Building XML Query Forms and Reports with XQForms*

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

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

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

Abstract

XQForms is the first generator of web-based query forms and reports for XML data. XQForms takes inputs (i) an XML Schema that models the source data to be queried and presented, (ii) a declarative specification, called *XQForm annotation*, 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 form and report pages that provide automated query construction and report formatting so that the end users can query and browse the underlying XML data. XQForms separates content (given by the XML Schema of the source data), query form semantics (specified by the annotations) and presentation of the pages (provided by the template library). The system architecture is modular and consists of four main components: (a) a collection of *query controls* that generate query fragments based on the values that the end user submits via the query form page; a set of query controls makes up a query form; (b) a graphical user interface, called *XQForms Editor*, for building the annotations; (c) a *compiler* that creates the query form pages based on the annotations and the template presentation libraries; (d) a *run-time engine* that constructs the queries that are getting executed against the XML data, and creates the reports by rendering the results using the template presentation libraries. All the components of the system have been implemented and an on-line demonstration is available at http://www.db.ucsd.edu/xqforms/.

Keywords

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

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, an increasing amount of sources export XML views of themselves [19, 20]. XML Schemas [18] are used to model the information and an XML query language [16] will allow users and applications to select, extract and filter XML data, which may come from XML files, XML databases [8, 13] or from other information sources, such as relational databases that export an XML view of themselves [1, 5]. Easy development of powerful form and report pages for querying and browsing such XML data and views will soon become important.

* This paper is an extended version of [12].
† This work was performed while the author was at Enosys Markets Inc.

XQForms declaratively defines and generates sets of query form and report pages for XML data modeled by XML Schemas. These pages support automated query construction and report formatting and provide advanced querying and browsing functionality to the end users. As Figure 1 illustrates, the end users complete query form pages, which construct and issue XML queries to a data server, and the corresponding results are rendered into HTML to produce the report pages. XQForms separates the querying functionality and presentation by constructing a set of query form and report pages, called *XQForm*, from two components: an *XQForm annotation*, or simply annotation, that specifies declaratively the query form and report pages based on the XML Schema that models the source data, and a set of *template presentation libraries* that are used for the presentation of the generated pages.

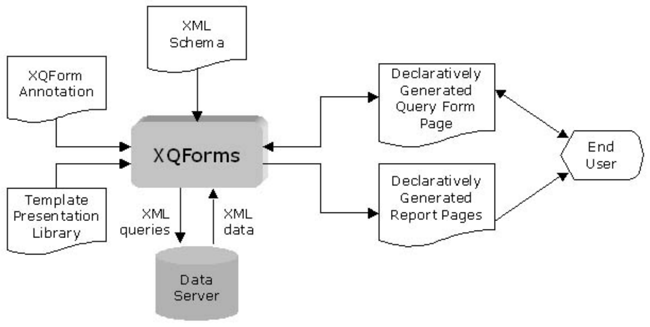

XML Schema
XQForm Annotation
XQForms
Template Presentation Library
XML queries
XML data
Data Server
Declaratively Generated Query Form Page
Declaratively Generated Report Pages
End User

Figure 1 XQForms Overview

Each one of these components can be developed at, potentially, different times and/or by different people and are independently customizable and extensible to allow for modular form development. For example, the person we call *annotations' author* can change the XQForm's specification by editing the annotation, while at the same time the *web designer* is adjusting the look and feel of the generated pages by changing the template presentation library. By providing this development flexibility, XQForms offers significant reduction in the development and maintenance cost of a web site. XQForms can easily be adapted to use any of the XML query languages recently proposed [4, 7], as well as any presentation language (e.g., HTML, WML, etc.).

In summary, XQForms-generated query form and report pages provide to the end user:

- 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.

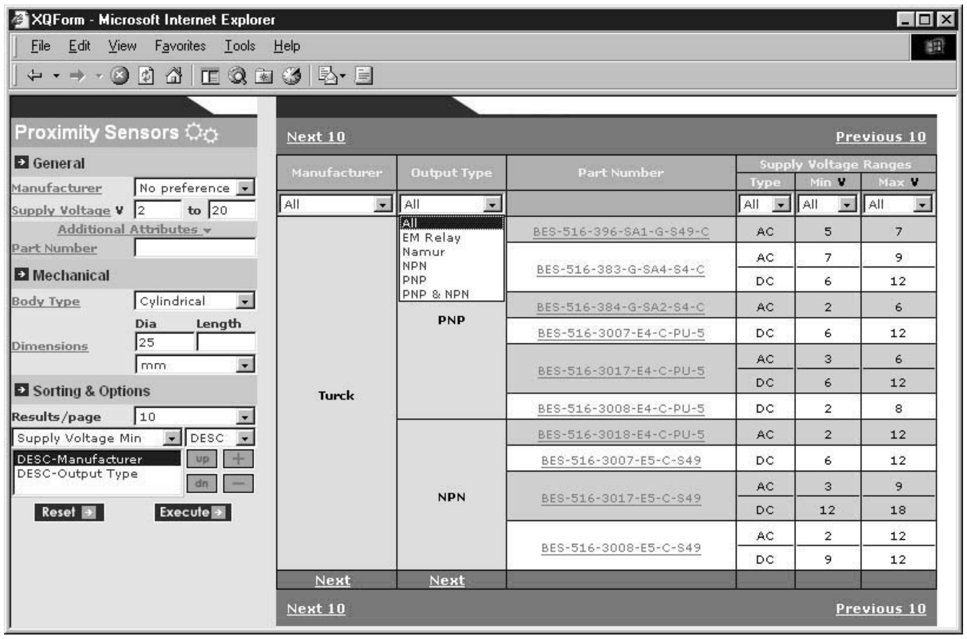
XQForms delivers this rich functionality while providing rapid development of the query form and report pages, since the query construction and the report formatting are automated, and the separation of querying functionality and presentation clearly decouples the tasks of the annotations' author and the web designer.

In the following section we present in detail the XQForms' architecture and its main components and we demonstrate the XQForms' lifecycle by introducing our running example. An on-line demonstration of the XQForms' components and the running example is available at http://www.db.ucsd.edu/xqforms/. In the same section, we also explain the interface between

XQForms and the data server. In Section 3 describes the syntax of the XQForm annotations, called *annotation scheme*, as well as the automatic creation of report pages. Section 4 presents the XQForms Editor, a graphical interface that builds XQForm annotations for the currently available implementation. Sections 5 and 6 discuss related work and provide conclusions.

2 XQForms' Architecture

In this section we illustrate the XQForms' functionality and lifecycle by introducing an XQForm that queries and browses sensor products and will serve as our running example. The interface between the XQForms and the data server is described in the last subsection.



Manufacturer	No preference	
Supply Voltage V	2	to 20
Part Number		
Body Type	Cylindrical	
	Dia	Length
Dimensions	25	
	mm	
Results/page	10	
Supply Voltage Min	DESC	

DESC-Manufacturer
DESC-Output Type

Output Type: All, EM Relay, Namur, NPN, PNP, PNP & NPN

Manufacturer	Output Type	Part Number	Supply Voltage Ranges		
			Type	Min V	Max V
All	All		All	All	All
Turck	PNP	BES-516-396-SA1-G-S49-C	AC	5	7
		BES-516-383-G-SA4-S4-C	AC	7	9
			DC	6	12
		BES-516-384-G-SA2-S4-C	AC	2	6
		BES-516-3007-E4-C-PU-5	DC	6	12
		BES-516-3017-E4-C-PU-5	AC	3	6
			DC	6	12
		BES-516-3008-E4-C-PU-5	DC	2	8
	NPN	BES-516-3018-E4-C-PU-5	AC	2	12
		BES-516-3007-E5-C-S49	DC	6	12
		BES-516-3017-E5-C-S49	AC	3	9
			DC	12	18
		BES-516-3008-E5-C-S49	AC	2	12
			DC	9	12

Figure 2 XQForms-generated forms and reports

2.1 Functionality

Figure 2 displays an example of a form and report page set and the functionality they provide to the end user for querying and browsing sensor products. The query form page, on the left panel, consists of a set of *form elements*, such as the "Manufacturer" selection list and the "Part Number" text box. The end user interacts with these form elements and constrains desirable characteristics of the sensors being sought, such as requiring the "Supply Voltage" to be in the range of 2 and 20 volts. In the lower left corner, she determines the number of sensors that will be displayed on the report page, and constructs the sort-by list and the type of sorting (ascending, descending) by adding options from the drop-down list.

The report page is displayed on the right panel and presents a subset of the sensors characteristics in a tabular form. The end user can navigate into the results by using the "Next XX" and "Previous XX" links. 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. Note also that the options added by the end user to the sort-by list of the query form are displayed grouped on the report page, as the "Manufacturer" and "Output Type" columns demonstrate. At the bottom of these columns, the "Next" link will return, for example, the sensors of the next "Manufacturer". A default overall presentation of the pages displayed in Figure 2 (fonts, color scheme, etc.) is automatically generated by XQForms and can be easily customized from the web designer using standard web design tools and methods (e.g., Cascading Style Sheets).

2.2 XQForms' Lifecycle

Let us now explain how the above functionality is built. The XQForms' lifecycle consists of four steps, as Figure 3 indicates. During the first step, the *design phase*, the annotation author declaratively specifies the form and report pages to be generated using the *XQForms Editor*, a graphical interface for building XQForm annotations (see Section 4.) In the second step, the *compilation phase*, the presentation of the form pages is generated from the compiler module of XQForms. In the *customization* phase, the web designer has the ability to customize the query form pages so that they match the look and feel as well as the structure of the web site in which the XQForm resides. In the last step, the *run-time phase*, the end users access the generated query form pages and the run-time engine module constructs queries out of the submitted values, issues them against the data server, and renders the returned XML data into the reports pages that are sent back to the end users.

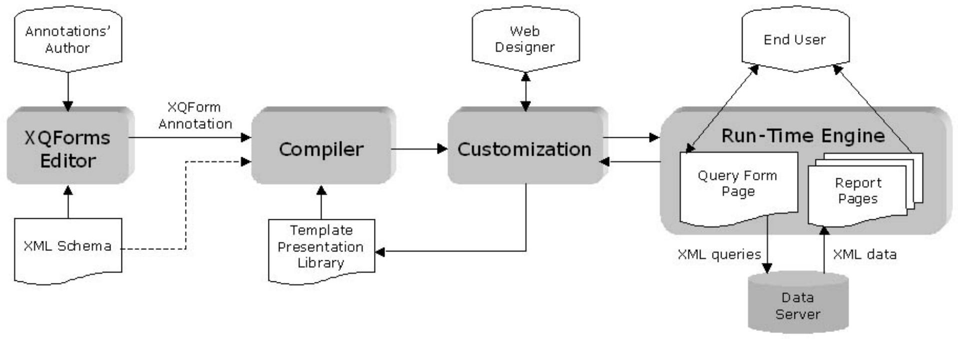


Figure 3 XQForms' Lifecycle

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 4 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. The annotations' author uses the XQForms Editor and this schema to create an XQForm annotation, which is a declarative specification of the form and report pages to be generated.

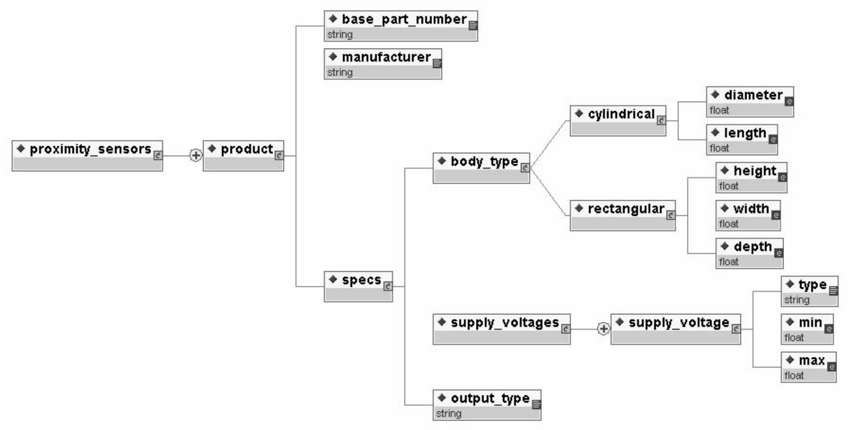
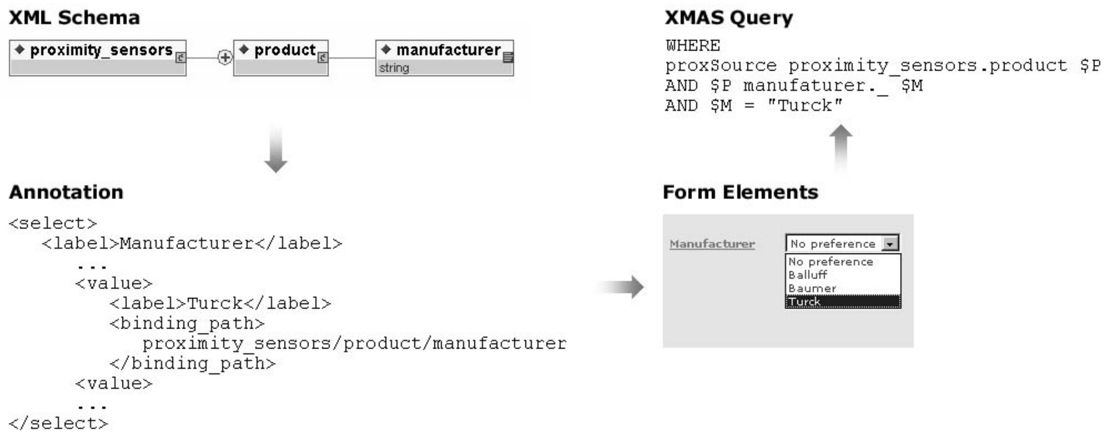


Figure 4 Sample Data XML Schema

The form elements, which appear on the query form page in Figure 2, are based on the *query controls* provided by XQForms. Each query control incorporates querying functionality and generates query fragments, such as selection conditions, based on the values the end user submits. The XQForm annotation binds these controls to the data elements of the proximity sensors' XML Schema to be queried. The lifecycle of a query control, shown in Figure 5, involves all four phases of the XQForms' lifecycle and is described in detail in Section 3.1. Part of an example annotation is shown in Figure 5, which binds the possible values of a drop-down list to the `manufacturer` element in the XML Schema.



```
WHERE
proxSource proximity_sensors.product $P
AND $P manufaturer._ $M
AND $M = "Turck"
```

```
<select>
   <label>Manufacturer</label>
      ...
      <value>
         <label>Turck</label>
         <binding_path>
            proximity_sensors/product/manufacturer
         </binding_path>
      <value>
      ...
</select>
```

Figure 5 Query Control's Lifecycle

The options contained in the sort-by drop down list in Figure 2 are also determined by a query control provided by XQForms, and again the binding of the data element (such as `output_type`) to the particular query control is specified by the annotation. Finally, the columns shown on the report page in Figure 2 correspond to data elements, which are also specified in the annotation. Notice that the structure of the report page follows the XML Schema of the sensors. For example, a sensor can have multiple supply voltages, i.e., `supply_voltage` is a repeatable element. In that case, multiple triplets of `type`, `min`, and `max` values appear for each sensor. The syntax of the XQForm annotations is specified by the annotation scheme, a language described in Section 3.

In the compilation phase, the compiler inputs the XML Schema, the XQForm annotation and a template presentation library to produce the presentation of the query form page that the end user interacts with. In particular, each query control has a presentation template that is instantiated during the compilation based on the settings for the query control properties in the XQForm annotation. As a result, the corresponding form elements such as text fields, drop-down selection lists, etc., appear on the query form page as shown in Figure 2. Figure 5 shows a label and a drop-down list containing the possible values that bind to the `manufacturer` element. These are the form elements produced from the presentation template for the `select` control. Also, an XSL script [15] could 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 3.3.

In the customization phase, the web designer adjusts the appearance and the structure of the query form page to that of the container web site. This phase can be repeated many times depending on the customization that is being done. As Figure 3 indicates, there are two feedback loops in the lifecycle since the web designer can change either the template presentation library, and then recompile, or change the compiled query form page directly, then test it in the run-time phase, and then change it again. The decision depends on the nature of the changes. Generic customization can be made to the library so that they can be applied to other XQForms for the same site/application. For this purpose XQForms makes use of templates that can be applied to specific data elements in the XML Schema. For example, the `base_part_number` element in Figure 4 is displayed as a hyperlink to, possibly, a detailed page of the sensor. Also, if an XSL script was generated during the compilation phase that will create the report pages during run-time, she can customize that too (see Section 3.3.)

Finally, in the run-time phase, when the end user requests an XQForm, she receives the query form page from where the run-time engine resides. At this point, the end user interacts with the query form page and through the form elements submits the desired values. The run-time engine receives these values and constructs an XML query that retrieves the data elements to be presented on the report page, constrained according to the conditions imposed from the query controls. Figure 5 shows the condition that the particular `select` query control imposes to the `manufacturer` element when one of the possible values of the drop-down list is selected. 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 either the static XSL script created during the compilation phase or the dynamic one that can be generated at run-time as Section 3.3 describes.

2.3 Data Server's Interface

In this section we give a brief presentation of the query language that XQForms uses to express queries, and the navigation interface provided by the data server.

2.3.1 XMAS Query Language

XQForms communicates with the data server by using the XMAS (XML Matching And Structuring) query language [10]. The XMAS query in Figure 6 is an example of the queries that the XQForms run-time engine will produce from the pages in Figure 2. We briefly explain the syntax and the semantics of the XMAS query language in the next paragraphs. It is straightforward to adjust XQForms to produce query statements of most proposed XML query languages [16].

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 6 is generated from the query form page of Figure 2. In line 14 the path `proximity_sensors.product` is matched against the XML source `proxSource` and variable `$P` receives the resulting bindings. In lines 15-16 the element `base_part_number` and the value of the `manufacturer` element are extracted from every binding of `$P`, to be included in the query result. In line 17, the variable `$SPEC` binds to the element `specs` so that the condition on `diameter`, shown in Figure 2, can be imposed in line 18 and the value of the `output_type` element retrieved in line 19.

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 4-8).

```
1   CONSTRUCT
2   <answer>
3      <product>
4         <manufacturer>$M</manufacturer>
5         $BPN {$BPN}
6         <specs>
7            <output_type>$OUT</output_type>
8         </specs>
9      </product> {$M, $OUT, $P}
10                order by{desc $M, desc $OUT}
11  </answer> {}
12
13  WHERE
14  proxSource proximity_sensors.product $P
15  AND $P base_part_number $BPN
16  AND $P manufacturer._ $M
17  AND $P specs $SPEC
18  AND body_type.cylindrical.diameter._ <= 25
19  AND $SPEC output_type._ $OUT
```

Figure 6 XMAS Query

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

- $V {$V1,…,$Vn}, as in line 5.
- element {$V1,…,$Vn}, where element is an element pattern, as in line 9.

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 {$BPN} in line 5 appearing within {$M, $OUT, $P} in line 9, 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 5, 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 9, one product element is created for each binding of the $M, $OUT and $P variables. Finally, in line 10, the sorting expression sorts the product elements by the variables $M (manufacturer) and $OUT (output_type), again as specified in Figure 2.

2.3.2 Navigation Interface

The interface that the data server we are currently using exports an API for navigation-driven evaluation of XMAS queries [10]. This API, in simple terms, has the ability to slice a query result in chunks. The parameters passed to the API are the size of the chunk and a navigation path pointing to an element in the **construct** clause we want to slice. XQForms exploits this API and provides to the end user control on the number of main elements returned from the data server and presented on the report page. In our running example, the navigation path points to the product element in the XML Schema in Figure 4. As Figure 2 illustrates, the end user can determine, on the query form page, the number of main elements that will be presented on the report page and then navigate into the chunks of query results using the “Next XX” and “Previous XX” links. XQForms also exploits the same API to provide the ability to continue from sorted elements on the report page. Again in Figure 2, the end user has the ability to continue from the sorted elements by clicking on the “Next” link displayed at the end of each column. Clicking on the “Next” link in the “Manufacturer” column, for example, will return the first ten product elements of the next “Manufacturer” value.

3 Annotation Scheme

The annotation scheme defines the syntax of the XQForm annotations and is the language that expresses the structure and the semantics of the generated query form and report pages. The annotation scheme itself is modeled by an XML Schema, which is shown in Figure 7, and it is divided into three main parts as the subelements of the `xqform` root element indicate.

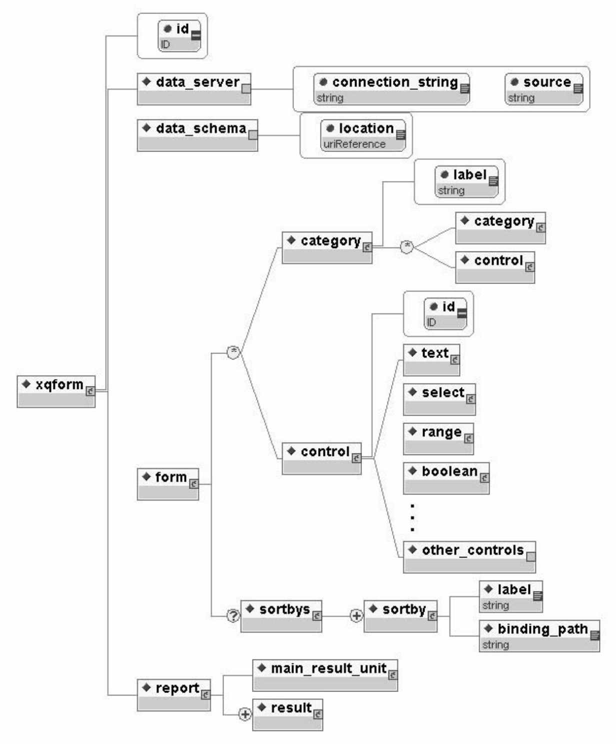


Figure 7 Annotation Scheme Overview

The first part captures the general attributes of an XQForm, such as the id of the XQForm, the data server that the queries will be sent to and the location of the XML Schema that will be used. The second part expresses the structure and the semantics of the query form page, as the `form` element indicates, and it is the part where the annotation scheme specifies the syntax of an easily extensible collection of query controls that have the ability to generate query fragments, such as selection and sorting conditions. The annotation scheme also provides the mechanism to bind these controls to data elements in an XML Schema, and allows the expression of potential dependencies between these controls. The `report` element is the third part of the annotation scheme and specifies which data elements will appear on the report page.

An XQForm annotation is an instantiation of this annotation scheme, and defines a particular XQForm. It is saved in an XML file, and is used by the compiler to produce the query form page, and by the run-time engine to produce a query upon the form submission, and a report page upon query execution.

In this section we elaborate on the query form part of an XQForm. Sections 3.3 presents in detail the specification of an XQForm report. Continuing with our running example, we use the XML Schema in Figure 4 to generate the XQForm that appears in Figure 2.

3.1 Query Controls

The building blocks of the query part of an XQForm are the query controls, referred to simply as 'controls' in the rest of the presentation. Each one of them has the ability to construct the appropriate selection condition or sorting condition that will potentially be included in the XML query when the user submits the form. Each control maps to one or more form elements on the query form page. The form elements interact with the end user and they pass their values to the query controls upon form submission. This architecture allows the specification of the query fragments that a control can produce to be decoupled from the way the end users interact and submit values via the generated query form and report pages, as in [9].

The current implementation of XQForms supports five types of controls, as Figure 7 indicates: `text`, for conditions involving string (contains, starts-with, etc.) and relational (>, <, ==, etc.) 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 on the report page. This set of controls is easily extensible depending on the query fragments we want to produce.

Let us walk through the specification of the `select` control, which is shown in Figure 8, as an example. The annotation scheme organizes the controls, and their corresponding form elements, 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 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 in the XML Schema in Figure 4 using a `select` control. The lifecycle of a query control, summarized in Figure 5, involves all four phases of the XQForms' lifecycle, shown in Figure 3.

1. In the design phase, the annotation author decides which data elements in the XML Schema she wants the end user to be able to pose conditions to via the query form page. 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, this control will construct a query fragment that poses a condition to a data element based on a predefined set of constants. Each constant is modeled with the `value` subelement of the `values` element, which has the following structure:

   a. the `label` subelement specifies a constant meaningful to the end user

   b. the `data_value` subelement specifies the constant as it appears in the data, if different than the one in `label`

   c. the `binding_path` subelement specifies the data element in the XML Schema the constant binds to, and

   d. the `condition` subelement specifies which predicate (such as ==, <=, etc.) will be used in the resulting query fragment.

   Note that the `binding_path` subelement is an absolute path of the form $e_1/\ldots/e_N$ (`e` is an element name) and uniquely identifies an element name e_N in the XML Schema. 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, the XML Schema and the template presentation library. The library contains one template for each query control, which the compiler instantiates in order to include on the query form page the form elements that map to the specific control. In our example, the form elements on the query form page that map to the `select` control are a text label and a single-selection drop-down list with the set of constants specified in the design phase. Note that this is one possible set of form elements that the end user can interact with in order to pose a condition to the `manufacturer` data element. Another way would be a set of radio buttons, or a multiple-selection drop-down list. In these cases the annotation/semantics of the query control remains the same and only the presentation template changes.

3. Instead of changing the presentation template of a control and recompile, the web designer has also the option to customize the form elements on the query form page during the customization phase.

4. When the end user submits the values of the form elements on the query form page, the XQForms run-time engine receives them and passes them to the corresponding query controls. Next, each control constructs and contributes its own query fragment, based on the submitted values, to the XMAS query that the run-time engine will send to the data server. Figure 5 shows the query fragment that our example `select` control contributes to the XMAS query when the "Turck" value is submitted via the form element on the query form. As can be seen in Figure 5 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 fragment to impose the condition on the submitted value, 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 form element and is rendered as a hyperlink to an explanatory text. The `additional` element specifies if the corresponding form elements of the control are initially hidden, as is the "Part Number" form element, 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 and mapped to form elements that give the ability to the end user to choose among several units in which she can express the submitted values[1], like the "Dimensions" form element in Figure 2. The rest of the elements are specific to the `select` control.

The `values` element models the set of constants of the `select` control and 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 XML Schema elements via different binding paths and can use different predicates to apply a condition to the XML data. The `dynamic` element is used when the set of constants appearing in the form elements on the query form page 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 constants, in which case a disjunction of the conditions generated from each value will be included in the XMAS query. The

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

`dependencies` element is used from the annotation author to express dependencies among the query controls and is further discussed in Section 3.2.

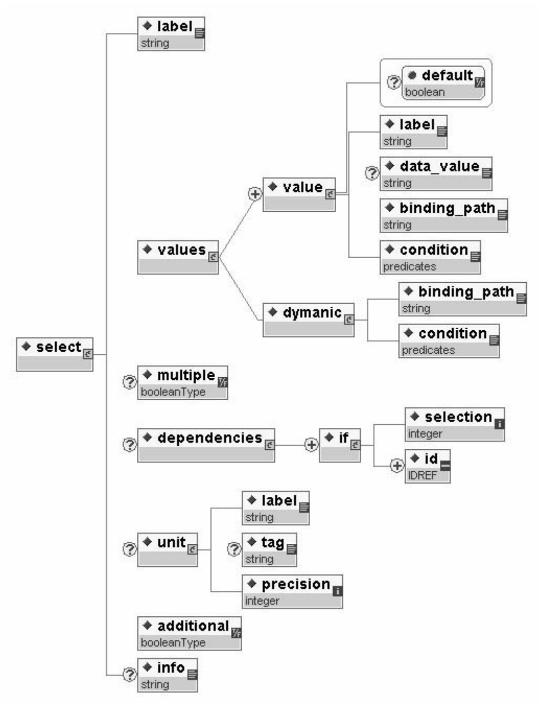


Figure 8 `select` Query Control Specification

The part of an example annotation referring to the `select` control that maps to the "Manufacturer" form element in Figure 2 is shown below. Note the correspondence, which is illustrated in Figure 5, between the properties of the control, its form elements on the query form page and the condition that it contributes to the XMAS query.

```
<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` Query Control Annotation

3.2 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` subelement we want a different set of query controls to contribute in querying the corresponding dimensions, and we also want the set of corresponding form elements for querying these dimensions to be shown on the query form page. The annotation author uses a `select` control to query the body_type element in the sensors' XML Schema, and defines a set of two constants; “Cylindrical” that binds to the cylindrical subelement of the body_type element; and “Rectangular” that binds to the rectangular subelement. The author also binds controls to the dimensions of either body type elements. She wants the corresponding form elements to appear whenever one or the other value is selected. So when the end user interacts with the “Body Type” form element and selects the “Cylindrical” value, the form elements that map to the query controls for diameter and depth data elements should appear on the page, and when she selects the “Rectangular” values, the form elements that map to the query controls for height, width and depth data elements should appear.

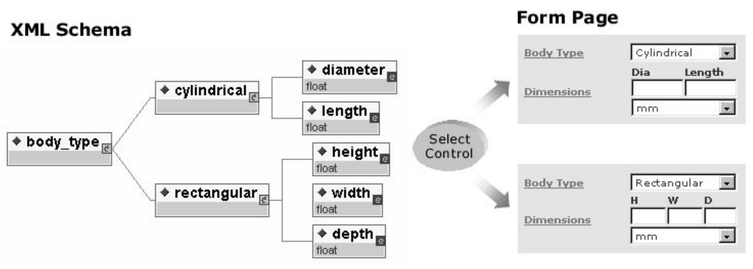


Figure 10 Expressing Dependencies

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 constant defined in the select control. The selection element is set to one of the label elements and the id identifies a query control[2]. The action carried out when a dependency is detected is defined in the semantics of the query controls and in their presentation templates, and thus can easily customized or extended. 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.

3.3 Reports

XQForms produces the report pages in two steps. First, the data elements that will be presented on the page have to be retrieved by including them in the XMAS query that is sent to the data server, and second, an XSL script has to be generated and applied to the returned data in order to render them into the presentation language on the report page. XQForms automates both of these steps based on the annotation the author provides, where she declaratively specifies the data that she wants to appear on the report page. The query language syntax and the complexity of the XSL script for rendering the data are completely hidden, while a mechanism for easy customization of the report page is provided. These facts facilitate the rapid development of XQForms and also enable the support of advanced web services through dynamic report structuring, e.g., personalized report settings.

More specifically, the annotation author creates an annotation for the report page by instantiating the report element, which includes both elements that contribute to the query construction and elements that contribute to the XSL script construction. Its structure is shown in detail in Figure 11 and part of the annotation referring to the report page in Figure 2 is shown in Figure 12. The main_result_unit element specifies the element we use to navigate into the results on the report page. In our example, the product element in the XML Schema in Figure 4 is the one we use to navigate into the results on the report page in Figure 2.

[2] Note in Figure 7 that every control has an id element.

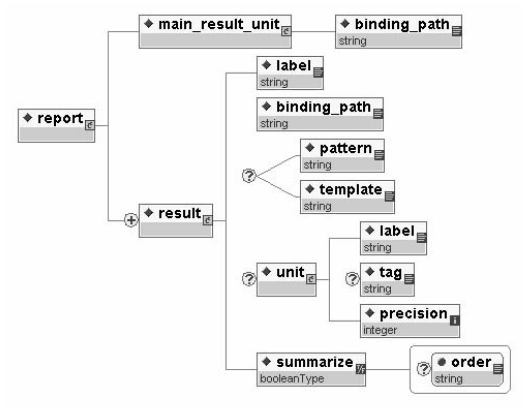


Figure 11 Report Specification

Another property of the main result unit element is that all the data elements that will be presented on the report page must be its subelements, because the constructed query follows the structure of the XML Schema under the main result unit element and includes only the subelements specified in the annotation. These subelements are specified from the `result` elements in Figure 11, and specifically from their `binding_path` subelement, which is a path relative to the main result unit. In Figure 12, the example annotation specifies that elements such as `manufacturer` and `base_part_number` will be presented on the report page as Figure 2 shows. All these elements are subelements of `product` element. The summarize element also contributes to the query construction by specifying if the possible distinct values of a data element should be extracted, and if so in what order should they be presented on the report page (`order` attribute). The effect of this annotation can be seen in Figure 2 for the `output_type` element.

The other subelements of the `result` element are used to generate the XSL script that will render the query results. The `label` element and the `unit` element, which has the same structure with the one in the query controls' specification, are used in the header of the values of the corresponding data element. In Figure 12 the unit for the `min` element of `supply_voltage` is "V" (Volts) and the precision for its numeric value is 0 (no decimal digits), as shown in Figure 2.

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

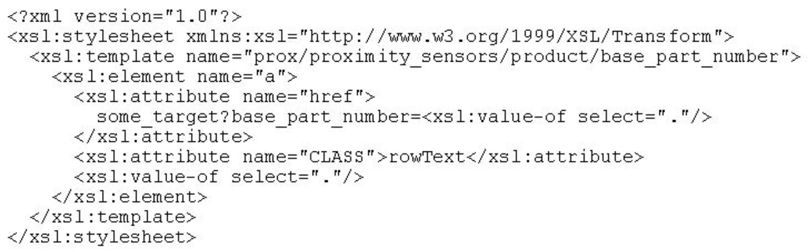
Figure 12 Example Report Annotation

The XSL script that is automatically constructed presents the data on the report page by also following the structure of the XML Schema. The paradigm of nested tables is heavily used for the visual translation of nested and/or repeatable elements. Note, for example, the correspondence between the structure of the `supply_voltage` element in Figure 4 and the visual effect that it has by including it in the annotation for the report page in Figure 2. 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.

The idea of using templates is applied in the presentation of the report page, where templates are used for the presentation of simple elements (leaves in the data XML Schema) and for the presentation of complex elements (elements that have subelements). These templates can be generic, i.e., based on the data type, or specific, e.g., for a particular element in an XML Schema. In both cases the web designer edits only a single template and not a long, complicated XSL script. The process of putting these templates together in order to constructs the XSL script that renders the query results on the report page is done by the XQForms run-time engine and is discussed further in the next section.

XQForms allows the web designer to provide a separate generic XSL template for each data type, which is applied by default to simple data elements included in the report, and a generic one for complex elements. Currently, the simple data types of the XML Schema specification that are supported are `integer`, `float`, `boolean`, and `string`. The web designer can customize the presentation of specific data elements using either the `pattern` or the `template` subelements of the `report` element in 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 the example annotation in Figure 12, 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. For example, on the report page in Figure 2, 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 absolute path in the XML Schema that leads to the element that it will render. So the `name` attribute of the XSL `template` element for the base part number is the path `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 figure 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>
```

Figure 13 Customizing Presentation with an External XSL Template

3.4 Report System Implementation

The XQForms component that provides the automated query and XSL script construction is the run-time engine and the overall process is illustrated in Figure 14. When the end user submits the query form, an XMAS query is constructed from three different parameters. The parameters the end user submitted, which are used form the query controls, the XML Schema and the XQForm annotation. The run-time engine generates the **construct** clause of the XMAS query starting from the main result unit element specified in the annotation. This is always the first element constructed in the **construct** clause and is always grouped under an `answer` element as shown in Figure 6. Then, the subelements of the main result unit element that are specified from the `report` elements of the annotation are included in the **construct** clause of the query by following the structure of the XML Schema. Their `binding_path` element is used to extract these element in the **where** clause of the query and reconstruct them under the main result unit in the **construct** clause, like the `output_type` element in Figure 6.

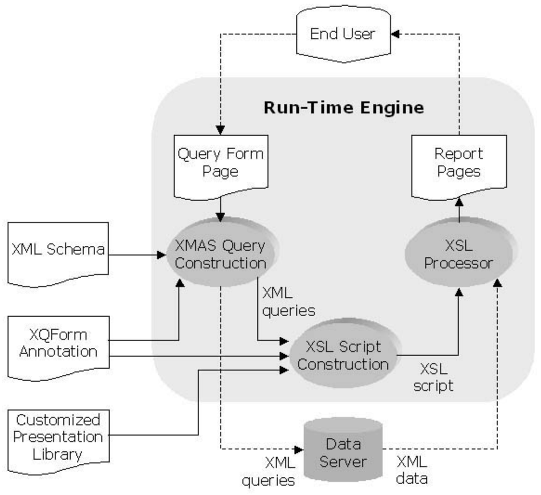


Figure 14 Report System Architecture

This query is sent to the data server and at the same time the run-time engine starts constructing the XSL script that will render the XML data that satisfy the query. The run-time engine, in order to *automatically* generate this XSL script, called *final*, takes three inputs: the XML query that was generated, the XQForm annotation, and the presentation library, which contains the XSL templates for simple and complex elements. The **construct** clause of the XMAS query reveals the structure of the XML data that the data server will respond with, which is contained in the XML Schema. The engine applies a set of system XSL scripts, called *metaXSL* scripts, which follow the structure of the **construct** clause and construct the visual structure of the report page accordingly. These metaXSL scripts also attach to the simple and complex elements the XSL templates that will render the data that the server will provide into the presentation language. The metaXSL scripts are also provided from the presentation library. The annotation provides the patterns and the customized templates that the annotations' author and/or the web designer have specified.

In our running example, the final XSL script that renders the query result on the report page shown in Figure 2 is constructed from the metaXSL scripts as Figure 15 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. It also contains the declarations of all the global parameters that are passed to the final XSL script. This template is *static* in the sense that it is executed directly against the XML data and is the same for every XMAS query an XQForm generates. The XSL templates created from all the other metaXSL scripts are appended to this *stylesheet*. The templates listed next are called metaXSL scripts in the sense that their output is different for different queries generated from a single XQForm. They take as input an XML version of the **construct** clause of the XMAS query and they output an XSL scripts that the run-time engine composes to form the final XSL script.

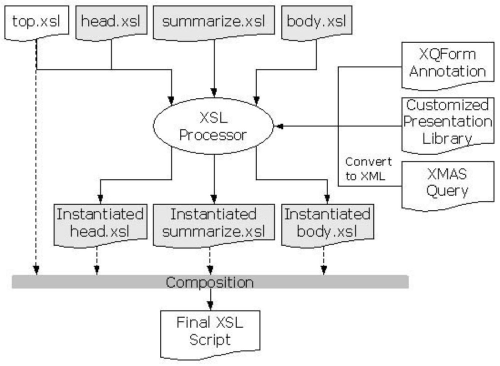


Figure 15 Automatic Report Formatting

- 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 in the annotation, and by following the structure of the XMAS **construct** clause.

- 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 simple and complex elements in order to reveal this structure visually. It also checks if a pattern or a custom template has been specified for the simple elements. If this not the case, a default XSL template based on the simple element's data type is attached to it.

Note that the automatic XSL script creation described above can be carried out in two different phases of the XQForms' lifecycle. The script can be created at run-time, as described so far, which does not increase the query response time, because the creation of the final XSL script starts 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. 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 same, static customized script to render the reports for all queries the XQForm generates.

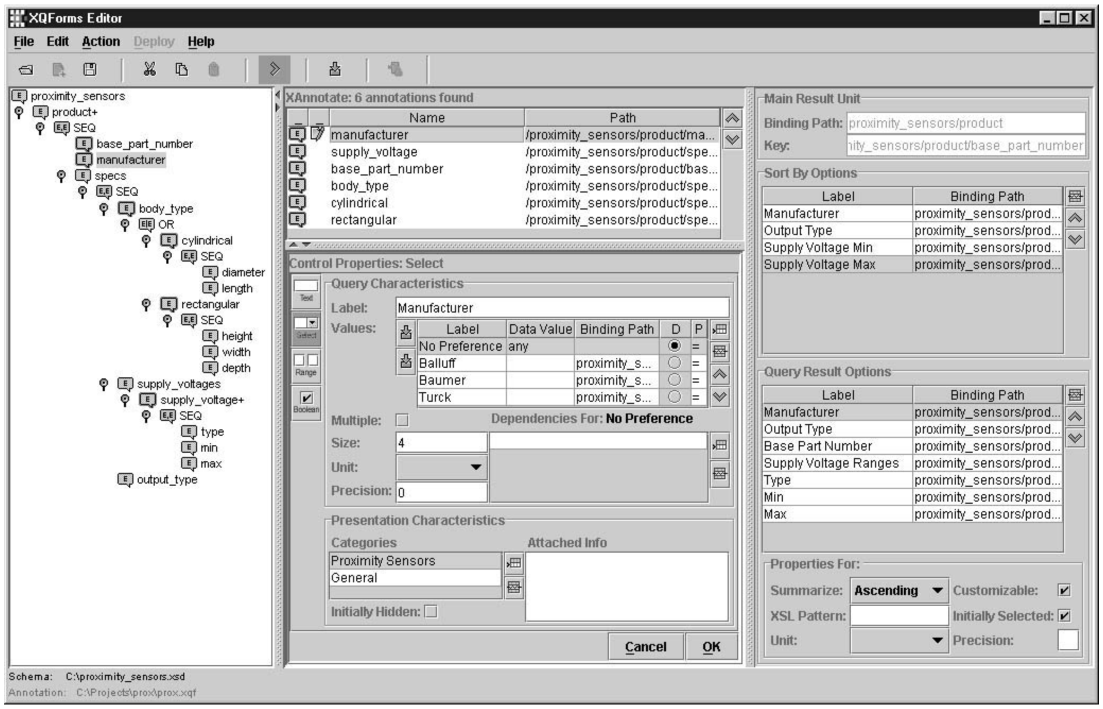
Finally, the styles applied to all presentation templates of XQForms are controlled from a central CSS file. The web designer can easily determine the look and feel of entire XQForms by changing the styles used across the presentation templates for the query controls and/or the styles used in the XSL templates for simple and complex elements presented on the report page.

Note that the report system can work with any XMAS query if an XML Schema is provided for the query results and not only with the queries that an XQForm can produce.

4 XQForms Editor

The XQForms Editor is a graphical user interface (GUI) that the annotations' author uses to create new XQForm annotations, edit existing ones, as well as deploy them to an application server. The basic editor screen is composed of the following three panes, as illustrated in Figure 16.

- The *schema* pane on the left displays the XML Schema in a simple format. The annotations' author can select any of the elements and bind it to a query control, include in the sort-by list or present it on the report page.
- The *query control annotation* pane on the center displays the query controls included in the form and provides an editor for the properties of each one of them, as they appear in the annotation scheme in Figure 7. These editors are also used to specify the dependencies between the query controls.
- In the *report annotation* pane, positioned on the lower right corner, the annotations' author simply drags and drops the data elements from the schema that she wants to be presented on the report page and then specify the summarization and pattern parameters as discussed in Section 3.3. On the right side, the main result unit and the sort-by list are also specified.



```
proximity_sensors
  product+
    SEQ
      base_part_number
      manufacturer
    specs
      SEQ
        body_type
          OR
            cylindrical
              SEQ
                diameter
                length
            rectangular
              SEQ
                height
                width
                depth
        supply_voltages
          supply_voltage+
            SEQ
              type
              min
              max
        output_type
```

XAnnotate: 6 annotations found

Name	Path
manufacturer	/proximity_sensors/product/ma...
supply_voltage	/proximity_sensors/product/spe...
base_part_number	/proximity_sensors/product/bas...
body_type	/proximity_sensors/product/spe...
cylindrical	/proximity_sensors/product/spe...
rectangular	/proximity_sensors/product/spe...

Control Properties: Select

Query Characteristics

Label: Manufacturer

Values:

Label	Data Value	Binding Path	D	P
No Preference	any		●	=
Balluff		proximity_s...	○	=
Baumer		proximity_s...	○	=
Turck		proximity_s...	○	=

Multiple: ☐ Dependencies For: **No Preference**

Size: 4

Unit:

Precision: 0

Presentation Characteristics

Categories: Proximity Sensors, General

Attached Info

Initially Hidden: ☐

Cancel OK

Main Result Unit

Binding Path: proximity_sensors/product

Key: nity_sensors/product/base_part_number

Sort By Options

Label	Binding Path
Manufacturer	proximity_sensors/prod...
Output Type	proximity_sensors/prod...
Supply Voltage Min	proximity_sensors/prod...
Supply Voltage Max	proximity_sensors/prod...

Query Result Options

Label	Binding Path
Manufacturer	proximity_sensors/prod...
Output Type	proximity_sensors/prod...
Base Part Number	proximity_sensors/prod...
Supply Voltage Ranges	proximity_sensors/prod...
Type	proximity_sensors/prod...
Min	proximity_sensors/prod...
Max	proximity_sensors/prod...

Properties For:

Summarize: **Ascending** Customizable: ✓

XSL Pattern: Initially Selected: ✓

Unit: Precision:

Schema: C:\proximity_sensors.xsd

Annotation: C:\Projects\prox\prox.xqf

Figure 16 XQForms Editor

5 Related Work

The XML Forms Language [9] 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 [17], 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 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 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 [6] 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, 11]. 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 [14], where advanced techniques and data structures for summarizing are presented.

6 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 controls* have the ability to generate query fragments based on the annotations and the values the end user submits.
- The *XQForms Editor* assists the annotations' author in building the XQForm annotations that specify the form and report pages to be generated. The syntax of the annotations is defined from the *annotation scheme*.
- The *compiler* inputs XQForm annotations, XML Schemas and template presentation libraries and produces the query form pages.
- The *run-time engine* performs the automatic query construction and report formatting.

XQForms provide templates and styles for a default rendition of query 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 form and report pages 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 [6] 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://www.db.ucsd.edu/xqforms/.

7 References

[1] S. Banerjee et al.: *Oracle8i - The XML Enabled Data Management System*, in proceedings of the 16th International Conference on Data Engineering (ICDE), 2000, pp. 561-568.

[2] M. Carey, L. Haas, V. Maganty, J. Williams: *PESTO: An Integrated Query/Browser for Object Databases*, in proceedings of the 22nd International Conference on Very Large Databases (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, 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] J. M. Cheng, J. Xu: *XML and DB2*, in proceedings of the 16th International Conference on Data Engineering (ICDE), 2000, pp. 569-573.

[6] M. Fernandez, D. Suciu and I. Tatarinov: *Declarative Specification of Data-intensive Web sites*, in proceedings of the 2nd Conference on Domain-Specific Languages (DSL), 1999.

[7] D. Florescu, A. Deutsch, A. Levy, D. Suciu, and M. Fernandez: *A Query Language for {XML}*, in proceedings of WWW9, 1999.

[8] R. Goldman, J. McHugh, and J. Widom: *From Semistructured Data to XML: Migrating the Lore Data Model and Query Language*, in proceedings of the 2nd International Workshop on the Web and Databases (WebDB), 1999.

[9] A. Kristensen: *Formsheets and the XML Forms Language*, in proceedings of WWW9, 1999.

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

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

[12] M. Petropoulos, V. Vassalos, Y. Papakonstantinou: *XML Query Forms (XQForms): Declarative Specification of XML Query Interfaces*, in proceedings of WWW10, 2001.

[13] H. Schöning, J. Wäsch: *Tamino - An Internet Database System*, in Extending Database Technology (EDBT), 2000, pp. 383-387.

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

[15] S. Adler et al.: *Extensible Stylesheet Language (XSL) Version 1.0*, W3C Recommendation 15 October 2001.
http://www.w3.org/TR/xsl/

[16] D. Chamberlin et al.: *XML Query Requirements*, W3C Working Draft 15 February 2001.
http://www.w3.org/TR/xmlquery-req

[17] M. Dubinko et al.: *XForms Requirements*, W3C Working Draft 04 April 2001.
http://www.w3.org/TR/xhtml-forms-req

[18] D. Fallside: *XML Schema Part 0: Primer*, W3C Recommendation 2 May 2001.
http://www.w3.org/TR/xmlschema-0/

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

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

Vitae

Michalis Petropoulos received a Diploma in Electronic and Computer Engineering from the Technical University of Crete in Chania, Greece, in 1998, and a M.Sc. in Computer Science from University of California, San Diego, in 2000, where he is currently a Ph.D. student and is expected to graduate in Spring 2003. His research interests are in the field of web and databases and include mediator systems, web services and web application modeling. Prior work includes design of front-end generation tools, XML wrappers for relational databases, news on demand systems and multimedia applications design and development.

Yannis Papakonstantinou serves on the Faculty of Computer Science and Engineering at the University of California, San Diego, since 1996. His research is in the intersection of database and Internet technologies. Yannis has published over forty research articles in scientific conferences and journals, given tutorials at major conferences, and served on journal editorial boards and program committees for numerous international conferences and symposiums. He is the co-chair of WebDB 2002 and the general chair of SIGMOD 2003. In 1998, Yannis received the NSF CAREER award for his work on integrating heterogeneous data. In 2000 Yannis founded Enosys Markets, Inc., which provides software for XML-based querying and integration of distributed sources. Yannis holds a Diploma of Electrical Engineering from the National Technical University of Athens and MS and Ph.D. in Computer Science from Stanford University (1997).

Vasilis Vassalos serves on the Faculty of the Information Systems Department at the Stern School of Business of the New York University since 1999. Vasilis conducts research in databases, Web-based information systems and middleware. He was a leading member of the pioneering TSIMMIS project on integration and processing of XML information. Vasilis has published over a dozen research articles and given tutorials on the topics of integration systems, languages, and algorithms, and their applications in business information systems. In 2000 Vasilis co-founded Enosys Markets, Inc., which provides the state of the art software for XQuery-based integration of distributed sources. Vasilis received his Diploma in Electrical Engineering (highest honors) from the National Technical University of Athens and his M.S. and Ph.D. in Computer Science from Stanford University.