XML Query Forms (XQForms): Declarative Specification of XML Query Interfaces

Michalis Petropoulos*
Dept. of Computer Science and Engineering, University of California, San Diego
mpetropo@cs.ucsd.edu

Vasilis Vassalos
Information Systems Dept., New York University and Enosys Markets, Inc.
vasilis@enosysmarkets.com

Yannis Papakonstantinou
Dept. of Computer Science and Engineering, University of California, San Diego and Enosys Markets, Inc.
yannis@cs.ucsd.edu

ABSTRACT

XQForms is the first generator of Web-based query forms and reports for XML data. XQForms takes as input (i) XML Schemas that model the data to be queried and presented, (ii) declarative specifications, called *annotations*, of the logic of the query forms and reports that will be generated, and (iii) a set of template presentation libraries. The output is a set of query forms and reports that provide automated query construction and report formatting in order for the end users to query and browse the underlying XML data. Thus XQForms separates content (given by the XML Schema of the data), query form logic (specified by the annotations) and presentation of the forms and reports. The system architecture is modular and consists of four main components: (a) a collection of *query form controls* that incorporate query capabilities and allow parameter passing from the end users via the form page. A set of query form controls makes up a query form. (b) An *annotation scheme* for binding these controls to data elements of the XML Schema and for specifying their properties, (c) a *compiler* for creating the HTML representation of the query forms, and (d) a *run-time engine* that constructs and executes the queries against the XML data and renders the query results to create the reports.

General Terms

Design, Standardization, Languages.

Keywords

Query Forms & Reports, XML Query Language, XML, XSL.

1. INTRODUCTION

Vast amounts of information, stored in a variety of information systems, are made accessible to people worldwide through Web-based query forms that allow users to selectively view the information. At the same time, XML provides a powerful and simple way to represent and exchange structured and semistructured data.XML is being widely adopted and increasing amounts of data are made available and are exchanged in XML format [16, 17]. XML Schemas [14] are used to model the information and an XML query language [13] allows users and applications to select, extract and filter XML data, which may come from XML files or from other information sources, such as relational databases, that export an XML Schema of themselves.

Powerful access to XML-modeled data through query forms accessible via browsers and other Web-enabled devices is crucial. Moreover, recent research on how to develop and maintain web sites [3, 5, 11] indicates that, as the interaction between the Web and end users becomes more complex and information-rich, it is important to separate

- the page *information content* and how it is organized in the back-end,
- the *logic* of the web site pages and
- the *presentation* that renders the data for a specific device.

XQForms focuses on declaratively defining and generating query forms and reports for XML data modeled by XML Schemas. As Figure 1 illustrates, the end users submit form pages, which construct and issue XML queries to an XML data server and the corresponding XML results are rendered to HTML in order to produce the report pages. XQForms accomplishes the separation of content, logic and presentation by representing an XQForm by three components: an XML Schema, which represents the information content (i.e. models the data), an *annotation* that specifies declaratively the logic of the query forms and reports, and template presentation libraries that are used to produce a presentation of the generated pages.

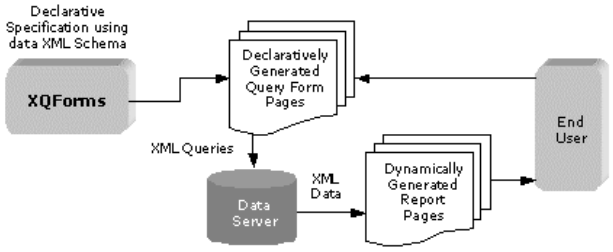

Figure 1 - XQForms Overview

Each one of these components is developed by, potentially, different people, namely the *data modeler*, the *annotations author* and the *web designer*. The components are independently customizable and extensible and allow for modular form development. For example, the annotations author can change the form's logic, while at the same time the web designer is adjusting the look and feel of the presentation. By providing this development flexibility, XQForms offer significant reduction in development and maintenance costs of a Web site.

*This work was performed while the author was at Enosys Markets Inc.

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

The output of XQForms is a set of query forms and reports that support automated query construction and report formatting and provide advanced functionality to the end users in terms of querying and browsing the underlying XML data. XQForms can accommodate any of the XML query languages recently proposed [4, 6], as well as any presentation language for rendering of results (e.g., HTML, WML, XML, etc.) A summary of the functionality that the XQForms-generated forms and reports provide to the end user is the following:

- Ability to express arbitrary selection and projection conditions on the query form.
- Ability to express arbitrary sorting conditions.
- Control on the number of elements that are returned from the data server.
- Advanced navigation capabilities in the returned reports.
- Ability to dynamically summarize and filter the query results on the reports pages.

XQForms delivers this rich functionality while providing rapid implementation of the Web site, since the query construction and the report formatting are automated, and the separation of form content, logic and presentation clearly decouples web design from the forms and reports development.

In the following section we introduce our running example of an XQForm. In Section 3, we present in detail the development process of the XQForms and we describe the main modules of the XQForms architecture. In Section 4 the annotation scheme for defining the logic of the XQForms is described. Section 5 describes the automatic creation of reports and Section 6 presents the XQForms Editor, a graphical interface for the currently available implementation of XQForms. Sections 7 and 8 discuss related work and offer some conclusions.

2. XQFORMS EXAMPLE

In this section we introduce our running example of an XQForm for querying and browsing sensor products that we will use to illustrate the XQForms development process. The set of form and report pages and the functionality they provide is displayed in Figure 2.

The Query Form on the left panel consists of a set of query elements, such as the *Manufacturer* selection list and the *Part Number* text box. Each such query element is generated by a query form control provided by XQForms that is bound to a data element of the proximity sensors' XML Schema (shown in Figure 3). The binding between the query form control and the data element to be queried is specified by an annotation to the XML schema of the data (XQForm annotation). In the lower left corner, the user can determine the sort-by list and the type of sorting (ascending, descending) by choosing and adding elements from a drop-down list. The possible values contained in the drop down list are also determined by a query form control provided by XQForms, and again the binding of the data element (such as `output_type`) to the particular control is specified by the XQForm annotation. By using the query elements on the form, the end user is able to constrain desirable characteristics of the product being sought, such as *Manufacturer*, *Body Type*, *Dimensions*, etc., and then, submit the query form.

The Report page is displayed on the right panel. The main entity returned is a proximity sensor product. The columns shown in the report correspond to elements of the XML Schema of the proximity sensors in Figure 3. Both the main entity of the report and the data

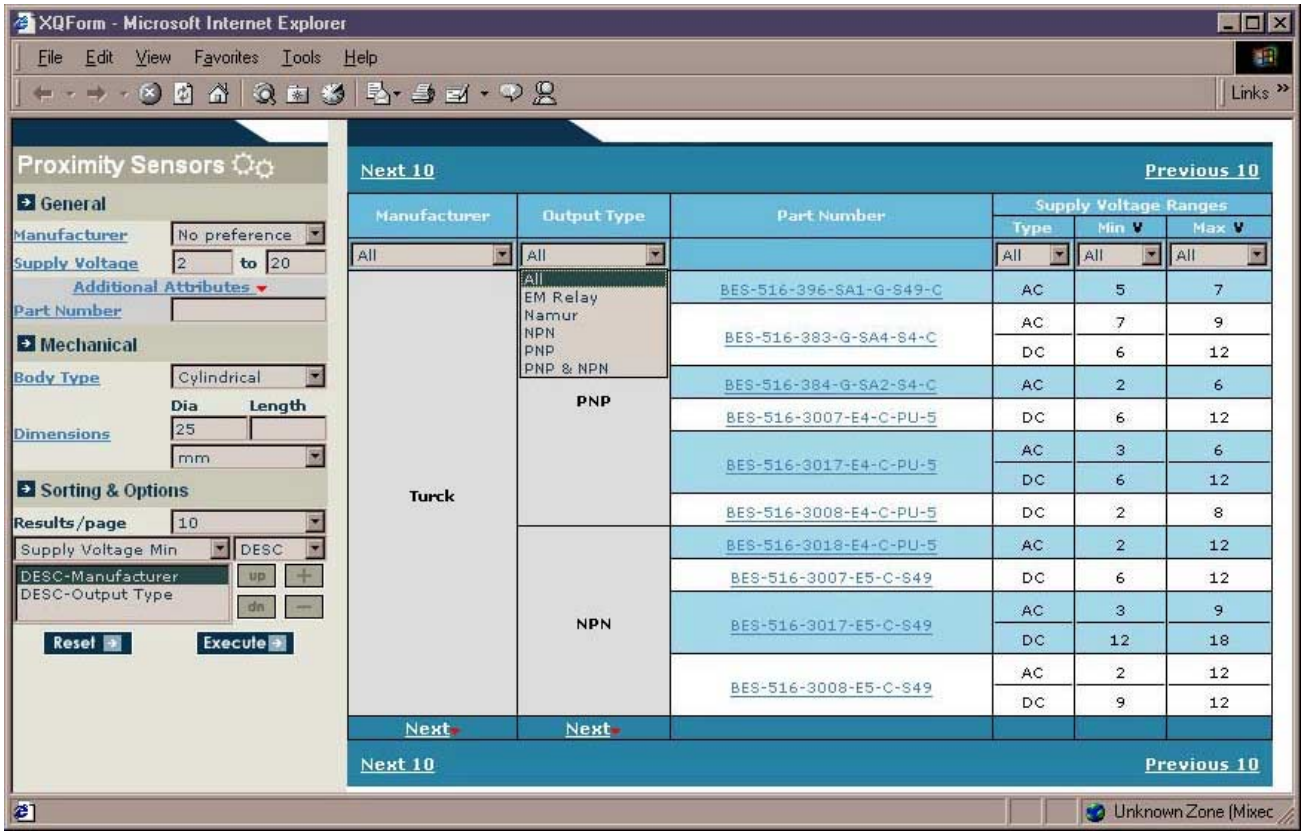

Figure 2 - XQForms-generated forms and reports

elements to be shown in the columns of the report are specified in the XQForm annotation. The structure of the page follows the XML Schema of the proximity sensors. Notice, for example, that a sensor may have multiple supply voltages, i.e., `supply_voltage` is a repeatable element. In that case, multiple triplets of voltage type, min and max appear for the each sensor.

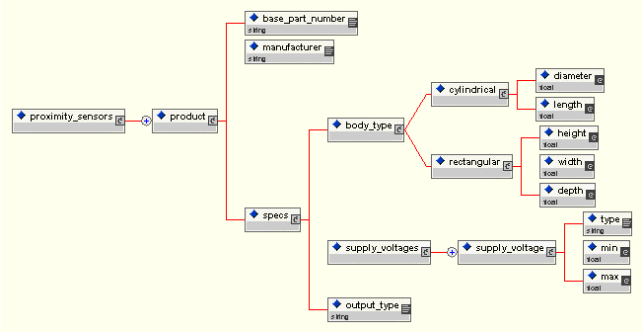

Figure 3 - Sample Data XML Schema

Under the headers of the table there are summarization/filtering lists that show the possible values for each column for the specific report. By selecting one of these values, the results are constrained to this particular value. The XQForm annotation specifies which elements in the XML Schema the report will include summarization/filtering lists for.

Note that the elements added by the end user to the sort-by list of the query form are displayed grouped in the report, like the Manufacturer and Output Type values. Finally, note that the overall presentation of the pages displayed in Figure 2 (fonts, color scheme, etc.) is automatically generated by XQForms and can be easily customized by a web designer using standard web design tools and methods (e.g., Cascading Style Sheets).

3. XQFORMS' DEVELOPMENT PROCESS

As Figure 4 indicates, XQForms' development process consists of three steps. During the first step, the *design phase*, the annotation author declaratively specifies the logic of the forms and reports to be generated using the annotation scheme. In the second step, the *compilation phase*, the presentation of the forms and reports is generated from the compiler module of XQForms. In the last step, the *run-time phase*, the end users access the generated pages and the run-time engine module constructs queries out of the submitted values and issues them against the data server in order to render the returned XML data to the reports that are sent back to the end users.

More specifically, in the design phase, the starting point of developing a single XQForm is the XML Schema, which describes the structure of the XML data to be queried. Figure 3 displays in graphical form, provided by the XML Authority® schema editor, part of the XML Schema for proximity sensors that is used in our running example, expressed in the XML Schema Definition (XSD) language. From this schema, the annotation author uses the *annotation scheme* to create a declarative specification of the form and report to be generated (an *annotation*). The annotation scheme is a language that defines an easily extensible collection of query form controls, such as text boxes and selection lists, their properties, and how they are bound to data elements of an XML Schema. It also allows the expression of potential dependencies between these controls, and the specification of the data elements that will appear in the report. An annotation is an instance of the annotation scheme that binds particular query form controls to elements in a particular XML Schema and sets their properties. The annotation describes the logic of the XQForm, is saved in an XML file, and is used by the run-time engine to produce a query upon the form submission.

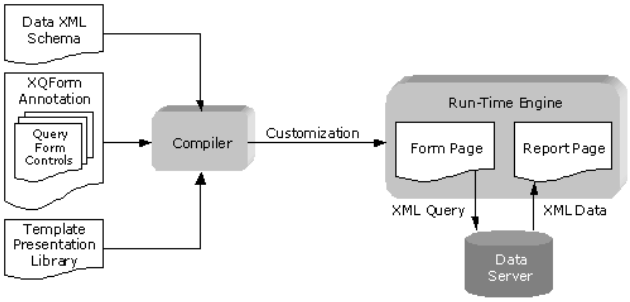

Figure 4 – XQForms' Development Process

In the compilation phase, the XQForms compiler takes as input the XML Schema, the XQForm annotation and a predefined template presentation library, and outputs a presentation of the query form and report pages. The template presentation library is used to translate the XQForm annotation into the pages that the end user interacts with. In particular, each query form control has a presentation template that is instantiated during the compilation based on the settings for the query form control properties in the XQForm annotation. As a result, query elements such as text fields, drop-down selection lists, etc., appear on the query form as shown in Figure 2. Also, an XSL script can be generated during this phase that will be used during run-time to create the report pages from the query results, as explained in more detail in Section 5.

Finally, in the run-time phase, when the end user requests an XQForm, she gets back the query form page as a response from where the XQForms run-time engine resides. At this step, the run-time engine receives the submitted parameters from the end user and constructs an XML query that retrieves the data elements to be presented in the report, constrained according to the conditions imposed from the query form controls. In particular, each query form control incorporates query capabilities such as selection conditions. The query form controls included in the XQForm annotation contribute accordingly to the query constructed by the run time engine. The constructed query is submitted to the data server and the returned XML data are rendered from an XSL processor and sent back to the end user. The XSL processor uses the XSL script that was created during the compilation phase.

3.1 XMAS Query Language

XQForms communicates with the data server by using the XMAS (XML Matching And Structuring) query language [8]. The XMAS query in Figure 5 is an example of the queries that the XQForms run-time will produce from the pages in Figure 2. We briefly explain the XMAS query language in the next paragraphs. It is straightforward to adjust the query form controls to produce query statements of most proposed XML query languages [13].

The semantics of XMAS are similar to OQL. The query consists of two parts or clauses: the **construct** and **where** clauses. The **where** clause specifies which elements of the input XML source are needed to produce the output and it is composed of *path expressions* similar to OQL's. A path expression is matched against the source and results in bindings to variables for XML fragments that matched the

expression. The **construct** clause specifies how the output should be produced from the fragments supplied by the **where** clause.

The query shown in Figure 5 is generated from the query form of Figure 2. In line 15 the path `proximity_sensors.product` is matched against the XML source `proxSource`. Variable `$P` receives the resulting bindings. In lines 16-17 the element `base_part_number` and the value of the `manufacturer` element are extracted from every binding of `$P`, to be included in the query result. In line 18, the variable `$SPEC` binds to the element `specs` so that the condition on `diameter`, shown in Figure 2, can be imposed in line 19 and the value of the `output_type` element retrieved in line 20.

```
1  CONSTRUCT
2  <answer>
3    <product>                                % start with the main result unit
4                                             % that contains the elements to be presented
5      <manufacturer>$M</manufacturer>        % manufacturer element
6      $BPN ($BPN)                            % base_part_number element (one for each $BPN)
7      <specs>                                % reconstruct the specs element
8        <output_type>$OUT</output_type>      % output_type
9      </specs>
10   </product> ($M, $OUT, $P)                % group by $P, because spec is repeatable in source
11             order by(desc $M, desc $OUT)   % order by $M (manufacturer) and $OUT (output_type)
12 </answer> ()                               % create one answer element (= for each ())
13
14 WHERE
15 proxSource proximity_sensors.product $P    % get product elements from source proxSource
16 AND $P base_part_number $BPN               % get base_part_number element
17 AND $P manufacturer._ $M                   % get manufacturer's value
18 AND $P specs $SPEC                         % get specs element
19 AND body_type.cylindrical.diameter._ <= 25 % get diameter's value and apply the condition
20 AND $SPEC output_type._ $OUT               % get output_type's value
```

Figure 5 - XMAS Query

The **construct** clause specifies how the XML output is assembled from the variable bindings produced from the **where** clause. The **construct** clause uses tree patterns along with grouping and sorting expressions. In our example, in line 2 an `answer` element is constructed as the root of the XML output. Inside it, the element `product` is constructed. Inside the `product` element, we construct all elements that are included in the report shown in Figure 2 (`supply_voltage` is omitted from the query for simplification) by following the exact structure of the data XML Schema (lines 5-9).

There are two kinds of grouping expressions used in the above **construct** clause:

- `$V {$V1, …, $Vn}`, as in line 6.
- `element {$V1, …, $Vn}`, where `element` is an element pattern, as in line 10.

The list of variables included in the curly brackets is called a *collection list*. Both expressions create a list of elements or values for each distinct combination of variable bindings in the collection list. If the collection list appears in the scope of another collection list, as in the case of `{$BPN}` in line 6 appearing within `{$M, $OUT, $P}` in line 10, then the inner collection list produces a list of elements or values that occur within a distinct combination of variable bindings of the outer collection lists.

In our example, in line 6, one `base_part_number` element is created for each binding of the `$BPN` variable within each distinct combination of the `$M, $OUT` and `$P` variables. In line 10, one `product` element is created for each binding of the `$M, $OUT` and `$P` variables. Finally, in line 11, the sorting expression sorts the `product` elements by the variables `$M` (`manufacturer`) and `$OUT` (`output_type`), again as specified in Figure 2.

4. ANNOTATION SCHEME

The annotation scheme is a language that defines an easily extensible collection of *query form controls*, such as text boxes and selection lists, their properties, and how they are bound to data elements of an XML Schema. It also allows the expression of potential dependencies between these controls, and the specification of the data elements that will appear in the report. The annotation scheme itself is modeled by an XML Schema, part of which is shown in Figure 6 below. Particular XQForms are defined via instantiations of this annotation scheme, called *XQForms annotations* (or simply annotations). In this section we elaborate on the query form part of an XQForm, which is specified by the `form` subelement of the `xqform` root element. Section 5 presents in detail the specification of an XQForm report.

Continuing with our running example, we use the data XML Schema in Figure 3 to generate the XQForm that appears in Figure 2.

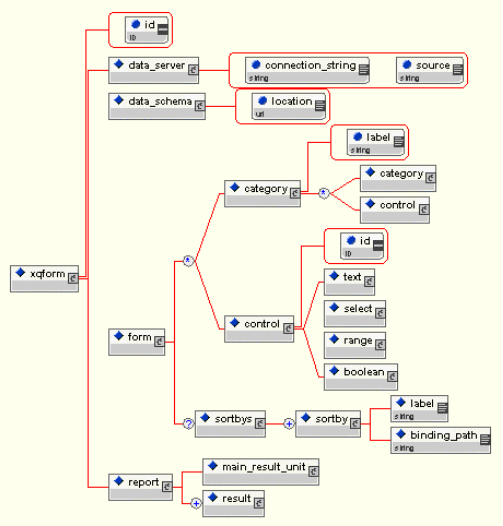

Figure 6 - Annotation Scheme Overview

4.1 Forms and Query Form Controls

The building blocks of the query part of an XQForm are the query form controls, referred to simply as 'controls' in the rest of the presentation. Each consists of two components, the *server-side* component and the *client-side* component. The *server-side* component implements the query capability of the control, i.e., it is responsible for constructing the appropriate selection condition or sorting condition that will potentially be included in the XML query when the user submits the form with appropriate input parameters. The *client-side* component presents the control on the query form page, and is the *instantiated presentation template* for the control, as mentioned in Section 3. This architecture allows the specification of the query capabilities of the control to be decoupled from their presentation, as in [7]: the one defines the way to construct query conditions, while the other defines the way to interact and exchange data with the end user.

The current implementation of XQForms supports five types of controls, as Figure 6 indicates: `text`, for conditions involving string and relational predicates, `range`, for range conditions, `select`, for conditions involving a set of constant values and string, relational and element existence predicates, `boolean`, for boolean conditions, and `sortbys`, for specifying the data elements that the end user can sort the query result by in the report page. This set of controls is easily extensible depending on the query capabilities we want to incorporate.

As shown in Figure 6, the `form` subelement of the `xqform` element contains the specifications of the controls. The details of these specifications are not shown in Figure 6. The details of the specification for the `select` control are shown in Figure 8. The annotation scheme organizes the controls into an arbitrary hierarchy of categories as the structural recursion involving the `category` element denotes. Each `category` has a `label` attribute and contains a set of controls and nested categories. This hierarchy is useful for the management and customization of large forms. In our running example displayed in Figure 2, the XQForms annotation (which as we explained is an instance of the annotation scheme) defines a first-level category with the label "Proximity Sensors" and organizes the query form controls into two second-level categories named "General" and "Mechanical."

Let us explain in more detail how the end user can impose a condition on the `manufacturer` element of the XML Schema in Figure 3 using a `select` control. The lifecycle of a query form control involves all three phases of the development process of XQForms, shown in Figure 4, and is summarized in Figure 7.

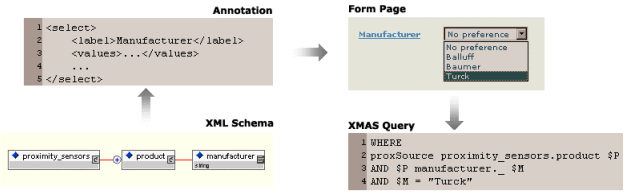

Figure 7 - Query Form Control's Lifecycle

1. In the design phase, the annotation author decides which data elements in the XML schema she wants to include in the query form. For each one of them she picks a control and sets its properties that are given in the annotation scheme. For the `manufacturer` element, a `select` control is chosen and its properties are set according to the relevant part of the annotation scheme shown in Figure 8. In particular, the `values` property is set by:
   a. specifying the values of the select list for the control in the `label` subelement in Figure 8
   b. specifying the values as they appear in data, if different that `label`, in the `data_value` subelement
   c. specifying the elements of the data XML Schema these values bind to in the `binding_path` subelement, and
   d. specifying which predicate (such as ==, <=, etc) will be used in association with the values.

   An example annotation setting these properties is shown in Figure 9 and described in detail later in this section.
2. In the compilation phase, the XQForms compiler takes as input the annotation and the template presentation library. The library contains a template for each control. The compiler instantiates the template and thus creates the client-side component of the control. The client-side component of the control is included on the query form page and presented to the end user. In this example, the presentation template for the `select` control renders the manufacturer control as a single-selection drop-down list including the possible values specified in the design phase. Note that this is one way to present this control. Another way would be a set of radio buttons, or a multiple-selection drop-down list. In these cases the annotation scheme remains the same but the presentation template changes.
3. When the end user hits the XQForm, the run-time engine takes as input the annotation and creates the server-side component of each control. This component receives the parameters the end user submits and contributes accordingly to the XMAS query sent to the data server. Figure 7 shows the part that the server-side component of the manufacturer control contributes to the XMAS query when the *Turck* value is submitted via the query form. As can be seen in Figure 7 and the annotation shown in Figure 9, the XML Schema element specified in the binding path of the value *Turck* is used to bind variable `$M` to the `manufacturer` data element of the *proxSource* XML data source. Similarly, the equality predicate is used in the query to impose the condition on the *Turck* submitted parameter, as specified in the annotation. If multiple values were selected, the condition would be a disjunction of these values.

Figure 8 shows in detail the specification of the `select` control as it appears in the annotation scheme and how its properties are structured. The set of properties common to all controls include the `label` of the control and the `info`. The info is 'attached' to the label and is rendered as a hyperlink to an explanatory text. The `additional` element specifies if the control is initially hidden, as is the *Part Number* control, rendered in darker background color, in Figure 2. The `unit` element specifies if the bound data elements have a specific measuring unit and the precision that will be used to show the returned values in the report. Controls can be easily built that give the ability to the end user to choose among several units in which she can express the parameters[1], like the *Dimensions* control in Figure 2. The rest of the elements are specific to the `select` control.

The `values` element models the possible values of the `select` control. These values can either be specified manually using the `value` elements or they can be extracted from the data at run-time using the `dynamic` element. The `value` elements can bind to different data XML Schema elements via different binding paths and can use different predicates to apply a condition to the XML data. Their `label` and `data_value` subelements give the option to the author to rename a data value to something more meaningful to the end user, because the possible data values might be constrained by other factors (e.g., whitespace is not allowed for XML element names). The `dynamic` element is used when the possible values appearing in the form are extracted directly from the XML data by executing an XMAS query. This feature is very useful in the case of frequently updated data. The `multiple` element determines if the end user will be allowed to select multiple values, in which case a disjunction of the conditions generated from each value will be included in the XMAS query. The `dependencies` element is used from the annotation author to express dependencies among the controls and is further discussed in section 4.2.

[1] XQForms already includes controls, beyond the basic five we described, with this capability.

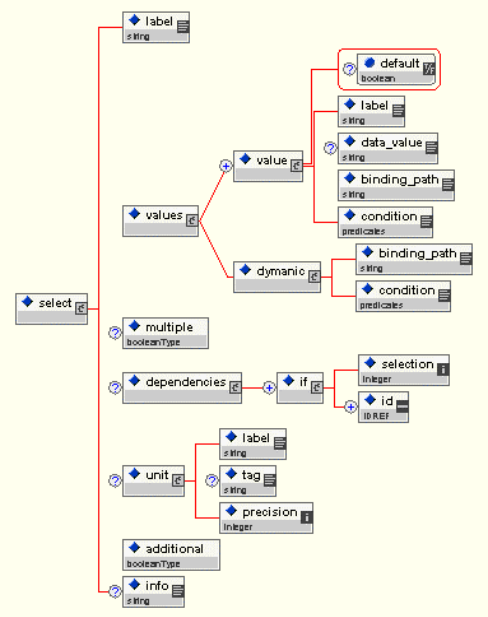

Figure 8 - `select` Control Detail Specification

The part of an example annotation referring to the *Manufacturer* `select` control in Figure 2 is shown below. Note the correspondence of the properties of the control to its presentation and the condition that it imposes in the XMAS query as they appear in Figure 7. The presentation of the control places the *Manufacturer* label next to the drop-down list, which lists the possible values based on the `label` subelement of each `value` element. If the end user chooses the *Turck* value, the `binding_path` and `condition` subelements of the specific `value` element are used to construct the XMAS query fragment shown in Figure 7.

```
<select>
  <label>Manufacturer</label>
  <values>
    <value default="true">
      <label>No preference</label>
      <data_value>any</data_value>
      <binding_path></binding_path>
      <condition>EQ</condition>
    </value>
    ...
    <value>
      <label>Turck</label>
      <binding_path>
proximity_sensors/product/manufacturer
      </binding_path>
      <condition>EQ</condition>
    </value>
  </values>
  <multiple>false</multiple>
  <additional>false</additional>
</select>
```

Figure 9 - Example `select` Control Annotation

4.2 Expressing Dependencies

The `dependencies` element of the `select` control in Figure 8 allows the annotation author to express dependencies among controls as the example in Figure 10 demonstrates. For each body type, a different set of controls has to be involved in querying the dimensions of the body type, and the appropriate set of client-side components for querying these dimensions has to be shown on the query form. The annotation author uses a `select` control to query the `body_type` element of the sensors' XML Schema, which has one value with label *Cylindrical* that binds to the `cylindrical` subelement of the XML schema and one value with label *Rectangular* that binds to the `rectangular` subelement. The author also binds controls to the dimensions of either body type elements. She wants the corresponding controls to appear whenever one or the other value is selected. So when the *Body Type* control takes the value *Cylindrical* the client-side components of the controls for `diameter` and `depth` should appear on the page, and when *Rectangular* is selected, the corresponding components for the controls for `height`, `width` and `depth` controls should appear.

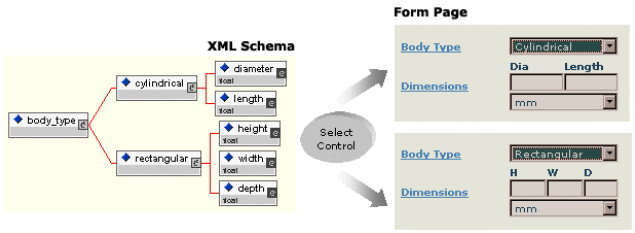

Figure 10 - Expressing Dependencies

The `dependencies` element gives the ability to express these dependencies. From Figure 8, the `if` subelement of the `dependencies` element captures the different cases for each possible value of the `select` control. The `selection` element is set to one of the `label` elements and the `id` identifies a query form control that has to appear upon value selection[2]. The action carried out when a dependency is detected is defined in the presentation templates of the query from controls, and thus can easily change. So, instead of the XQForms default 'show/hide' action, the controls could get disabled and enabled. Note that dependencies can be expressed among controls that bind to any element in the XML Schema.

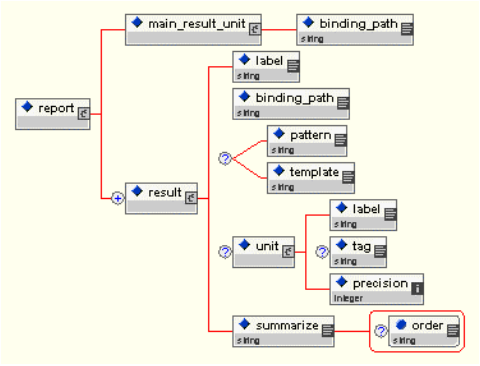

Figure 11 - Report Specification

5. REPORTS

Using the annotation scheme shown in Figure 6, the annotation author specifies the data element that will appear on the report page of the XQForm. She does it by instantiating the `report` subelement of the `xqform` element, whose structure is shown in detail in Figure 11. Each XQForm needs exactly one `main_result_unit` element. It serves as a projection over an element of the XML Schema of the data. Usually it is the first

[2] Note in Figure 6 that every control has an `id` subelement.

repeatable element, as in our running example, where the main result unit is the `product` data element. But in big schemas it can be used to project out only a small subtree of it. All the data elements that are included in the report must be its subelements, since this element is always the first element constructed, under the `answer` element, in the XMAS query produced from the XQForms run-time engine.

The `result` subelements of the `report` element capture the structure of the report. Each `result` element specifies which data subelement of the main result unit will be included in the report and its structure is quite simple. It has a `label`, a `binding_path`, which is pointing to the data element via an XPath, and a `summarize` element that indicates whether the distinct values of the corresponding element should be presented in the report, and if so, ascending or descending. The `unit` element has the same semantics with the one in the query form controls' specification. The `pattern` and `template` elements are used to customize the presentation of the specific data element and are further discussed in section 5.2. The part of the annotation referring to the report page in Figure 2 is shown below:

```
<report>
  <main_result_unit>
    <binding_path>
      proximity_sensors/product
    </binding_path>
  </main_result_unit>
  <result>
    <label>Manufacturer</label>
    <binding_path>
proximity_sensors/product/manufacturer
    </binding_path>
    <pattern>
      concat(., ' Manufacturer')
    </pattern>
    <summarize order="ASC">
      true
    </summarize>
  </result>
  <result>
    <label>Part Number</label>
    <binding_path>
proximity_sensors/product/base_part_number
    </binding_path>
    <template>
      base_part_number.xsl
    </template>
    <summarize>false</summarize>
  </result>
  ...
  <result>
    <label>Min</label>
    <binding_path>
proximity_sensors/product/specs/supply_voltag
es/supply_voltage/min
    </binding_path>
    <unit>
      <label>V</label>
      <precision>0</precision>
    </unit>
    <summarize order="ASC">
      true
    </summarize>
  </result>
  ...
</report>
```

5.1 Report Formatting

When the end user submits the query form, an XMAS query is constructed from the run-time engine on the server side using the submitted parameters. This query is sent to the data server, and asks only for the data elements that have been included in the annotation from the annotations author. The data server responds with the XML data that satisfy the query. These data are rendered with an *automatically generated* XSL script and the resulting report is sent back to the end user's browser.

The idea of using templates is applied in the formatting of the report, where templates are used for the presentation of simple elements (leaves in the data XML Schema), one for each data type supported, and for the presentation of complex elements (elements that have subelements). Putting these templates together to construct the final XSL script is done by the XQForms run-time engine.

The run-time engine, in order to produce the final XSL script, takes as input the XSL templates for simple and complex elements, which are provided from the template presentation library, and the **construct** clause of the XMAS query. The **construct** clause reveals the structure of the XML data that are sent back from the data server. The engine applies a special XSL script, a *metaXSL* script, which follows the structure of the **construct** clause and attaches to the simple and complex elements the XSL templates that will render them in the presentation language. Essentially, a *metaXSL* script is applied to the metadata of the XML result data in order to produce the XSL script for the actual data.

In our running example, the final XSL script that renders the query result to the report shown in Figure 2 is constructed from several different XSL templates as Figure 12 shows. The functionality these XSL scripts encapsulate, and the way they are composed to form the final XSL script, are described next.

The template named *top.xsl* contains the main XSL *stylesheet* that is used to render the XML query result. This template is *static* in the sense that it is executed directly against the XML data and is the same for every XQForm. The XSL *templates* created from all the other *metaXSL* scripts are appended to this *stylesheet*. It also contains the declarations of all the global parameters that are passed to the final XSL script. The templates listed next are *metaXSL* scripts in the sense that they are dependent on the query executed and they change depending on the query result.

- *head.xsl*
  Displays the headers of the of the XML data, which can be arbitrarily nested as the *Supply Voltage Ranges* column in Figure 2 shows. The tree structure for the headers is constructed from the `result` elements, which are specified through the annotation scheme and the XMAS **construct** clause. The nodes of this structure are of the special type *header* and so they are presented using the same XSL template.
- *summarize.xsl*
  This template detects the columns that are summarized in the XMAS query and shows the drop-down lists under the leaves of the headers tree structure. These lists contain the possible distinct values of the corresponding columns. In this example, these lists are included in their own HTML forms that are submitted any time the end user selects a specific value in order to filter the data presented in the report.
- *body.xsl*
  This XSL template also follows the structure of the XMAS

construct clause and attaches the XSL templates for complex elements in order to reveal this structure visually. It also checks the data type of each simple element that is presented and attaches the corresponding XSL template to it.

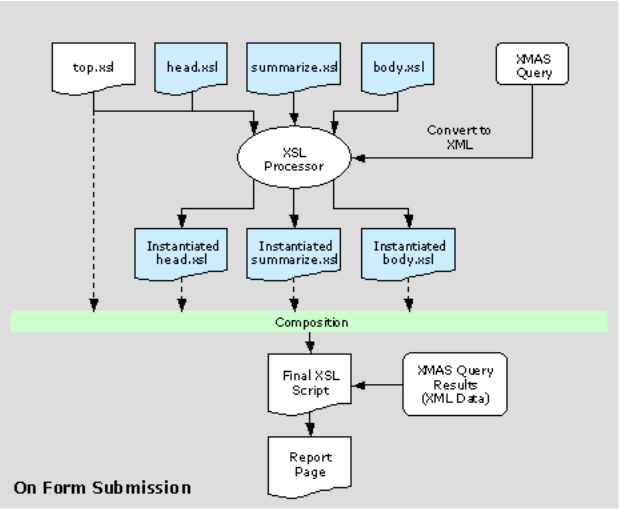

Figure 12 - Automatic Report Formatting

The paradigm of nested tables is heavily used for the visual translation of nested and/or repeatable elements. The visual presentation of tables is specific to the presentation language. As a structure, nested tables are quite general and intuitive. In particular, from a practical point of view, in an important application for query forms, complex product catalogs for e-commerce use nested tables as the basic presentation structure. Also, from a theoretical point of view, the complex value algebra uses the similar notion of nested relations as the basic structure [1].

Note that the automatic XSL script creation described above can be carried out in two different phases of the XQForms development process. The script can be created at run-time, as described so far, if we want to support dynamic report structure, e.g., personalized settings. This does not add any latency to the query response since the creation of the final XSL script can start as soon as the XMAS query is constructed and is carried out in parallel with the query execution. In this case, the web designer can only customize the presentation templates for simple and complex elements as the next section describes. The final XSL script can also be created during the compilation phase. The web designer can then customize the whole script and the XQForm uses this static customized script to render the reports on the fly.

5.2 Report Customization

As mentioned in the previous section, XQForms provide a separate XSL template for each data type of a simple element and a generic one for complex elements. Currently, the simple data types of the XML Schema specification that supported are `integer`, `float`, `boolean`, and `string`. The web designer can customize the default presentation of the data elements using either the `pattern` or the `template` element of the report specification of Figure 11. Using the first option, we can define a pattern, which is a standard XSL function that manipulates the data value for a specific element in the report. In our running example, we defined a pattern for the *Manufacturer* element that concatenates the element value with the ' Manufacturer' string (not shown in Figure 2).

The web designer can also use the `template` element to define an external XSL template that customizes the presentation of an element. In the sensors example, the `base_part_number` element is presented as a hyperlink to, possibly, a detailed product page. For this purpose, an XSL template is defined in *base_part_number.xsl*. The name of the XSL template is composed from the form name and the XPath that leads to the element that it will render. So the XSL template for the base part number is named `prox/proximity_sensors/product/base_part_number`, where `prox` is the name of the XQForm. It presents the value of the `base_part_number` element as a hyperlink to *some_target*, as the following code shows.

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template
name="prox/proximity_sensors/product/base_part_number">
    <xsl:element name="a">
      <xsl:attribute name="href">
some_target?base_part_number=
<xsl:value-of select="."/>
      </xsl:attribute>
      <xsl:attribute name="CLASS">
        rowText
      </xsl:attribute>
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

The XQForms run-time engine loads these templates and detects the bindings during the final XSL script construction. Finally, the styles applied to all presentation templates of XQForms are controlled from a central CSS file. The web designer can easily determine the look and feel of entire XQForms by changing the styles used across the presentation templates for the query form controls and/or the styles used in the XSL templates for simple and complex elements included in the report.

6. XQFORMS EDITOR

The XQForms Editor is a graphical user interface (GUI) for creating XQForms annotations. The basic editor screen is composed of the following three panes, as illustrated in Figure 13.

- The *schema* pane on the left displays the data XML Schema in a simple format.
- The *query form annotation* pane on the center displays the query form controls included in the form and provides an editor for the properties of each one of them.
- The *report annotation* pane, positioned on the right, displays the data elements that will be presented in the report.

7. RELATED WORK

The XML Forms Language [7] also defines forms independent of how they are rendered and presented to the end user. On the submission of the form, though, an XSL script called *Formsheet* is applied to the form values in order to transform them into an arbitrary XML structure instead of producing a query against XML data.

The same idea is adopted by XForms [12], a W3C working group that builds a specification on how to extend HTML forms. The goal is to enable the end user to communicate with the web server

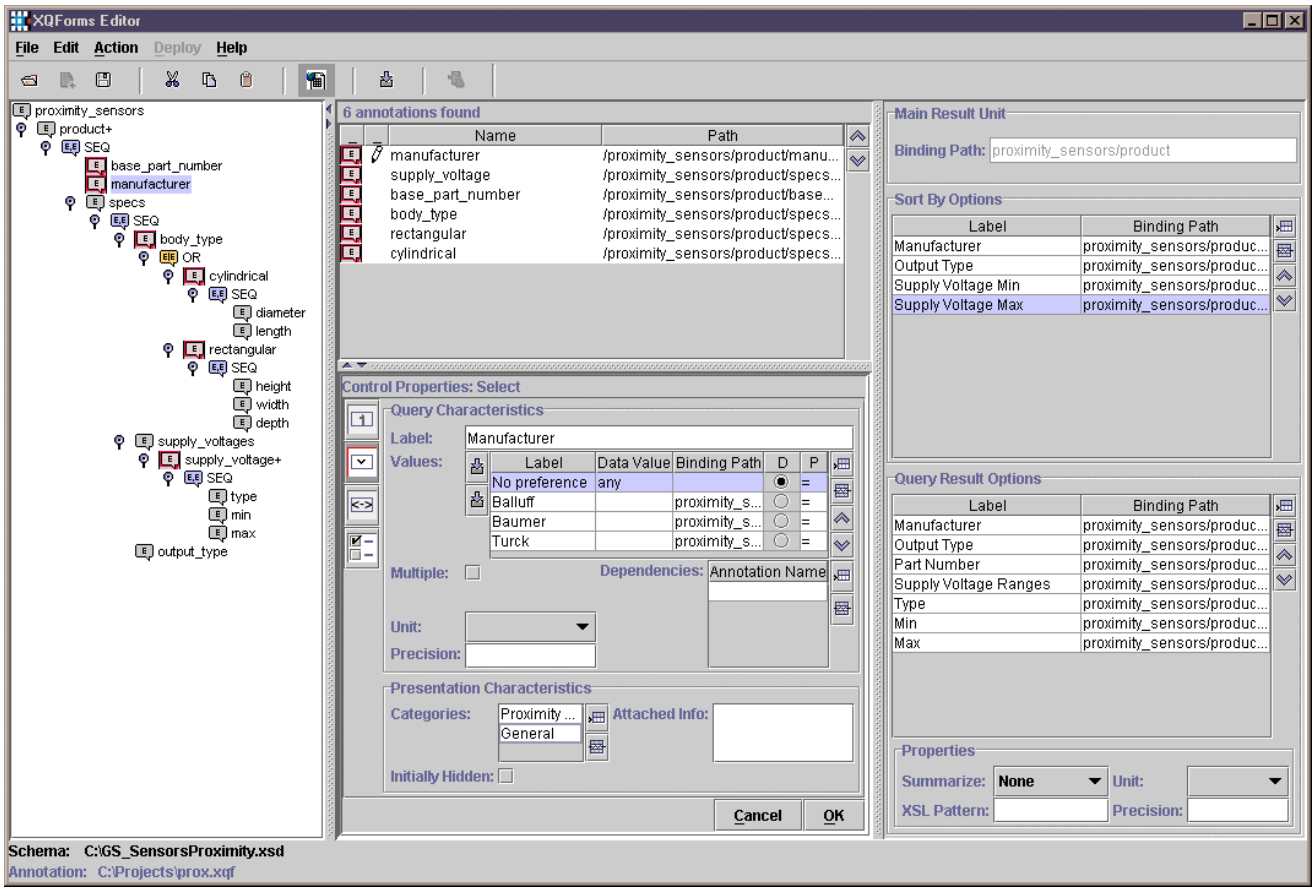

Figure 13 - XQForms Editor

through XML structured documents that conform to XML Schemas instead of the current method of a list of parameter/value pairs. XForms also tries to push as much functionality as possible to the end user's side. The idea of expressing dependencies between query form controls presented in this paper is also present in XForms. No capability for queries against XML data exists. We intend to fully implement the upcoming XForms specification that allows richer and more functional interfaces for data gathering and exchange.

Strudel [5] is an integration system that is based on labeled directed graphs for both data and site modeling. The query language, StruQL, is used to define both the way data is integrated from multiple sources (data graph), and the pages that make up the web site and the way they are linked (site graph). A form is defined on the edges of the site graph by specifying a set of free variables in the query that produces the node underneath the edge. These variables will be bound by the controls of the query form. A major difference between this and our approach is that, in Strudel, the form designer needs to write the StruQL query to produce the reports. Another significant difference is that Strudel does not enable the automatic rendering of query results.

There are also many web site development platforms, like Microsoft's Visual InterDev, that implement an automated process to graphically build advanced query form and report pages with summarization and navigation capabilities. These platforms, though, operate only on top of relational databases and there is no separation between the specification and the implementation of the pages.

The XQForms functionality is influenced from database interfaces and techniques to query XML and object oriented data [2, 9]. These systems use a graphical environment to navigate and drill-down into the instance of the data model by blending querying and browsing either on a tree or a graph structure. The summarization functionality is influenced from the work in [10], where advanced techniques and data structures for summarizing are presented.

8. CONCLUSIONS

This paper has presented the XQForms generator for declarative specifications of Web-based query forms and reports for XML data. The architecture consists of four main components:

- The *query form controls* that are the building components of a query form.
- The *annotation scheme* that allows the rapid and declarative specification of XML query forms and reports by binding query form controls to elements defined in the XML Schema of the data.
- The *compiler* that translates the declarative specification of an XQForm to one or more query form and report pages.
- The *run-time engine* that performs the automatic query construction and report formatting.

XQForms provide templates and styles for a default rendition of query form controls on the query form page, and of simple and complex XML elements on the report page. Web designers have the option to customize the layout and the look and feel of the presentation either before or after the compilation of XQForms using widely available Web-authoring tools.

In the near future, we plan to extend our model with navigation capabilities similar to the ones that Strudel [5] and WebML [3] support for linking static and dynamic pages.

An on-line demonstration of the example presented in this paper can be found at: http://www10.enosysmarkets.com

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu, Foundations of Databases, Addison Wesley, 1996.

[2] M. Carey, L. Haas, V. Maganty, J. Williams: PESTO: An Integrated Query/Browser for Object Databases, in Proceedings of VLDB 1996, pp. 203-214.

[3] S. Ceri, P. Fraternali, A. Bongio: Web Modeling Language (WebML): a modeling language for designing Web sites, in Proceedings of WWW9, Toronto, Canada, May 1999.

[4] D. Chamberlin, J. Robie, and D. Florescu: Quilt: An XML Query Language for Heterogeneous Data Sources, In Lecture Notes in Computer Science, Springer-Verlag, 2000.

[5] M. Fernandez, D. Suciu and I. Tatarinov: Declarative Specification of Data-intensive Web sites, Proc. Workshop on Domain Specific Languages, 1999.

[6] D. Florescu, A. Deutsch, A. Levy, D. Suciu, and M. Fernandez: A Query Language for {XML}, in Proceedings of Eighth International World Wide Web Conference, 1999.

[7] A. Kristensen: Formsheets and the XML Forms Language, in Proceedings of WWW9, Toronto, Canada, May 1999.

[8] B. Ludascher, Y. Papakonstantinou, P. Velikhov, Navigation-Driven Evaluation of Virtual Mediated Views, In Extending Database Technology (EDBT) 2000.

[9] K. Munroe, Y. Papakonstantinou, BBQ: A Visual Interface for Browsing and Querying XML, In Visual Database Systems (VDB) 2000.

[10] J. Shafer, R. Agrawal: Continuous Querying in Database-Centric Web Applications, in Proceedings of WWW9, 2000.

[11] S. Staab, J. Angele et al.: Semantic Community Web Portals, in Proceedings of WWW9, 2000.

[12] M. Dubinko et al.: XForms Requirements, W3C Working Draft 21 August 2000. http://www.w3.org/TR/xhtml-forms-req

[13] D. Chamberlin et al.: XML Query Requirements, W3C Working Draft 15 August 2000. http://www.w3.org/TR/xmlquery-req

[14] D. Fallside, XML Schema Part 0: Primer, W3C Candidate Recommendation 24 October 2000. http://www.w3.org/TR/xmlschema-0/

[15] S. Adler et al.: Extensible Stylesheet Language (XSL) Version 1.0, W3C Working Draft 18 October 2000. http://www.w3.org/TR/xsl/

[16] Microsoft BizTalk Server. http://www.microsoft.com/biztalk/

[17] OASIS, the Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org