

# QURSED: Querying and Reporting Semistructured Data

Yannis Papakonstantinou  
Computer Science and Eng. Dept.  
University of California, San Diego

yannis@cs.ucsd.edu

Michalis Petropoulos  
Computer Science and Eng. Dept.  
University of California, San Diego

mpetropo@cs.ucsd.edu

Vasilis Vassalos  
Information Systems Dept.  
New York University

vassalos@stern.nyu.edu

## ABSTRACT

QURSED enables the development of web-based query forms and reports (QFRs) that query and report semistructured XML data, i.e., data that are characterized by nesting, irregularities and structural variance. The query aspects of a QFR are captured by its query set specification, which formally encodes multiple parameterized condition fragments and can describe large numbers of queries. The run-time component of QURSED produces XQuery-compliant queries by synthesizing fragments from the query set specification that have been activated during the interaction of the end-user with the QFR. The design-time component of QURSED, called QURSED Editor, semi-automates the development of the query set specification and its association with the visual components of the QFR by translating visual actions into appropriate query set specifications. We describe QURSED and illustrate how it accommodates the intricacies that the semistructured nature of the underlying database introduces. We specifically focus on the formal model of the query set specification, its generation via the QURSED Editor and its coupling with the visual aspects of the web-based form and report.

## 1. INTRODUCTION

XML provides a powerful and simple way to represent and exchange data, largely due to its self-describing nature. Its advantages are especially strong in the case of semistructured data, i.e., data whose structure is not rigid and is characterized by nesting, optional fields, and high variability of the structure. An example is a catalog for complicated products such as sensors: they are often nested into manufacturer categories and each product of a sensor manufacturer comes with its own variations. For example, some sensors are rectangular and have height and width, and others are cylindrical and have diameter and barrel style. Some sensors have one or more protection ratings, while others have none. The relational data model is cumbersome in modeling such semistructured data because of its rigid tabular structure.

The database community perceived the relational model's limitations early on and responded with labeled graph data models [1] first and XML more recently. XML query languages (with most notable the emerging XQuery standard [26]), XML

databases [22] and mediators [7,9,16,30] have been researched and developed. They materialize the in-principle advantages of XML in representing and querying semistructured data. Indeed, mediators allow one to export XML views of data found in relational databases [9,31], HTML pages, and other information sources, and to obtain XML's advantages even when one starts with non-XML legacy data. QURSED automates the construction of web-based query forms and reports for querying semistructured XML data.

Web-based query forms and reports are an important aspect of real-world database systems [3,23], albeit semi-neglected by the database research community. They allow millions of web users to selectively view the information of underlying sources. A number of tools [31,32,35] facilitate the development of web-based query forms and reports that access relational databases. However, these tools are tied to the relational model, which limits the resulting user experience and impedes the developer in his efforts to quickly and cleanly produce web-based query forms and reports. QURSED is, to the best of our knowledge, the first web-based query forms and reports generator with focus on semistructured XML data.

QURSED produces query form and report pages that are called *QFRs*. A QFR is associated with a *query set specification*, which typically describes a large set of parameterized queries that may be instantiated and emitted from the query form page to the XML query processor in the course of its interaction with the end-user. The emitted queries are expressed in XQuery and the query results are expressed directly in HTML, for performance reasons.

### 1.1 System Overview and Architecture

We discuss next the QURSED system architecture (see Figure 1), the process and the actions involved in producing a QFR and the process by which a QFR interacts with the end-user, emits a query and displays the result. We also introduce terms used in the rest of the paper. QURSED consists of the *QURSED Editor*, which is the design-time component, the *QURSED Compiler*, and the *QURSED Run Time Engine*.

The Editor inputs the XML Schema that describes the structure of the XML data to be queried and an *HTML query form page* that provides the visual (HTML) part of the form page, including the HTML form controls [29], such as `select` ("drop-down menus") and `text` ("fill-in-the-box") input controls, that the end-user will be interacting with. It may also input:

1. An optional *HTML template report page* that provides the visual pattern of the report page. In particular, it depicts the nested tables and other components of the page. It is just a template, since we may not know in advance how many rows appear in each table. The query form and template report

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SIGMOD '2002, June 4-6, Madison, Wisconsin, USA.  
Copyright 2002 ACM 1-58113-497-5/02/06...\$5.00.

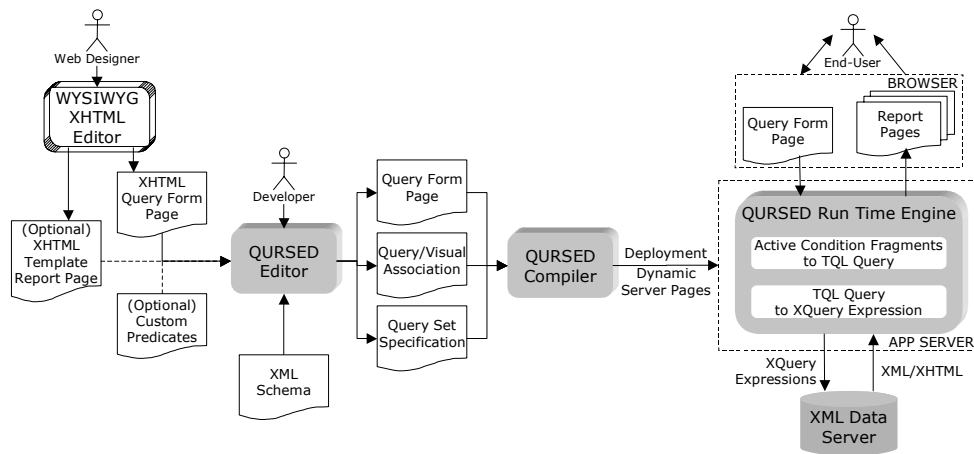


Figure 1. QURSED System Architecture

pages are typically developed with an external “What You See Is What You Get” (WYSIWYG) editor, such as Macromedia HomeSite. If a template report page is not provided, the developer can build one using the Editor.

2. An optional set of functions and predicates (and their signatures) understood by the target XML query processor and a set of functions and predicates understood by the runtime engine of QURSED.

Then the Editor displays the XML Schema and the HTML pages to the developer, who uses them to visually build the *query set specification* of the QFR and the *query/visual association*. The specification focuses on the query aspects of the QFR and describes the set of queries that the form may emit. The query description is based on the formalism of the *Tree Query Language (TQL)* described in Section 4. The specification’s key components are the parameterized *condition fragments* and the *result tree*. Each condition fragment stands for a set of conditions (typically navigations and selections, joins are also possible) that contain *parameters*. The query/visual association indicates how each parameter is associated with corresponding *HTML form controls* [29] of the query form page. The form controls that are associated with the parameters contained in a condition fragment constitute its *visual fragment*. The result tree specifies how the source data instantiate and populate the HTML template report page.

The *QURSED Compiler* takes as input the output of the Editor and produces *dynamic server pages*, in the form of Java Server Pages, which control the interaction with the end-user. The dynamic server pages, the query set specification and the query/visual association are inputs to the *QURSED Run-time Engine*. In particular, the dynamic server pages allow the end-user to enter her input parameters on the query form page and handle the navigation on the report page. The engine, based on the query set specification and the query/visual association, generates an XQuery expression when the end-user clicks “Execute Query”, which is sent to the XML Data Server and its HTML result is displayed on the report page. Query generation proceeds in two steps: The set of *active condition fragments*, i.e., of fragments whose parameters (if any) have been given values, is combined into a TQL query. Then the TQL query is translated into an XQuery expression that directly produces the HTML report page.

The rest of the paper is organized as follows. The related work and the list of contributions of QURSED are presented in Section 2. In Section 3 the running example is introduced and the end-

user experience is described. Section 4 describes TQL, and Section 5 presents the query set specification formalism. Section 6 presents the Editor.

## 2. RELATED WORK & NOVEL CONTRIBUTIONS OF QURSED

The QURSED system relates to three wide classes of systems, coming from both academia and industry.

*Web-based Form and Report Generators*, such as Macromedia DreamWeaver Ultradev and ColdFusion, and Microsoft Visual Interdev. All of the above enable the development of web-based applications that create form and report pages that access relational databases. QURSED is classified in the same category, except for its focus on semistructured data.

In all of the above generators the developer uses a set of wizards to visually explore the tables and views defined in a relational database schema and selects the one(s) she wants to query using a query form page. By dragging ‘n’ dropping the attributes of the desired table to HTML form controls [29] on the page, she creates conditions that, during run-time, restrict the attribute values based on the end-user’s input. The developer can also select the tables or views to present on a report page, and by dragging ‘n’ dropping the desired attributes to HTML elements on the page, e.g., table cells, the corresponding attribute values will be shown as the element’s content. The developer also specifies the HTML region that will be repeated for each record found in the table, e.g., one table row per record. These actions are translated to scripting code or a set of custom HTML tags, such as the JSP library of tags, that these products support and generate. The custom tags incorporate common database and programming languages functionality and one may think of them as a way of folding a programming/scripting language into HTML.

Those tools are excellent when flat uniform relational tables need to be displayed. The visual query formulation paradigm offered to the developer allows the expression of projections, sort-bys, and simple conditions. However, the development of form and report pages that query and display semistructured data requires substantial programming effort.

*Visual Querying Interfaces*, such as QBE [24] and Microsoft’s Query Builder (part of Visual InterDev), which target the querying of relational databases, and EquiX [8], BBQ [18], VQBD [6], the Lorel’s DataGuide-driven GUI [14], and PESTO [4], which target the querying of XML or object-oriented

databases. These are applications that allow the exploration of the schema and/or content of the underlying database and the formulation of queries.

Unlike the form and report generators, which produce web front-ends for the “general public”, visual querying interfaces present the schema of the underlying database to experienced users, who are often developers building a query, help them formulate queries visually, and display the result in a default fashion. The user has to, at the very least, understand what is the meaning of “schema” and what is the model of the underlying object structure, in order to be able to formulate a query. For example, the QBE user has to understand what a relational schema is and the user of Lorel’s DataGuide GUI has to understand that the tree-like structure displayed is the structure of the underlying XML objects. These systems have heavily influenced the design of the Editor because they provide an excellent visual paradigm for the formulation of fairly complex queries.

*Data-Intensive Web Site Generators*, such as Autoweb [12], Araneus [2] and Strudel [10]. These are excellent examples of the ongoing research on how to design and develop web sites from database content. An extensive discussion on this class of systems can be found in [11]. All of them offer a data model, a navigation model and a presentation model. They provide important lessons on how to decouple the query aspects of web development from the presentation ones. (Decoupling the query from the presentation aspects is an area where commercial web-based form and report generators suffer.) Strudel is based on labeled directed graphs for both data and web site modeling, which is close to the XML model of QURSED. The query language of Strudel, called StruQL, is used to define the way data are integrated from multiple sources (data graph), the pages that make up the web site, and the way they are linked (site graph). Each node of the site graph corresponds to exactly one query, which is manually constructed. Query forms are defined on the edges of the site graph by specifying a set of free variables in the query, which are instantiated when the page is requested, producing the end node of the edge. Similarly, Autoweb and Araneus perceive query forms as a single query, in the sense that the number of conditions and the output structure are fixed. In Strudel, if conditions need to be added or the output structure to change, a new query has to be constructed and a new node added to the site graph. In other words, every possible query and output structure has to be written and added to the site graph. QURSED is complementary to these systems, as it addresses the problem of encoding a large number of queries in a single QFR and grouping and representing different reports by a single site graph node.

This paper is a continuation of the work in [19], where we described a software architecture that allows an extensible set of HTML input controls to be associated with element definitions of an XML schema via an annotation on the XML Schema. The paper did not describe how the system encodes or composes queries and results of queries based on user actions.

## 2.1 Contributions

*Forms and Reports for Semistructured Data.* QURSED generates form and report pages that target the needs of interacting with and presenting semistructured data. Multiple features contribute in this direction:

1. QURSED generates queries that handle the structural variance and irregularities of the source data by employing appropriate forms of disjunction. For example, consider a sensor query form that allows the user to check whether the sensor fits within an envelope with length  $X$  and width  $Y$ , where  $X$  and  $Y$  are user-provided parameters. The corresponding query has to consider whether the sensor is cylindrical or rectangular, since  $X$  and  $Y$  have to be compared against a different set of dimension attributes in each case.
2. On the report side, data can be automatically nested according to the nesting proposed by the source schema or can be made to fit HTML tables that have variance in their structure and different nesting patterns. Structural variance on the report page is tackled by producing heterogeneous rows in the resulting HTML tables.

*Loose Coupling of Query and Visual Aspects:* QURSED separates the logical aspects of query forms and reports generation from the presentation ones, hence making it easier to develop and maintain the resulting form and report pages. The visual component of the forms can be prepared with any HTML editor. Then the developer can focus on the logical aspects of the forms and reports: Which are the condition fragments? How should the report be nested? The coupling between the logical and the visual part is loose, simple, and easy to build: The query parameters are associated with HTML form controls, the condition fragments are associated with sets of HTML form controls, and the *grouped* elements (see Section 4) of the result tree are associated with the nested tables of the report.

*Powerful and Succinct Query Set Specification:* We provide formal syntax and semantics for the QFR query set specifications, which describe large numbers of meaningful semistructured queries. The specifications primarily consist of parameterized condition fragments, whose combinations lead to large numbers of parameterized queries.

The query set specifications are using the Tree Query Language (TQL), which is a calculus-based language. TQL is designed to handle the structural variance and missing fields of semistructured data. Nevertheless, TQL’s purpose is not to be yet another general-purpose semistructured query language. Its design goals are to:

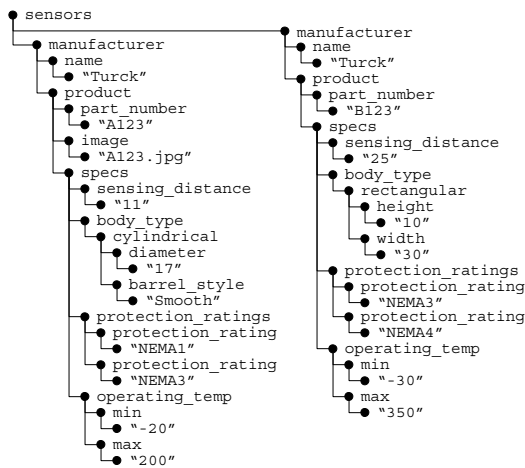
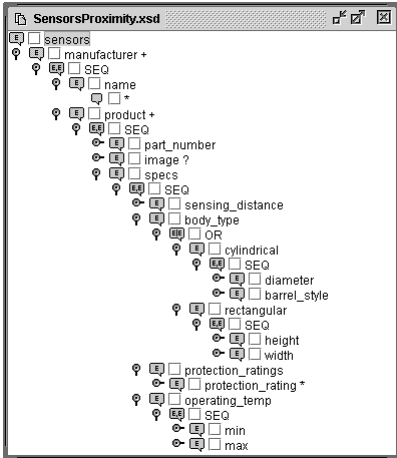
1. Facilitate the definition of query set specifications and, in particular, of condition fragments.
2. Provide a tree-based query model that captures easily the schema-driven generation of query conditions by the forms component of the Editor and also maps well to the model of nested tables used by the reports.

## 3. EXAMPLE

This section describes an example XML Schema and the data model of QURSED, and introduces as the running example a QURSED-generated web interface. It concludes by describing the end-user experience with that interface.

### 3.1 Example XML Schema and Data Model

Consider the example XML Schema of Figure 2, which models proximity sensor products, and a sample data set that conforms to it. This is the form in which the Editor displays the schema to the developer. Indicated are the optional (? suffix) and repeatable (\* and + suffixes) elements and the choices and sequences (OR and



**Figure 2. Example XML Schema and Conforming Data Set**

SEQ elements) of elements. Also, the elements of primitive type [25] are indicated with a wildcard (\* label) as element name (leaf nodes only.) Like many XML Schemas, it has nesting and many “irregular” structures such as choice groups and optional elements [28]. The top element is called `sensors` and contains one

manufacturer element for each manufacturer whose sensors are featured in the data set. Each manufacturer contains a name and a list of product subelements, whose direct subelements model the basic information of each sensor. The technical specification of each sensor is modeled by the `specs` element, whose content is quite irregular. For example, the body type may be `rectangular`, in which case the sensor has height and width dimensions, or `cylindrical`, in which case it has diameter dimension and `barrel_style`, and each sensor can have zero, one or more `protection_rating` elements.

XML Schemas, like the one in Figure 2, have the expressive power to describe irregularities and nesting, and they can be visualized in an intuitive manner. The developer can carry out a set of tasks, such as formulate queries and transform data, on the schema structure the underlying database system uses, without the need of another abstraction — as is the case with relational databases.

We model XML as labeled ordered tree objects (*lotos*). Each internal node of the labeled ordered tree represents an XML element and is labeled with the element’s tag name. The list of children of a node represents the sequence of elements that make up the content of the element. A leaf node holds the string value of its parent node.

### 3.2 Example QFR and End-User Experience

Using QURSED, a developer can easily generate a web interface like the one shown in Figure 3 that queries and reports proximity sensor products. This interface will be the running example and will illustrate the basic points of the functionality and the experience that QURSED delivers to the end-user of the interface.

The browser window displays a query form page and a report page. On the query form page form controls are displayed for the end-user to select or enter desired values of sensors’ attributes. The state of the query form page of Figure 3 has been produced by the following end-user actions:

- Placed the equality condition “NEMA3” on “Protection Rating 1”.
- Left the preset option “No preference” on “Body Type” and placed the conditions on “Dimension X” being less than 20 “mm” and “Dimension Y” less than 40 “mm”. These two

Image	Manufacturer	Part Number	Sensing Distance mm	Body Type
	Turck	BC 3-M12-AN6X	6.0	Cylindrical
				Diameter mm Barrel Style
				15 Smooth
	Turck	BC 3-M12-AP6X	6.0	Cylindrical
				Diameter mm Barrel Style
				19 Smooth
	Turck	BC 5-Q08-AN6X2	7.0	Rectangular
				Height mm Width mm
				14 9
	Turck	BC 5-Q08-AP6X2	7.5	Rectangular
				Height mm Width mm
				10 35
	Turck	BC 5-S18-AN4X	10.0	Rectangular
				Height mm Width mm
				15 10

**Figure 3. Example QFR Interface**

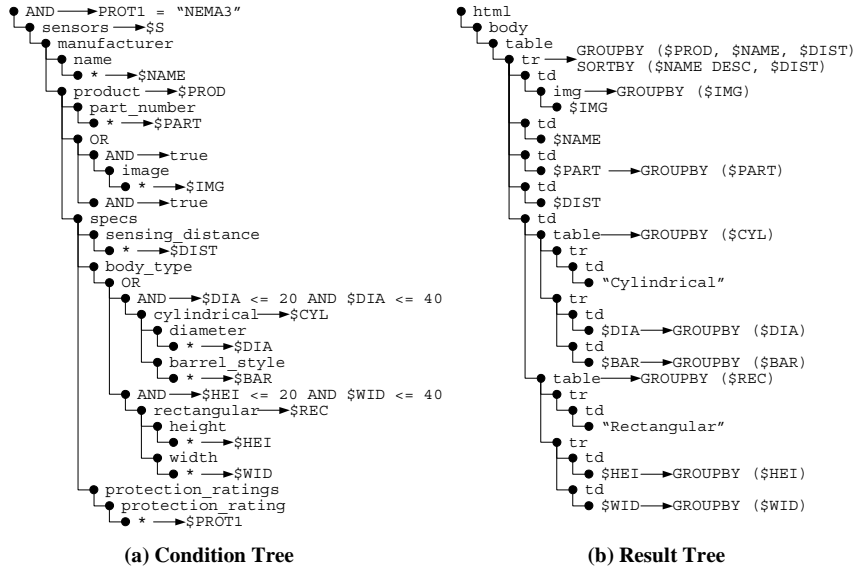


Figure 4. TQL Query Corresponding to Figure 3

dimensions define an envelope in which the end-user wants the sensors to fit, without specifying a particular body type.

After the end-user submits the form, she receives the report of Figure 3. The results depict the information of product elements: the developer had decided earlier that products should be returned. QURSED organizes the presentation of the qualifying XML elements in a way that corresponds to the nesting suggested by the XML Schema. Notice, for example, that each product display has nested tables for rectangular and cylindrical values.

Sections 4 and 5 illustrate the query set specification QURSED uses to represent the possible queries. Section 6 elaborates on the visual steps the developer follows on the Editor interface to deliver query form and report interfaces, like the one shown in Figure 3, using QURSED.

#### 4. TREE QUERY LANGUAGE (TQL)

User interaction with the query form page results in the generation of TQL queries, which are subsequently translated into XQuery statements. TQL shares many common characteristics with previously proposed XML query languages like XML-QL [27], XML-GL [5], LOREL [21], XMAS [16] and XQuery [26]. TQL facilitates the development of query set specifications that encode large numbers of queries and the development of a visual interface for the easy construction of those specifications. This section describes the structure and semantics of TQL queries. The structure and semantics of query set specifications are described in the next section.

A TQL query  $q$  consists of a *condition tree* and a *result tree*. An example of a TQL query is shown in Figure 4, and corresponds to the TQL query generated by the end-user's interaction with the query form page of Figure 3.

**Definition 1 (Condition Tree).** The condition tree of a TQL query  $q$  is a labeled tree that consists of:

- Element nodes  $n$  having an element name  $name(n)$ , which is a constant, a name variable or a wildcard (\*), and an element variable  $var(n)$ . There can be multiple nodes with the same

constant element name in a condition tree, but element and name variables are unique and are denoted by the \$ symbol.

- AND nodes, which are labeled with a boolean expression  $b$  consisting of predicates combined with the boolean connectives AND, OR and NOT. The predicates consist of arithmetic and comparison operators and functions that use element and name variables and constant values as operands and are understood by the underlying query processor. Each element and name variable used in  $b$  belongs to an element node that is either an ancestor of the AND node, or a descendant of the AND node such that the path from the AND node to the element node does not contain any OR nodes. The boolean expression may also take the values *true* and *false*.
- OR nodes.

The following constraints apply to condition trees:

1. The root element node of a condition tree is an AND node.
2. OR nodes have AND nodes as children.
3. Element nodes with a wildcard as element name can only appear as leaf nodes.

Figure 4 shows the TQL query for the example of Figure 3. Note that conditions are placed on height and width of rectangular sensors, and two conditions are placed on diameter of cylindrical sensors. These conditions correspond to the conditions on Dimensions X and Y on the query form page of Figure 3. Omitted are the variables not used in the condition or the result tree.

The semantics of condition trees is defined in two steps: OR-removal and binding generation. The formal definition of both steps is given in [20]. OR-removal is the process of transforming a condition tree with OR nodes into a forest of condition trees without OR nodes, called *conjunctive condition trees* in the remainder of the paper. Intuitively, OR-removal is analogous to turning a logical expression to disjunctive normal form [13]. OR-removal for the condition tree of Figure 4a produces four condition trees, two of which are shown in Figure 5. The

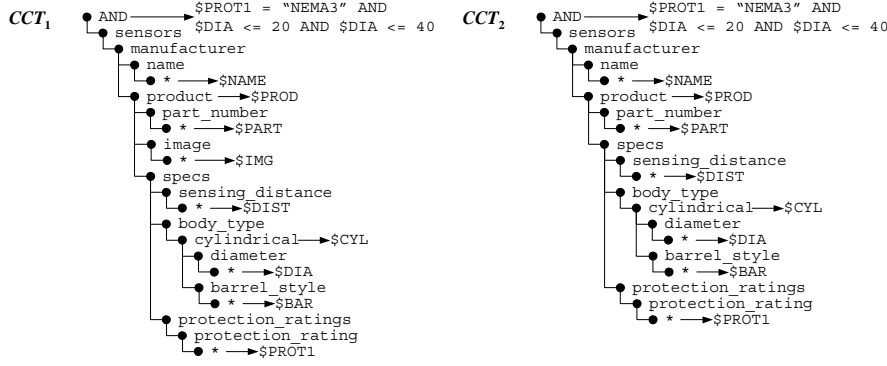


Figure 5. Conjunctive Condition Trees

semantics of the original condition tree is given in terms of the semantics of the resulting conjunctive condition trees.

A conjunctive condition tree  $C$  produces all variable bindings for which an input loto  $t$  “satisfies”  $C$ . For a conjunctive condition tree with element and name variables  $\$V_1, \dots, \$V_k$ , a binding is represented as a tuple  $[\$V_1:v_1, \dots, \$V_k:v_k]$  that binds  $\$V_i$  to node  $v_i$ , where  $1 \leq i \leq k$ . A binding of some of the variables in a (conjunctive) condition tree is called a *partial* binding. A binding requires total tuple assignment [21], i.e., every variable binds to a node or a string value.

The semantics of a condition tree is defined as the union of the bindings returned from each of the conjunctive condition trees in which it is transformed by OR-removal. For example, the result of the four conjunctive condition trees that are the result of OR-removal on the condition tree of Figure 4a on the source loto of Figure 2 is shown in Table 1. The union of the sets of bindings does not remove duplicate bindings or bindings that are subsumed by other bindings (e.g., CCT2 row is subsumed by CCT1 row in Table 1.) The necessary duplicate elimination is performed during construction. Notice that the union is heterogeneous, in the sense that the conjunctive condition trees can contain different element variables and thus their evaluation produces heterogeneous binding tuples.

**Remark.** The semantics of an OR node is that of union and it cannot be simulated by a disjunctive boolean condition labeling an AND node. OR nodes therefore are critically necessary for queries over semistructured data sources (e.g., sources whose XML Schema makes use of choice elements and optional elements.)

The condition tree corresponds intuitively to the WHERE part of XML query languages such as XML-QL [27], LOREL [21] and XMAS [16], to the *extract* and *match* parts of XML-GL [5], and

to the FOR and WHERE clauses of a FLWR expression of the upcoming XQuery standard [26]. As is described in what follows, the result tree correspondingly maps to the CONSTRUCT clause of XML-QL and XMAS, the SELECT clause of LOREL, the *clip* and *construct* parts of XML-GL, and the RETURN clause of a FLWR expression of XQuery. A result tree specifies how to build new XML elements using the bindings provided by the condition tree.

**Definition 2 (Result Tree).** A result tree of a TQL query  $q$  is a node-labeled tree that consists of:

- Element nodes  $n$  having an element name  $name(n)$ , which is either a constant (if  $n$  is an internal node or a leaf node) or a variable (if  $n$  is a leaf node) that appears in the condition tree of  $q$ .
- A group-by label  $G$  and a sort-by label  $S$  on each node. A group-by label  $G$  is a (possibly empty) list of variables  $[\$V_1, \dots, \$V_n]$  from the condition tree of  $q$ . A sort-by label  $S$  is also a list of variables from the condition tree of  $q$ , where an ascending or descending order is determined for each variable. Each variable in the sort-by list of a node must appear in the group-by list of the same node.

Every occurrence of an element or name variable in an element node must be in the scope of some group-by list. Similar to logical quantification, the scope of a group-by list of a node is the subtree rooted at that node. Figure 4b shows the result tree for the example of Figure 3. Note that we omit the headers of the HTML tables from the result tree because of space limitations.

Given a TQL query with condition tree  $CT$  and result tree  $RT$ , the answer of the query on given input is constructed from the set of bindings of  $CT$ . The result is a loto constructed by structural recursion on the result tree. The recursion uses partial bindings of

Table 1 Bindings for Conjunctive Condition Trees of Figure 5

\$NAME	\$PROD	\$PART	\$IMG	\$DIST	\$BODY	\$CYL	\$DIA	\$BAR	\$PROT1	
Turck	product part_number "A123"	A123	A123.jpg	11	cylindrical	cylindrical diameter "17"	17	Smooth	NEMA3	CCT <sub>1</sub>
Turck	product part_number "A123"	A123		11	cylindrical	cylindrical diameter "17"	17	Smooth	NEMA3	CCT <sub>2</sub>
Turck	product part_number "B123"	B123		25	rectangular				NEMA3	CCT <sub>4</sub>

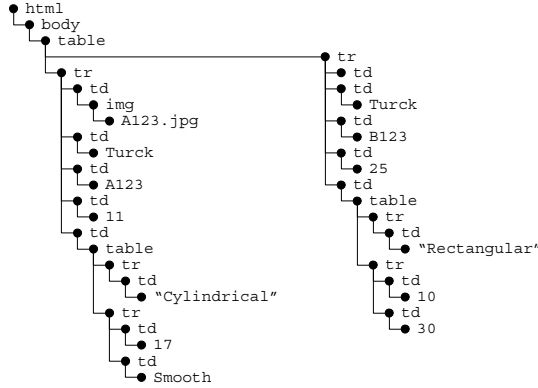


Figure 6. Resulting loto for Bindings of Table 1

the variables to instantiate the group-by variables of element nodes. The formal semantics of result trees can be found in [20].

Figure 6 shows the resulting loto from the TQL query of Figure 4 and the bindings of Table 1. For each of the two distinct partial bindings of the triple  $[\$PROD, \$NAME, \$DIST]$ , one `tr` element node is created. For each such binding and `tr` element, different subtrees are created, corresponding to the two different bindings.

The QURSED system uses the TQL queries internally, but issues queries in the XQuery language [26] by translating TQL queries to equivalent XQuery statements. The algorithm for translating TQL queries to equivalent XQuery statements is given in [20]. The TQL query generated by a query form page is a member of the set of queries encoded in the query set specification of the QFR. The next section describes the syntax and semantics of query set specifications.

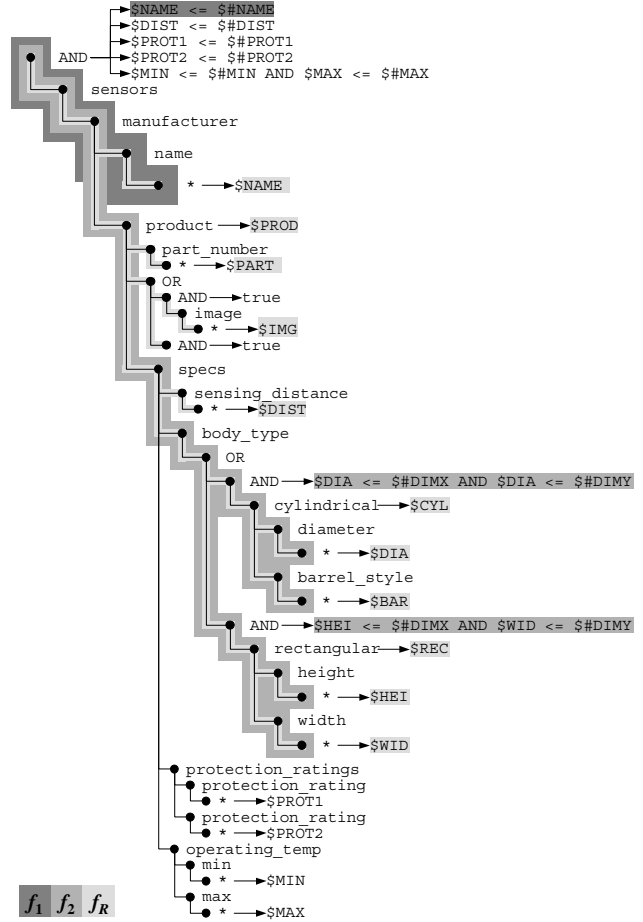
## 5. QUERY SET SPECIFICATION

Query set specifications are used by QURSED to succinctly encode in QFRs large numbers of possible queries. In general, the query set specification can describe a number of queries that is exponential in the size of the specification.

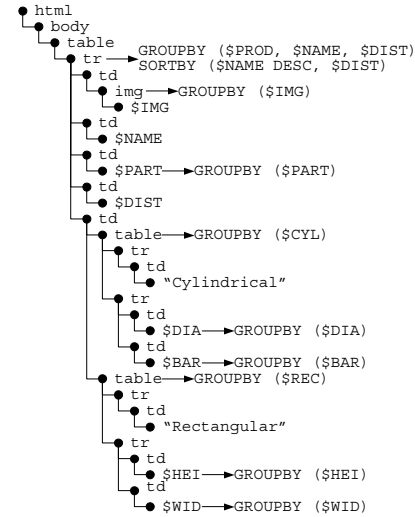
The developer uses the Editor to visually create a query set specification, like the one in Figure 4. This section formally presents the query set specification that is the logical underpinning of QFRs, including the visual interfaces and interactions described in Section 6.

**Definition 3 (Query Set Specification).** A query set specification  $QSS$  is a triple  $\langle CTG, RT, F \rangle$ , where:

- $CTG$ , the *condition tree generator*, is a condition tree with two modifications. First, AND nodes  $a_i$  can be labeled with a set of boolean expressions  $B(a_i)$ , and second, boolean expressions can use *parameters* (a.k.a. placeholders [15]) as operands of their predicates. Parameters are denoted by the  $\$$  symbol and must have a primitive type [25]. The same constraints apply to a  $CTG$  as to a condition tree.
- $RT$  is a result tree.
- $F$  is a non-empty set of *condition fragments*. A condition fragment  $f$  is defined as a subtree of the  $CTG$ , rooted at the root node of  $CTG$ , where each AND node  $a_i$  is labeled with exactly one boolean expression  $b \in B(a_i)$ . Each variable used in  $b$  belongs to a node included in  $f$ .  $F$  always contains a special condition fragment  $f_R$ , called *result fragment*, that



(a) Condition Tree Generator



(b) Result Tree

Figure 7. Query Set Specification

includes all the element nodes whose variables appear in  $RT$  and all its AND nodes are labeled with the boolean value *true* and has no parameters. The result fragment intuitively guarantees the “safety” of the result tree.

For example, the query set specification of Figure 7 encodes, among others, the TQL query of Figure 4. The *CTG* in Figure 7a corresponds partially to the set  $F$  of condition fragments defined for the query form page of Figure 3. Three condition fragments are indicated with different shades of gray: the subtree and the boolean expression on the root AND node of condition fragment  $f_1$  (dark gray) that applies a condition to the child of the `name` element node; the subtree and the boolean expressions of condition fragment  $f_2$  (medium gray) that apply conditions to the element nodes of the dimensions of cylindrical and rectangular sensors; and the subtree of the result fragment  $f_R$  (light gray) that includes all the element nodes whose variables appear in *RT* in Figure 7b.

Given a partial valuation  $v$  over  $P$ , where  $P$  is a subset of the parameters appearing in the query set specification, the set of TQL queries  $Q$  encoded by a query set specification  $QSS$  consists of:

1. the set of condition trees generated by
  - a. instantiating each parameter  $p_i$  that appears in *CTG* and is a member of  $P$  with the constant value  $v(p_i)$ .
  - b. picking a subset  $S \subseteq F$  of condition fragments that includes the result fragment  $f_R$  and creating the tree *CT* that is their union.
  - c. for each AND node  $n_{AND}$  in *CT*, labeling it with the conjunction of the boolean expressions that label  $n_{AND}$  in each condition fragment  $f \in S$ .
2. the result tree *RT*.

The condition fragments included in the subset  $S \subseteq F$  must have all their parameters instantiated during Step 1 above. Such condition fragments are called *active* fragments. Since the partial valuation  $v$  does not provide values for all the parameters used in *CTG*, some condition fragments will be *inactive* and cannot participate in  $S$ . During the end-user's interaction with the query form page, whenever the end-user fills out all the form controls on the query form page that correspond to the parameters of a condition fragment  $f$ , then  $f$  becomes active, and it is automatically included in  $S$  by the QURSED run-time engine.

Figure 4 shows a TQL query, where the condition tree is generated from the query set specification of Figure 7 by the following steps:

- Use the constant values the end-user provides in Figure 3 to instantiate the corresponding parameters. More specifically, the partial valuation  $v$  is  $v(\$ \# \text{PROT1}) = \text{"NEMA3"}$ ,  $v(\$ \# \text{DIMX}) = \text{"20"}$  and  $v(\$ \# \text{DIMY}) = \text{"40"}$ .
- Include in  $S$  the condition fragment  $f_2$ , which imposes conditions on the dimensions of the sensor's body type, the condition fragment that imposes a condition on protection rating (not indicated in Figure 7a), and the result fragment  $f_R$ . The condition fragment  $f_1$  on manufacturer's name is excluded from  $S$ , because the parameter  $\$ \# \text{NAME}$  used in its boolean expression is not instantiated, as Figure 3 shows.
- Take the union of the condition fragments in  $S$ .
- Label the root AND node of Figure 4a with the boolean expression that imposes a condition on protection rating, and the other two AND nodes with the boolean expressions that impose conditions on the sensor's dimensions.

The result tree of the TQL query of Figure 4 is the same with the one of the query set specification of Figure 7. How the developer produces a query set specification via the Editor is described in Section 6.

## 6. QURSED EDITOR

This section presents the QURSED Editor, which is the interface the developer uses to build QFRs. The Editor takes as input an XML Schema and allows a set of visual actions that result in the development of a query set specification  $QSS$ . These actions are grouped according to the part of the  $QSS$  they build, namely, condition tree generator and condition fragments, and result tree. The Editor also takes as input two HTML pages, the query form page and the template report page. The query form page is used with actions related to the specification of the condition tree generator and the condition fragments, while the report page is used in actions related to the creation of the result tree. A key benefit of the Editor is that it enables the easy generation of semistructured queries with OR nodes by considering the structure of the schema, namely choice elements, and automatically performing corresponding actions. The following subsections describe the visual actions and their translation to corresponding parts of the query set specification, using the  $QSS$  of Figure 7 and the QFR of Figure 3 as an example.

### 6.1 Building Condition Tree Generators

The developer builds a condition tree generator, like the one in Figure 7a, by defining a set of condition fragments driven by the input schema. Figure 8a shows the main window of the Editor, where the left panel presents the schema in the form of Figure 2, described in Section 3.1, and the right panel presents the query form page. The query form page on the right panel is displayed as an HTML tree that contains a form and a set of form controls, i.e., `select` and `input` elements nodes that have a unique name attribute [29]. The HTML tree corresponds to the page shown on Figure 8b rendered in the Macromedia HomeSite WYSIWYG HTML editor.

The developer uses the Editor to define the condition fragment  $f_1$  of Figure 7a that imposes an equality condition on the manufacturer's name, by performing the four actions indicated by the arrows on Figure 8a:

- Action 1 *Create Condition Fragment*: Click on the "New Condition Fragment" button and provide a unique ID, which is `manufacturer_name` in this case. On the middle panel, a new row appears in the upper table that lists the condition fragments defined so far, and the expression editor opens at the bottom.
- Action 2 *Build Boolean Expression*: In the expression editor, drag 'n' drop the equality predicate that has two, initially unspecified, operands.
- Action 3 *Specify Elements as Operands*: Set the left operand of the equality by dragging 'n' dropping the \* child node of the `name` element node from the schema. The path from the schema root to the dragged element node appears in the left operand box and is also indicated by the highlighting of the \* node on the left panel.
- Action 4 *Bind Form Controls to Operands*: Bind the right operand of the equality predicate to an HTML form control, which will provide the value for the operand at



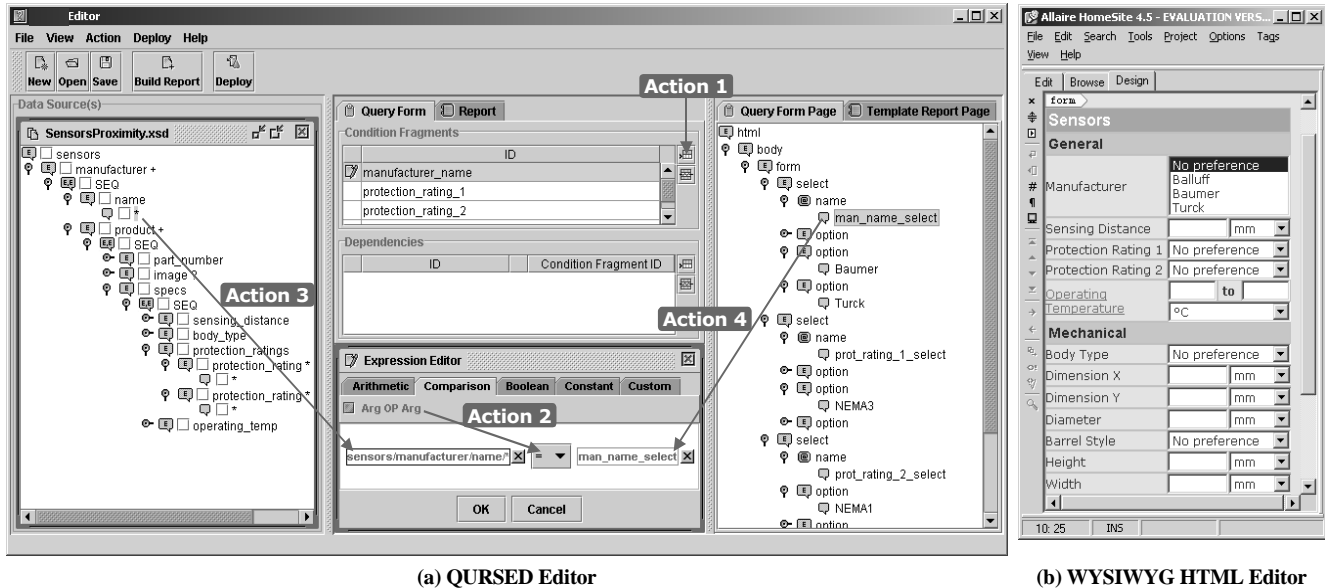


Figure 8. Building a Condition Fragment

run-time. Perform the binding by dragging ‘n’ dropping the `select` element node named `man_name_select` from the query form page. The name of the form control appears in the right operand box.

The actions of Figure 8a generate the subtree of the condition tree generator of Figure 7a that is indicated as condition fragment  $f_1$  from elements in the source schema. The construction of the *CTG* is accomplished by incrementally constructing the subtree necessary for each condition fragment. In particular, in Action 3, the selection of a schema element  $e$  as an operand to a predicate has the following effect to the *CTG*:

- the addition of  $e$  to the *CTG* (if it's not already in *CTG*.)
- the creation of a name variable for  $e$  (again, if one doesn't already exist.)
- the addition of the path from the schema root to  $e$  to the *CTG*, ignoring SEQ and OR elements in the schema.

For example, the developer’s action to drag the \* child node of the name element node from the schema and drop it to the left operand of the equality predicate (Action 3) results in the construction of the path of the condition tree generator of Figure 7a that leads from the root to the \* child node of the name element node and the generation of the  $\$NAME$  element variable. Moreover, Action 4 of Figure 8a binds parameters in the condition fragment to HTML form controls thus establishing a query/visual association and creating the visual fragment corresponding to the condition fragment (cf Section 1.1.) For example, the visual fragment for the condition fragment  $f_1$  of Figure 7a is the “Manufacturer” form control shown in Figure 8b. Action 4 also results in a parameter being created and bound to an HTML form control. In our example, the parameter  $\$NAME$  that appears in the boolean expression of  $f_1$  in Figure 7a is generated and is associated with the `man_name_select` form control on the query form page. Note that, even though the visual actions

generate variables and parameters, the developer does not need to be aware of their existence or semantics.

In the above process, paths from the schema are added to the condition tree generator during the creation of condition fragments *without repetition*. For example, if the developer drags the \* child node of the name element node and drops it to the predicate of another condition fragment  $f$ , only one path will be created in the *CTG*, and only one element variable,  $\$NAME$ , will be associated with the name element node. Both fragments  $f$  and  $f_1$  will use this same path and variable. In general, a path of the schema is not repeated in the condition tree generator and only one variable is generated for each element node. There are cases, though, where the same element node needs to be used multiple times (as in relational self-joins [17].) To accomplish that, the Editor provides the developer with an action that *expands* the schema [18]. This action can be performed only on a repeatable element of the schema and results in multiple copies of the element having the same name appearing on the schema panel of the Editor. Figure 8a shows two copies of the `protection_rating` element created on the schema panel, and the condition tree generator in Figure 7a illustrates the effect of the two condition fragments.

Finally, we demonstrate how the Editor introduces OR nodes in the condition fragments based on the choices of elements that appear in the schema. On the query form page of Figure 8b, the end-user has the option to input two dimensions X and Y that define an envelope for the sensors, without specifying a particular body type, i.e., selecting the “No preference” option of the “Body Type” form control. The schema of Figure 2 shows that sensors can be either cylindrical or rectangular, denoted by the choice (OR) element that has the cylindrical and rectangular elements as children. If the sensor is cylindrical, it has a diameter, and if it is rectangular, it has height and width. The developer defines the condition fragment and builds the following boolean expression for it:

$(\$DIA \leq \$\#DIMX \text{ AND } \$DIA \leq \$\#DIMY) \text{ OR } (\$HEI \leq \$\#DIMX \text{ AND } \$WID \leq \$\#DIMY)$

The Editor detects that the above condition fragment with that boolean expression will generate unsatisfiable queries, since no sensor has both diameter and height, and the semantics of bindings, as explained in Section 4, demand full variable assignment. The Editor then tries to resolve this problem by automatically transforming the OR boolean connective of the above expression to an OR node in the condition fragment, as the resulting condition fragment  $f_2$  in Figure 7a indicates. The OR node has as parent the `body_type` element node, and it intuitively corresponds to the choice element in the schema of Figure 2. Two AND nodes are also introduced, one for each child of the `body_type` element node, having as only child the `cylindrical` and `rectangular` element node respectively. The AND nodes are labeled with the disjuncts in the initial boolean expression:  $(\$DIA \leq \$\#DIMX \text{ AND } \$DIA \leq \$\#DIMY)$  and  $(\$HEI \leq \$\#DIMX \text{ AND } \$WID \leq \$\#DIMY)$ . In general, the Editor brings boolean expressions with disjunction to disjunctive normal form and tries to identify potential unsatisfiable disjuncts: if two element variables in a disjunct correspond to schema nodes that are nested under the same choice element, then the Editor notifies the developer that the expression is unsatisfiable. Otherwise, the Editor tries to rewrite the boolean expression and the condition tree generator to replace the disjunctions with OR nodes in the tree. The checking and rewriting algorithm is given in [20].

## 6.2 Building Result Trees

The Editor allows the developer to easily build the result tree component of a query set specification. The developer only specifies which element nodes of the schema she wants to present on the report page. Then, the Editor automatically builds a result tree that creates report pages presenting the source data in the form of HTML tables that are nested according to the nesting present in the source schema. If the developer wants to structure the report page in a different way than the one the schema dictates, the Editor provides a second option, where the developer provides as input a template report page to guide the result tree

generation. The automatic, schema-driven result tree construction is presented next. Template-driven result tree construction is described in [20].


Name	Part Number	Image	Sensing Distance	Cylindrical	
Turck	A123		11.0	Diameter mm	Barrel Style
				17	Smooth
	Part Number	Image	Sensing Distance	Rectangular	
	B123		25.0	Height mm	Width mm
				10	30

Figure 9. Automatically Generated Report Page

### 6.2.1 Schema-Driven Result Tree Construction

The developer can automatically build a result tree based on the nesting of the input schema. For example, Figure 9 shows a report page created from the result tree for the schema and the data set of Figure 2. The creation of the *RT* and the template report page is accomplished by performing the following two actions, indicated by the numbered arrows on the Editor's window of Figure 10.

**Action 1** *Select Element Nodes*: The developer uses the check boxes that appear next to the element nodes of the schema to select the ones she wants to present on the report page. On Figure 10, the element nodes name, part\_number, image, sensing\_distance, cylindrical and rectangular are selected. This action builds the result fragment  $f_R$  indicated in the condition tree generator of Figure 11a. The variables that will be used in the result tree are also indicated.

**Action 2** *Build the Template Report Page*: The developer clicks on the "Build Report" button and the Editor automatically generates the template report page displayed on the right panel of Figure 10 as a tree of HTML element nodes. It also automatically generates the *element mappings* and the *group-by mappings* that appear in the tables of the middle panel. Figure 11c shows how a WYSIWYG HTML editor renders the template report page.

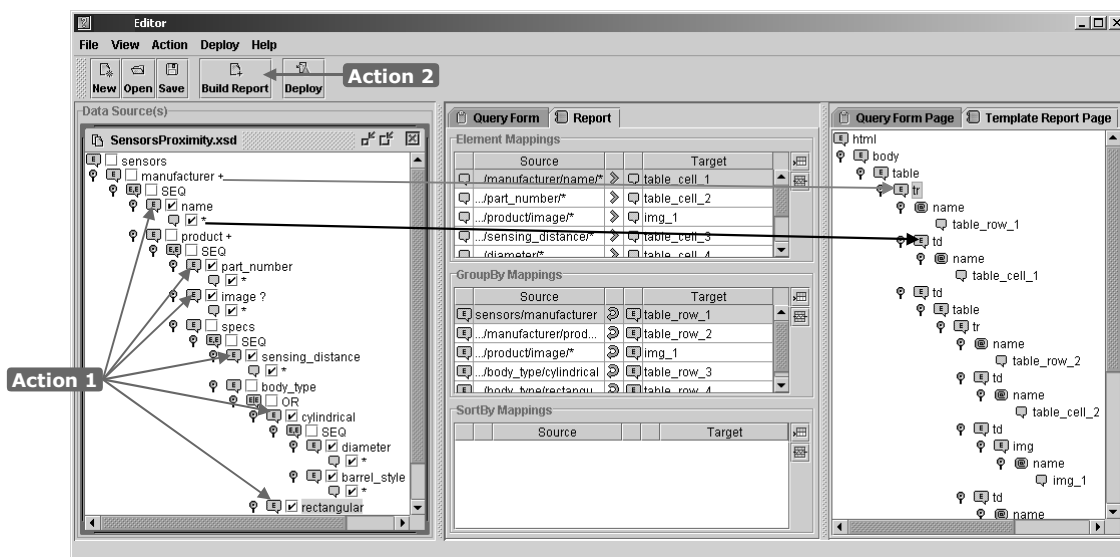


Figure 10. Selecting Elements Nodes and Constructing Template Report Page

With Action 2, the Editor automatically generates the result tree of Figure 11b that presents the element nodes selected in 6.2.1 using HTML table element nodes that are nested according to the nesting of the schema. For illustrating purposes, each table element node in Figure 11b is “annotated” with the schema element node that it corresponds to. This demonstrates, for example, that the “product” table is nested in the “manufacturer” table, as is the case in the schema. The table headers in Figure 11c are also created, from the tag names of the selected element nodes. The headers are omitted from Figure 11b for presentation purposes. In the tables, the Editor places the element variables of the element nodes selected in 6.2.1 as children of td (table data cell) element nodes. For example, in the result tree of Figure 11b the element variable \$NAME appears as the child of the td element node of the “manufacturer” table.

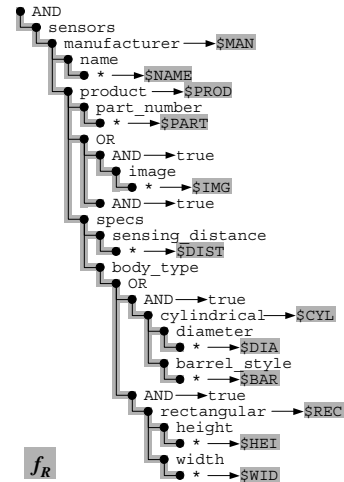
As with the actions of Section 6.1, Action 2 also defines mappings of element nodes from the schema (called by the Editor *source* element nodes) to nodes in the template report page (called *target* element nodes). The mappings appear in the “Element Mappings” table in the middle panel of Figure 10. The target HTML element nodes are identified by the system by their unique name attribute and the path from the root of the schema to the element node identifies the source element nodes. As in the previous section, the effect of the mapping action is that the path from the root of the schema to the selected element nodes is copied to the result fragment of the CTG (if it’s not already there via the actions of Section 6.1), and that an element variable is generated in the result fragment and added as a leaf to the result tree. The placement of the \$NAME variable in the result tree is indicated in Figure 10 with the black arrow from the left to the right panel. The mappings in Figure 10 correspond to the result tree of Figure 11b and the result fragment of the CTG of Figure 11a.

Figure 11a demonstrates the ability of QURSED and TQL to flexibly deal with structural variance and optional element nodes. For example, for the optional image element node selected in 6.2.1, the Editor introduces an OR node to the result fragment with two AND nodes as children, where one of them is labeled with the boolean value *true* and has no children. According to the semantics presented in Section 4, this tree will generate bindings for the sensors that don’t have an image element node, as in the case of sensor “B123” in Figure 9. The Editor also handles the repeatable element nodes and the choice elements (i.e., OR elements) in the schema by

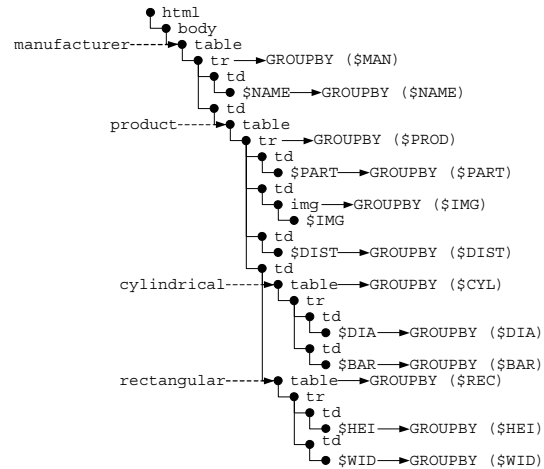
- automatically generating OR nodes in the result fragment.
- automatically generating corresponding table elements and group-by lists in the result tree.

For example, in Figure 11b the existence of the manufacturer element results in the generation the “manufacturer” table element node and the group-by list of its tr (table row) child element node. According to the semantics of Section 4, this group-by list will generate one table row for each binding of the \$MAN element variable. The group-by list of that tr element node is highlighted in the lower middle panel of Figure 10. Also the choice of cylindrical or rectangular element in the schema is translated to

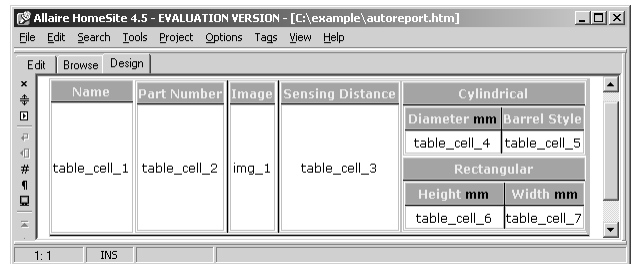
- an OR node in the result fragment in order to generate the bindings for sensors of either body type.



(a) Condition Tree Generator



(b) Result Tree



(c) Template Report Page

Figure 11. Automatically Generated Result Fragment, Result Tree and Template Report Page

- the group-by lists on the “cylindrical” and “rectangular” table element nodes in order to generate the appropriate table depending on each sensor’s body type. The group-by lists appear in the “GroupBy Mappings” table in the lower middle panel of Figure 10.

The complete algorithm for generating the table element nodes and the group-by lists, including the heuristics employed by the algorithm, are given in [20].

## 7. REFERENCES

- [1] S. Abiteboul, P. Buneman, D. Suciu: *Data on the Web*, Morgan Kaufman, California, 2000.
- [2] P. Atzeni, G. Mecca, P. Merialdo: *To Weave the Web*, in proceedings of the 23rd International Conference on Very Large Databases (VLDB), 1997.
- [3] P. Bernstein et al.: *The Asilomar report on database research*, SIGMOD Record 27(4), 1998.
- [4] 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.
- [5] S. Ceri et al.: *XML-GL: a Graphical Language for Querying and Restructuring XML Documents*, in proceedings of WWW8, 1999.
- [6] S. Chawathe, T. Baby, J. Yeo: *VQBD: Exploring Semistructured Data* (demonstration description), in proceedings of the ACM SIGMOD International Conference on Management of Data, page 603, 2001.
- [7] S. Cluet et al.: *Your Mediators Need Data Conversion!*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 1998.
- [8] S. Cohen et al.: *EquiX – Easy Querying in XML Databases*, in proceedings of the ACM Workshop on The Web and Databases (WebDB), 1999.
- [9] M. Fernandez, A. Morishima, D. Suciu: *Efficient Evaluation of XML Middle-ware Queries*, in proceedings of the ACM SIGMOD Conf., 2001.
- [10] M. Fernandez, D. Suciu and I. Tatarinov: *Declarative Specification of Data-intensive Web sites*, in proceedings of the Workshop on Domain Specific Languages, 1999.
- [11] P. Fraternali: *Tools and Approaches for Data Intensive Web Application Development: a Survey*, in the ACM Computing Surveys 31(3), 1999, pp. 227-263.
- [12] P. Fraternali, P. Paolini: *Model-Driven Development of Web Applications: the Autoweb System*, in the ACM Transactions on Office Information Systems 18 (4), 2000.
- [13] M.R. Genesereth and N.J. Nillson: *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann, 1987.
- [14] R. Goldman, J. Widom: *Interactive Query and Search in Semistructured Databases*, in proceedings of the ACM Workshop on The Web and Databases (WebDB), 1998.
- [15] A. Levy, A. Rajaraman, J. D. Ullman: *Answering Queries Using Limited External Processors*, in Principles of Database Systems (PODS), 1996, pp. 227-237.
- [16] B. Ludascher, Y. Papakonstantinou, P. Velikhov: *Navigation-Driven Evaluation of Virtual Mediated Views*, in Extending Database Technology (EDBT), 2000.
- [17] J. Melton, A. R. Simon: *Understanding the new SQL: A Complete Guide*, Morgan Kaufmann, 1993.
- [18] K. Munroe, Y. Papakonstantinou: *BBQ: A Visual Interface for Browsing and Querying XML*, in VDB5, 2000.
- [19] M. Petropoulos, V. Vassalos, Y. Papakonstantinou: *XML Query Forms (XQForms): Declarative Specification of XML Query Interfaces*, in proceedings of WWW10, 2001.
- [20] Y. Papakonstantinou, M. Petropoulos, V. Vassalos: *QURSED: Querying and Reporting Semistructured Data* (extended version.)  
[www.db.ucsd.edu/People/michalis/pubs/sig02ext.pdf](http://www.db.ucsd.edu/People/michalis/pubs/sig02ext.pdf)
- [21] D. Quass et al.: *Querying Semistructured Heterogeneous Information*, in proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases (DOOD), 1995, pp. 319-344.
- [22] H. Schöning, J. Wäsch: *Tamino - An Internet Database System*, in proceedings of EDBT, 2000, pp. 383-387.
- [23] A. Silberschatz, M. Stonebraker, J. D. Ullman: *Database Systems: Achievements and Opportunities - The "Lagunita" Report of the NSF Invitational Workshop on the Future of Database System Research held in Palo Alto, California, February 22-23, 1990*, SIGMOD Record 19(4): 6-22, 1990.
- [24] M. Zloof: *Query By Example*, in proceedings of the National Compute Conference, AFIPS, Vol. 44, 1975, pp. 431-438.
- [25] P. V. Biron, A. Malhotra: *XML Schema Part 2: Datatypes*, W3C Recommendation 02 May 2001.  
<http://www.w3.org/TR/xmlschema-2/>
- [26] D. Chamberlin et al.: *XQuery 1.0: An XML Query Language*, W3C Working Draft 07 June 2001.  
<http://www.w3.org/TR/xquery/>
- [27] A. Deutsch et al.: *XML-QL: A Query Language for XML*, W3C note, 1998.  
<http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/>
- [28] D. C. Fallside: *XML Schema Part 0: Primer*, W3C Recommendation 02 May 2001.  
<http://www.w3.org/TR/xmlschema-0/>
- [29] D. Raggett, A. Le Hors, I. Jacobs: *HTML 4.01 Specification*, W3C Recommendation 24 December 1999,  
<http://www.w3.org/TR/html>.
- [30] H. Garcia Molina et al.: *The TSIMMIS approach to mediation: Data models and Languages*, in the Journal of Intelligent Information Systems 8(2), 1997, pp. 117-132.
- [31] J. Shanmugasundaram et al.: *Efficiently Publishing Relational Data as XML Documents*, in proceedings of the 26th International Conference on Very Large Databases (VLDB), 2000.