test case passes or not:

```
<test pass="" res=""  
      req="-1">compare('<a>apple</a>', '<b>orange</b>')</test>
```

As many such test cases as necessary are then gathered together in an instance:

```
<instance>  
  <tests pass="" name="compare() function" xmlns="">  
    <test pass="" res=""  
      req="-1">compare(<a>apple</a>, <b>orange</b>)</test>  
    <test pass="" res=""  
      req="1">compare(<a>orange</a>, <b>apple</b>)</test>  
    <test pass="" res=""  
      req="0">compare(<a>apple</a>, <b>apple</b>)</test>  
    ...  
  </tests>  
</instance>
```

A bind is then used to calculate the individual results:

```
<bind ref="test/@res" calculate="compare(..@a, ../b)"/>
```

another bind, independent of which function is being tested, decides if each test case has passed:

```
<bind ref="test/@pass" calculate="if(..@res = ../@req, 'yes', 'no')"/>
```

and finally a bind for the attribute on the outmost element records if all tests have passed:

```
<bind ref="@pass" calculate="if(count(//test[@pass!  
='yes'])=0, 'PASS', 'FAIL')"/>
```


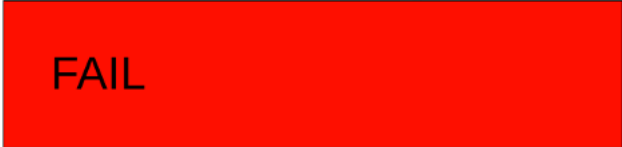
With this structure, every test form has an identical set of controls, that output the name of the test, an optional description (which is only displayed if present in the instance), whether all tests have passed, for quick inspection, and the list of each test with an indication if it has not passed:

```

<group>
  <label class="title" ref="@name"/>
  <output class="block" ref="description"/>
  <output class="{@pass}" ref="@pass"/>
  <repeat ref="test">
    <output value="."/> → <output ref="@res"/>
    <output class="wrong"
      value="if(@pass!='yes', concat(' expected: ', ►
@req), '')"/>
  </repeat>
</group>

```

This looks like this when run:

Success	Failure
<p>compare() function</p>  <pre> compare(apple, orange) → -1 compare(orange, apple) → 1 compare(apple, apple) → 0 compare(apple,) → 1 compare(, apple) → -1 compare(,) → 0 compare(1, 2) → -1 compare(2, 1) → 1 compare(2, 10) → 1 compare(10, 2) → -1 compare(1, -1) → 1 compare(apple, APPLE) → 1 compare(cafe, café) → -1 compare(small, 大) → -1 compare(大, small) → 1 compare(大, 大) → 0 compare(大, 小) → -1 compare(小, 大) → 1 </pre>	<p>seconds-from-dateTime() function</p>  <pre> 1970-01-01T00:00:00Z → 0 1970-01-01T00:00:00 → 0 1969-12-31T23:59:59 → -1 1970-01-01T00:00:00-08:00 → -2880000 expected: 28800 1970-01-01T00:01:00+00:01 → 120 expected: 0 1970-01-01T00:00:00-00:01 → -60 1970-01-01T00:00:00-00:02 → -120 1970-01-01T00:01:00-00:02 → -60 1970-01-01T00:02:00-00:02 → 0 1970-01-01T00:01:00-00:01 → 0 </pre>

4.2. Analysis

Not all test cases can be structured like this, but many can, and even tests that do not test functions can emulate this behaviour by writing results to the tests instance, and use the same mechanism for checking. These forms are introspective: they are requiring the XForms processor to reveal properties of itself, to itself. Although there is no *direct* input from the user, the input in this case can be seen as being the XForms processor itself.

5. Conclusion

In the introduction, the comparison of XForms's separation of data and controls with the separation of style and content using style sheets was not accidental. Someone typing a publishing contract into a word-processor may only be interes-

ted in the content; a designer styling the contract may only be interested in how it look; someone in the publishing industry may only be interested in the name of the author, and the sizes of the advance and the royalty being granted: a document may have several layers of abstraction. It is data-driven forms that can represent one of those layers. The dynamism afforded by computer-based forms has blurred the distinction between form and application, and allowed similar techniques and mechanisms to be applied to both.

As shown in the three cases here, the use of presence and relevance of data elements to drive the controls of the interface with the user gives a lot of power, and affords the declarative definition of applications that would normally be thought of as procedural in nature.

6. References

Bibliography

- [1] *CSS Snapshot 2017*. Tab Atkins Jr. et al.. W3C. 2017. <https://www.w3.org/TR/css-2017/> .
- [2] *Forms that Work*. Caroline Jarrett and Gerry Gaffney. Morgan Kaufmann. 2009.
- [3] *XML Path Language*. Anders Berglund et al.. W3C. 2010. <https://www.w3.org/TR/xpath20/> .
- [4] *XForms 1.0*. Micah Dubinko et al.. W3C. 2003. <https://www.w3.org/TR/2003/REC-xforms-20031014/> .
- [5] *XForms 1.1*. John M. Boyer et al.. W3C. 2009. <http://www.w3.org/TR/2009/REC-xforms-20091020/> .
- [6] *XForms 2.0*. Erik Bruchez et al.. W3C. 2017. https://www.w3.org/community/xformsusers/wiki/XForms_2.0.
- [7] *XForms 1.0 Test Suite*. W3C. 2003. https://www.w3.org/MarkUp/Forms/Test/XForms1.0/Edition3/front_html/XF103edTestSuite.html.
- [8] *XForms 1.1 Test Suite*. W3C. 2009. <https://www.w3.org/MarkUp/Forms/Test/XForms1.1/Edition1/driverPages/html/>.
- [9] *reveal*. <http://lab.hakim.se/reveal-js> .
- [10] *remark*. <https://remarkjs.com/> .
- [11] *webslides*. <https://webslides.tv/> .
- [12] *deck*. <http://imakewebthings.com/deck.js> .
- [13] *shwr*. <https://shwr.me/> .

Jiří Kosek (ed.)

**XML Prague 2018
Conference Proceedings**

Published by
Ing. Jiří Kosek
Filipka 326
463 23 Oldřichov v Hájích
Czech Republic

PDF was produced from DocBook XML sources
using XSL-FO and AH Formatter.

1st edition

Prague 2018

ISBN 978-80-906259-4-5 (pdf)
ISBN 978-80-906259-5-2 (ePub)