Paper ID: 156

QURSED: QUerying and Reporting SEmistructured Data

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, possibly interdependent 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 and guides the development of meaningful dependencies between condition fragments by translating the 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 [28]), XML databases [23] and mediators [7,9,10,17,24] 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 [10,24], 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 [3,25] - albeit semi-neglected by the database research community[1]. They allow millions of web users to selectively view the information of underlying sources. A number of tools [34,35,38] 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 XHTML, for performance reasons.

[1] But hopefully not cursed (or … qursed).

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 *XHTML query form page* that provides the visual (XHTML) part of the form page, including the HTML form controls [40], 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 *XHTML 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/tuples appear in each table. The query form and template report pages are typically developed with an external "What You See Is What You Get" (WYSIWYG) editor, such as Macromedia HomeSite [36]. 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 run-time engine of QURSED – QURSED evaluates the conditions of dependencies, as we will see next.

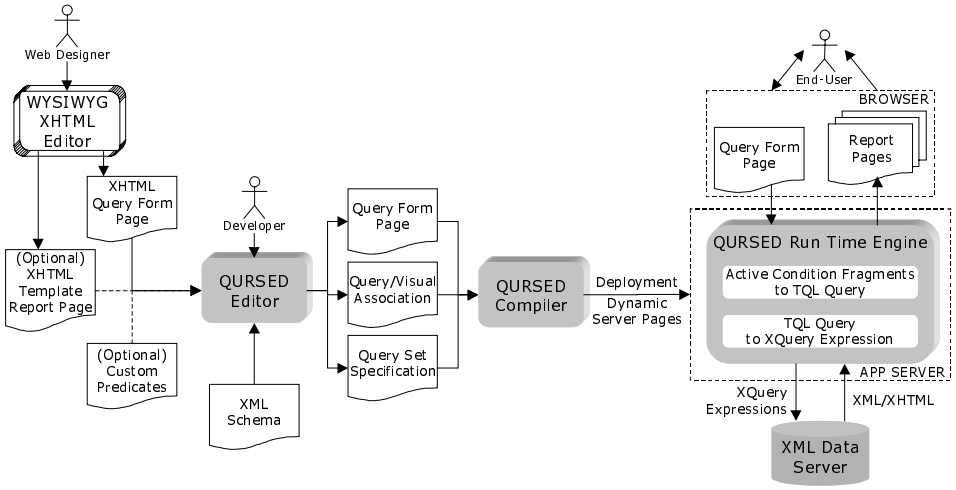


Figure 1 QURSED System Architecture

Then the Editor displays the XML Schema and the XHTML 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*, the fragment *dependencies* 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* [40] of the query form page. The form controls that are associated with the parameters contained in a condition fragment constitute its *visual fragment*. Dependencies can be established between condition fragments and between the values of parameters and fragments, and provide fine-grained control on what queries can be reasonably submitted and, on the visual side, what visual fragment are eligible for completion at each point. For example, if the user selects rectangular sensors, the visual fragment that allows us to put a condition on diameter has to be hidden. Finally, the result tree specifies how the source data instantiate and populate the XHTML 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 [33], which control the interaction with the end-user. Active Server Pages [37] are another possible option, though not implemented. 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 enforce the dependencies 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 XHTML 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. As we will see, TQL makes this combination straightforward. Then the TQL query is translated into an XQuery expression that directly produces the XHTML 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 that is the visual front-end to the development of a QFR, including its query set specification.

2 RELATED WORK & NOVEL CONTRIBUTIONS OF QURSED

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

1. *Web-based Form and Report Generators*, such as Macromedia DreamWeaver Ultradev [34], ColdFusion [35], and Microsoft Visual Interdev [38]. All of the above enable the development of web-based applications that create form and report pages that access relational databases, with the exception of [21], which targets XML data. QURSED is classified in the same category, except for its focus on semistructured data.
2. *Visual Querying Interfaces*, such as QBE [26] and Microsoft's Query Builder (part of Visual InterDev [38]), which target the querying of relational databases, and EquiX [8], BBQ [20], VQBD [6], the Lorel's DataGuide-driven GUI [15], and PESTO [4], which target the querying of XML or object-oriented databases.
3. *Data-Intensive Web Site Generators*, such as Autoweb [13], Araneus [2] and Strudel [11]. These are recent research projects proposing new methods of generating web sites, which are heavily based on database content. An additional extensive discussion on this class of systems can be found in [12].

Web-based Form and Report Generators create web-based interfaces that access relational databases. Popular examples are Macromedia Dreamweaver UltraDev [34], ColdFusion [35], and Microsoft Visual InterDev [38]. 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 [40] 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 that these products 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. The three most popular custom tag libraries today are Sun's Java Server Pages [33], Microsoft's Active Server Pages [37] and Macromedia ColdFusion Markup Language [35].

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 are applications that allow the exploration of the schema and/or content of the underlying database and the formulation of queries. Typical examples are the Query-By-Example (QBE) [26] interface and Microsoft's Query Builder [38], which target the querying of relational databases. Recent tools such as EquiX [8], BBQ [20], VQBD [6], the Lorel's DataGuide-driven GUI [15], and PESTO [4] target the querying of XML and object-oriented databases. 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.

In particular, EquiX allows the visual development of complex XML queries that include quantification, negation and aggregation. It requires basic knowledge of query language primitives and DTDs. Simple predicates, boolean expressions and variables can be typed at terminal nodes and quantifiers can be applied to non-terminal nodes. In a QBE-like manner, the user can select which elements of the DTD to "print" in the output but the XML structure of the query result conforms to the XML structure of the source, i.e., there is no restructuring ability.

Overall, EquiX provides many interesting ideas to be used by a subsequent version of the Editor, which will go beyond the class of queries currently supported. However, one has to be very careful about drawing the parallel between the described visual query formulation tools and the Editor: The goal of the former is the development of a query or a query template from a user who is familiar with database models and schemas. The goal of the latter is the construction from an average web developer of a form that represents and can generate a large number of possible queries.

Data-Intensive Web Site Generators. Autoweb [13], Araneus [2] and Strudel [11] are excellent examples of the ongoing research on how to design and develop web sites heavily dependent on database content. 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 and its model is very 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 also of grouping and representing different reports using a single site graph node.

Finally, there is very recent work on systems for the declarative generation of Web-based query forms and reports that generate XQueries [21]. The paper describes 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. It also presents different "hard-wired" ways the system provides for customizing the appearance of reports. The set of queries produced by the system are conjunctive and its spectrum is narrow because of the limitations of the XML Schema-based

annotation. The paper does not describe how the system encodes or composes queries and results of queries based on user actions.

Finally, there is the emerging XForms W3C standard [30], which promotes the use of XML structured documents for communicating to the web server the results of the end-user's actions on various kinds of forms. XForms also tries to provide constructs that change the appearance of the form page on the client side, without the need of coding. When XForms implementations become available QURSED will use these constructs for the evaluation of dependencies, thus simplifying the implementation.

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 take into consideration 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. Condition fragment dependencies control what the end-user can ask at every point. For example, consider another version of the sensor query form: Now it contains a selection menu where the user can specify whether he is interested in cylindrical or rectangular sensors. Once this is known, the form transforms itself to display conditions (e.g., diameter) that pertain to cylindrical sensors only or conditions (e.g., height and width) that pertain to rectangular only.
3. 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/tuples 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 XHTML editor. Then the developer can focus on the logical aspects of the forms and reports: Which are the condition fragments? What are their dependencies? 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 and dependencies. The combinations of the fragments lead to large numbers of parameterized queries, while the dependencies guarantee that the produced queries make sense given the XML Schema and the semantics of the data.

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.

XML, XHTML, and XQuery-Based Architecture. The QURSED architecture and implementation fully utilizes XQuery and the interplay of XML/XHTML. The result is an efficient overall system, when compared either against relational-based front-end generators or against conventional XML-based front-end architectures, such as Oracle's XSQL [39]. An XML-related efficiency is derived by the fact that XML is used throughout QURSED: XML is the data model of the source on which XML queries, in XQuery syntax, are evaluated, and is also used to deliver the presentation - in the form of XHTML.[2]

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 + suffices) elements and the choices and sequences (`OR` and `SEQ` elements) of elements. Also, the elements of primitive type [27] 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 [31]. 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.

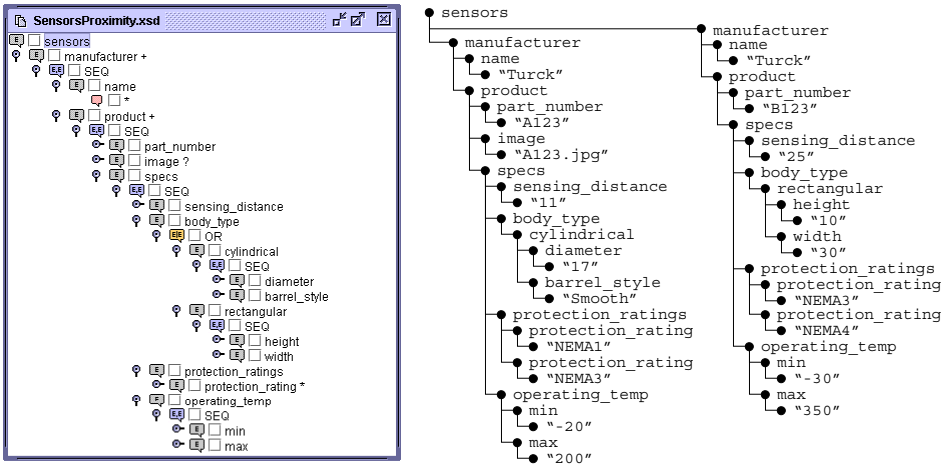


Figure 2 Example XML Schema and Conforming Data Set

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

[2] Though QURSED uses XML both as a source data model and to deliver XTHML results to the browser, the latter use is transparent to developers: they can simply enjoy the increased performance that results from the elimination of internal model mismatches.

elements that make up the content of the element. A leaf node holds the string value of its parent node. If *n* is a node of a loto, we denote as *tree*(*n*) the subtree rooted at *n*.

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.

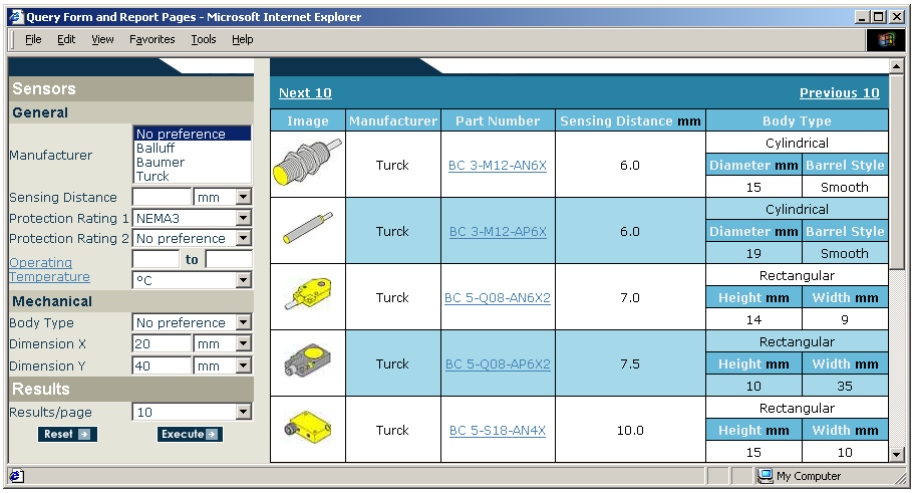


Figure 3 Example QFR Interface

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

As we will see in more detail in Section 6, the developer has specified a dependency between the end-user's input in the "Body Type" `select` form control and the form controls that ask about the sensors' dimensions. If the requested sensors are rectangular, the end-user would be prompted instead for height and width, since, according to the XML Schema of the sensors, these inputs make sense in this case.

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.

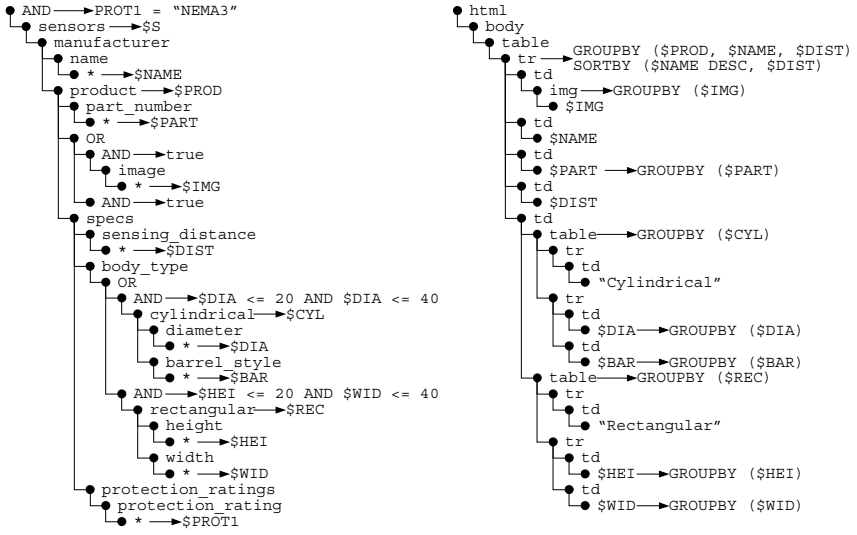
The following section illustrates the query model QURSED uses to represent the possible queries. Section 6 elaborates on the visual steps the developer follows on the Editor interface to deliver query form and report interfaces, like the one shown in Figure 3, using QURSED.

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 [29], XML-GL [5], LOREL [22], XMAS [17] and XQuery [28]. 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.



(a) Condition Tree **(b) Result Tree**

Figure 4 TQL Query Corresponding to 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 two conditions are placed on diameter of cylindrical sensors corresponding to height and width of rectangular sensors. Omitted are the variables that are not used in the condition or the result tree.

The semantics of condition trees is defined in two steps: OR-removal and binding generation. 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. OR-removal for the condition tree of Figure 4a results in the set of the four condition trees shown in Figure 5.

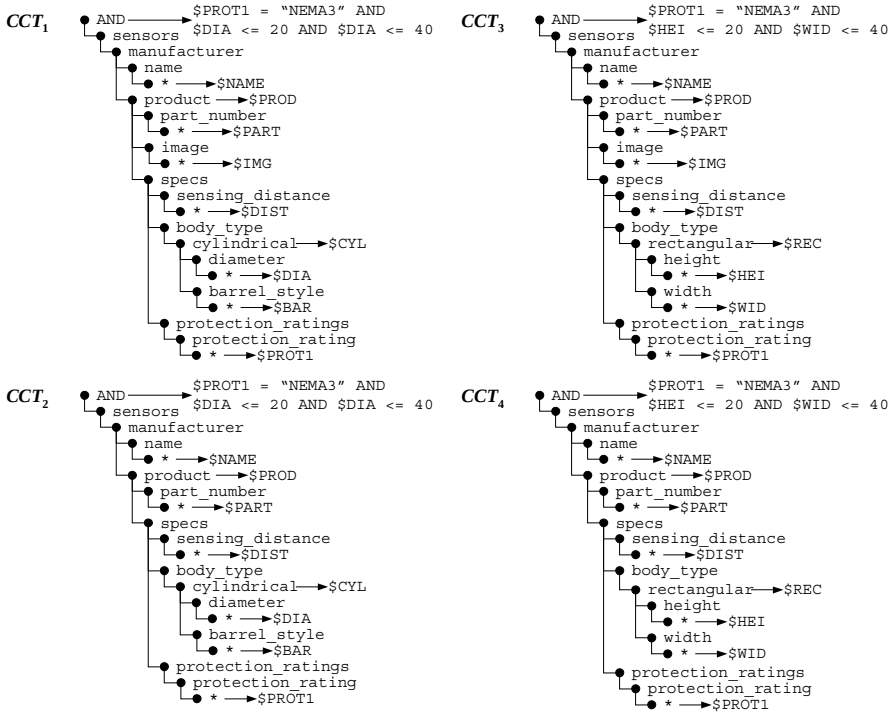


Figure 5 Conjunctive Condition Trees

Intuitively, OR-removal is analogous to turning a logical expression to disjunctive normal form [14]. In particular, we repeatedly apply the rules shown in Figure 6. Notice that when the subtrees of Figure 6 are presented with 2 or 3 children, this is without loss of generality. At the point when we cannot apply the rules further, we have produced a tree with an OR root node, which we replace with the forest of conjunctive condition trees consisting of all the children of the root OR node. Notice that wherever this process generates AND nodes as children of AND nodes, these can merged, and the boolean expression of the merged node is the conjunction of the boolean expressions of the original AND nodes. Also notice that the boolean expression of the root AND node in the first rule cannot contain any variables in subtrees B or C, per earlier definition of condition trees. Finally, notice that in the course of OR-removal "intermediate results" may not be valid condition trees per Definition 1 (in particular, constraint 2 can be violated), but the final results obviously are. The semantics of the original condition tree is given in terms of the semantics of the resulting conjunctive condition trees.

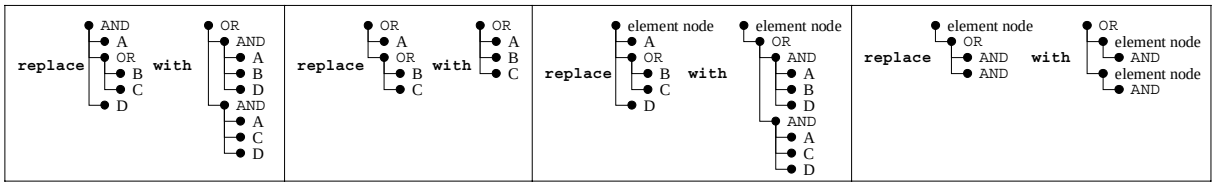


Figure 6 OR-Removal Replacement Rules

A conjunctive condition tree C produces all bindings for which an input loto t "satisfies" C. Formally, a binding is a mapping β from the set of element variables and name variables in C to the nodes and node labels of t, such that the child of the root of C (which is the AND node) matches with the root of t, i.e., $\beta(var(\text{child}(root(C)))) = root(t)$, and recursively, traversing the two trees top-down, for each child n_i of an element node n in C, assuming $var(n)$ is mapped to a node x in t, there exists a child x_i of x, such that $\beta(var(n_i)) = x_i$ and, if x_i is not a leaf node:

- if $name(n_i)$ is a constant, $name(n_i) = name(x_i)$
- if $name(n_i)$ is a name variable, $\beta(name(n_i)) = name(x_i)$

Importantly, AND notes in *C* are ignored in the traversal of *C*. In particular, in the definition above, by "child of the element", we mean either element child of the element, or the child of an AND node that is the child of the element. A binding is qualified if it makes *true* the boolean expressions that label the AND nodes of *C*.[3]

The result of *C* is the set of qualified bindings. For a conjunctive condition tree with element and name variables $\$V_1,\ldots,\$V_k$, a binding is represented as a tuple $[\$V_1{:}v_1,\ldots,\$V_k{:}v_k]$ that binds $\$V_i$ to node v_i, where $1 \le i \le k$. A binding of some of the variables in a (conjunctive) condition tree is called a *partial* binding. Note that the semantics of a binding requires total tuple assignment [22], i.e., every variable binds to a node or a string value.

Table 1 Bindings for Conjunctive Condition Trees of Figure 5

	$NAME	$PROD	$PART	$IMG	$DIST	$BODY	$CYL	$DIA	$BAR	$REC	$HEI	$WID	$PROT1
***CCT*$_1$**	Turck	product / part_number / "A123"	A123	A123.jpg	11	cylindrical	cylindrical / diameter / "17"	17	Smooth				NEMA3
***CCT*$_2$**	Turck	product / part_number / "A123"	A123		11	cylindrical	cylindrical / diameter / "17"	17	Smooth				NEMA3
***CCT*$_4$**	Turck	product / part_number / "B123"	B123		25	rectangular				rectangular / height / "10"	10	30	NEMA3

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 shown in Figure 5 on the source loto of Figure 2 is shown in Table 1. The union of the sets of bindings does not need to 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 three of the four conjunctive condition trees generate one binding each. Notice also 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 [29], LOREL [22] and XMAS [17], 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 [28]. As the following section describes, 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)[4] list of variables $[\$V_1,\ldots,\$V_n]$ 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.

[3] Notice that it is easy to do AND-removal on conjunctive condition trees. Let $a_1,\ldots,a_n$ be the AND nodes in a conjunctive condition tree with root *a*, and let $b_1,\ldots b_n$, *b* be their boolean expressions. We can eliminate $a_1,\ldots a_n$ from the tree, and replace *b* with *b AND* b_1 *and*…*and* b_n.

[4] Empty group-by and sort-by labels are omitted from figures in the remainder of the paper.

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 also that the headers of the HTML tables are omitted from the result tree. We omit the headers 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 qualified bindings of *CT*. In what follows, binding refers to qualified binding. The result is a loto constructed by structural recursion on the result tree as follows. The recursion uses partial bindings of the variables to instantiate the group-by variables of element nodes.

Traversing the result tree top-down, for each subtree *tree*(*n*) rooted at element node *n* with group-by label [$V_1,\ldots,V_k$] and, without loss of generality, sort-by label [$V_1,\ldots,V_m$] ($m \le k$), let $\mu=[V_{A1}:v_{A1},\ldots,V_{An}:v_{An}]$ be a partial binding that instantiates all the group-by variables of the ancestors of *n*. Recursively replace the subtree *tree*(*n*) in place with a list of subtrees, one for each binding $\pi=[V_{A1}:v_{A1},\ldots,V_{An}:v_{An},V_1:v_1,\ldots,V_k:v_k]$ such that $v_1,\ldots,v_m$ are string values, by instantiating all occurrences of $V_{A1},\ldots,V_{An},V_1,\ldots,V_k$ with $v_{A1},\ldots,v_{An},v_1,\ldots,v_k$. The list of instantiated subtrees is ordered according to the conditions in the sort-by label.

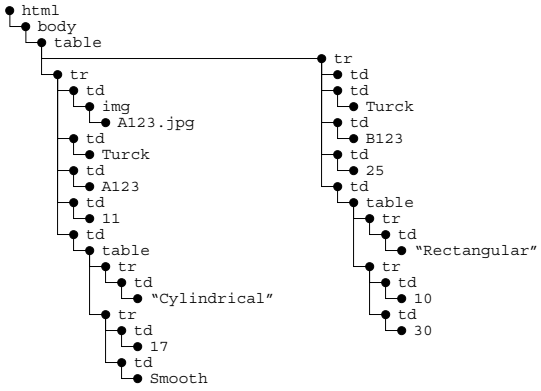


Figure 7 Resulting loto for Bindings of Table 1

Figure 7 shows the resulting loto from the TQL query of Figure 4 and the bindings of Table 1. Note, for example, that for each of the two distinct partial bindings of the triple [`$PROD`, `$NAME`, `$DIST`], one `tr` element node is created. Also note that for each such binding, different subtrees rooted at the nested `table` element node are created, corresponding to different π bindings.

The QURSED system uses the TQL queries internally, but issues queries in the (upcoming) standard XQuery language by translating TQL queries to equivalent XQuery statements. The algorithm for translating TQL queries to equivalent XQuery statements is given in Appendix 0. The XQuery specification is a working draft of the World Wide Web Consortium (W3C); for a more detailed presentation of the language and its semantics see [28] and [32].

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 specification also includes a set of dependencies that constrain the set of queries that can be produced.

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 4-tuple <*CTG*, *RT*, *F*, *D*>, where:

- *CTG* is called *condition tree generator*. *CTG* 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 [16]) as operands of their predicates. Parameters are denoted by the `$#` symbol and must have a primitive type [27]. 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 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.
- *D* is an optional set of dependencies that are defined in Section 5.1.

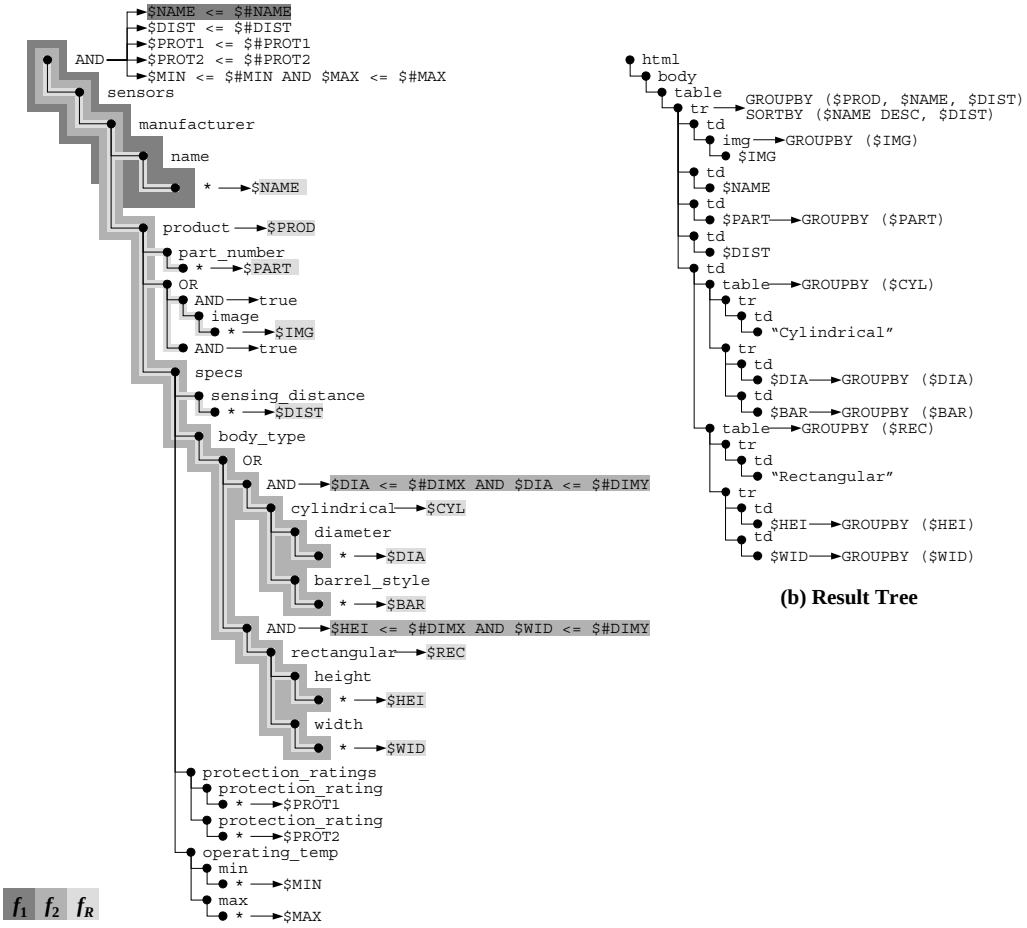


Figure 8 Query Set Specification

For example, the query set specification of Figure 8 encodes, among others, the TQL query of Figure 4. The *CTG* in Figure 8a 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 node of the `name` element node; the subtree and the boolean expressions on the AND nodes that are children of the OR node of condition fragment f_2

(medium gray) that applies a condition to the element nodes that correspond to 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 8b.

Given a partial valuation ν over *P*, where *P* is a subset of the parameters that appear 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 $\nu(p_i)$.
   b. picking a subset $S \subseteq F$ of condition fragments that includes the result fragment f_R and creating the tree *CT* that is the union of them.
   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 set of result trees generated by instantiating each parameter p_i that appears in *RT* and is a member of *P* with the constant value $\nu(p_i)$.

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 ν 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 she 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 is also automatically included in *S* by the QURSED run-time engine. The situation in the presence of dependencies is described in Section 5.1.

Figure 4 shows a TQL query, where the condition tree is generated from the query set specification of Figure 8 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 ν is ν(`$#PROT1`)="NEMA3", ν(`$#DIMX`)="20" and ν(`$#DIMY`)="40".
- Include in *S* the condition fragment f_2, which imposes a condition on the dimensions of the sensor's body type, the condition fragment that imposes a condition on protection rating (not indicated in Figure 8a), and the result fragment f_R. The condition fragment f_1 on manufacturer's name is excluded from *S*, because the parameter `$#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 8. How the developer produces a query set specification via the Editor is described in Section 6.

5.1 Dependencies

Dependencies are the last component of a query set specification and allow the developer to define conditions that include or exclude condition fragments from the condition tree depending on the end-user's input. Dependencies provide a flexible way to handle data irregularities and structural variance in the input data, and also provide a powerful and declarative way to control the appearance of a query form, by controlling the appearance of visual fragments. We will discuss both uses of dependencies in this section.

Definition 4 (Dependency). A dependency *d* is defined as a 3-tuple <*f*, *b*, *H*> over a set of condition fragments *F*, where $f \in F$ is the *dependent* condition fragment and *b* is the *condition* of the dependency consisting of predicates combined with the boolean connectives AND, OR and NOT. The predicates consist of arithmetic and comparison operators and

possibly custom functions that use parameters and constant values as operands. The set $H \subseteq F$ contains the condition fragments that use at least one parameter that appears in b. We say that the dependent fragment f *participates* in the dependency d.

A dependency d holds if all the parameters of b are instantiated in active condition fragments in H, and b evaluates to *true*. In the presence of dependencies, a fragment is active if all its parameters are instantiated *and* at least one of the dependencies it participates in holds. Intuitively, a set of dependencies constrains the set of queries a query set specification can generate by rendering inactive the condition fragments that participate in dependencies that all of them do not hold.

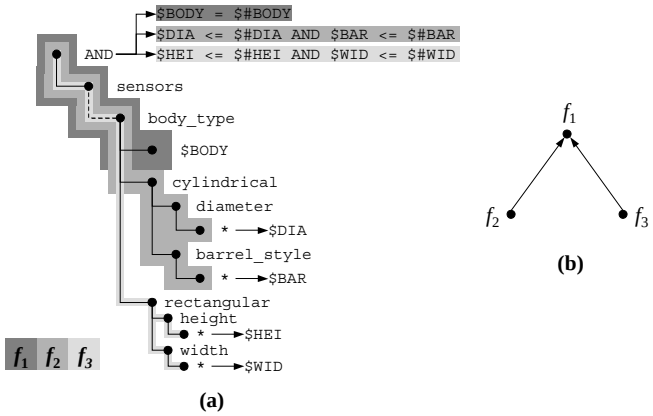


Figure 9 Condition Tree Generator and Dependencies Graph

For example, consider the condition tree generator and condition fragments of Figure 9a, and let the developer define two dependencies d_1 and d_2 as follows:

$$\langle f_2,\ \$\#BODY = \text{“cylindrical”},\ \{f_1\}\rangle \quad (d_1)$$
$$\langle f_3,\ \$\#BODY = \text{“rectangular”},\ \{f_1\}\rangle \quad (d_2)$$

The condition fragment f_1 uses the parameter `$#BODY` that appears in the condition of both dependencies on f_2 and f_3. If a value is not provided for `$#BODY`, then neither dependency holds, and f_2 and f_3 are inactive. If the value "cylindrical" is provided, then f_1 is active, the condition for d_1 is true, and so f_2 is rendered active.

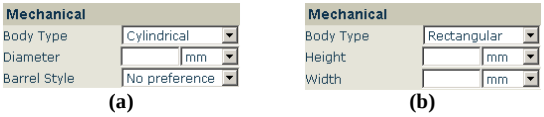


Figure 10 Dependencies on the Query Form Page

Dependencies are a powerful declarative way of affecting the appearance of a query form page as well. In particular, QURSED hides from the query form page those visual fragments whose condition fragments participate in dependencies that do not hold. For example, Figure 10 demonstrates the effect of dependencies d_1 and d_2 on the query form page of Figure 3, where different form controls are shown. The two shown sets of form controls are the visual fragments of the condition fragments shown in Figure 9a. For instance, the condition fragment f_1 applies a condition to the element node labeled with `$BODY` and its visual fragment consists of the “Body Type” form control. End-user selection of the “Cylindrical” option in the “Body Type” form control results in having d_1 hold, which makes the visual fragment for f_2 visible (Figure 10a.) Notice that f_2 is still inactive: values for “Diameter” and “Barrel Style” need to be provided. Notice also that an inactive condition fragment whose dependencies do not hold has no chance of becoming active: its visual fragment is hidden, so there is no way to provide values for the parameters of the condition fragment.

From the examples above, it becomes evident that circular dependencies, where for instance two condition fragments appear in the head of each other's dependencies, should be avoided, as the dependent fragments may never become active. To illustrate this, we define the *dependency graph* as follows:

Definition 5 (Dependency Graph). A dependency graph for a set of dependencies D and a set of condition fragments F is a directed graph $<V, E>$, where the nodes V are the condition fragments in F and for every dependency d in D there is an edge in E from the dependent condition fragment f to every condition fragment f_i in the head of D.

The dependency graph for the dependencies d_1 and d_2 defined above is shown in Figure 9b. In the presence of a non-trivial dependency between f_1 and f_2 (that would create a cycle), the visual fragment of f_1 would remain hidden, which means that there is no way to activate f_1 and consequently f_2. QURSED enforces the following simple sufficient condition:

The dependency graph of a set of dependencies must be acyclic.

If the dependency graph is acyclic, the set of dependencies is *acceptable.* The QURSED system activates the appropriate visual fragments (updating the query form page) and condition fragments, based on which parameters have been provided and which dependencies hold. The relevant algorithm for "resolving" the dependencies is based on topological sort [18] and is omitted because of space limitations. Note that dependencies are handled before or during the end-user interaction with the query form page, and therefore prior to the generation of the TQL query. Section 6.3 describes how the developer can define dependencies using the Editor.

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 enable 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, result tree, and dependencies. 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, the condition fragments and the dependencies, 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 8 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 8a, by defining a set of condition fragments driven by the input schema. Figure 11a 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 XHTML tree that contains a form and a set of form controls, i.e., `select` and `input` elements nodes that have a unique `name` attribute [40]. The XHTML tree corresponds to the page shown on Figure 11b rendered in the Macromedia HomeSite [36] WYSIWYG HTML editor.

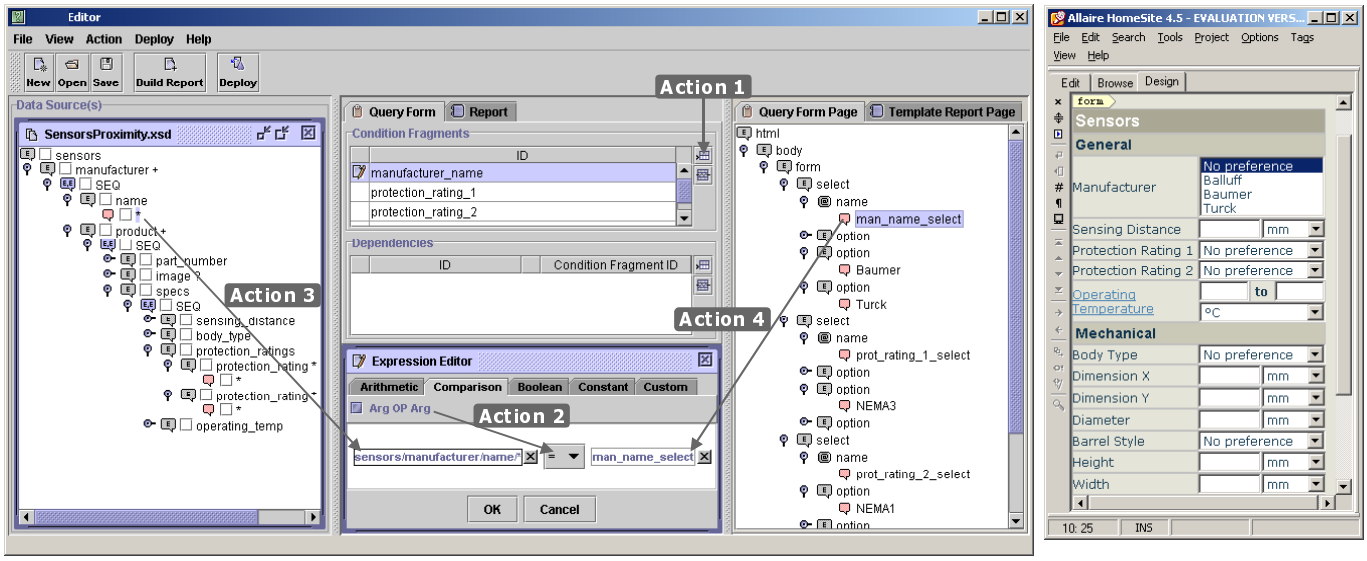
The developer uses the Editor to define the condition fragment f_1 of Figure 8a that imposes an equality condition on the manufacturer's name, by performing the four actions indicated by the arrows on Figure 11a:

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 root of the schema 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 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.



(a) QURSED Editor **(b) WYSIWYG HTML Editor**

Figure 11 Building a Condition Fragment

The actions of Figure 11a generate the subtree of the condition tree generator of Figure 8a 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 8a that leads from the root to the `*` child node of the `name` element node and the generation of the `$NAME` element variable. Moreover, Action 4 of Figure 11a binds parameters in the condition fragment to HTML form controls thus establishing a query/visual association and creating the visual fragment corresponding to the condition fragment (cf Section 1.1.) For example, the visual fragment for the condition fragment f_1 of Figure 8a is the "Manufacturer" form control shown in Figure 11b. Action 4 also results in a parameter being created and bound to an HTML form control. In our example, the parameter `$#NAME` that appears in the boolean expression of f_1 in Figure 8a is generated and is associated with the form control named `man_name_select` 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 [19].)

Consider, for example, that the developer wants to give the end-user the ability to specify two protection ratings for a sensor, as shown in Figure 11b (note that the `protection_rating` element is repeatable in the schema.) In this case, the developer builds two condition fragments, each with an equality predicate on `protection_rating`, and needs to visually specify that each one uses a different protection rating value. To accomplish that, the Editor provides the developer with an action that *expands* the schema[5]. 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 11a shows the two copies of the `protection_rating` element created on the schema panel, and the condition tree generator in Figure 8a illustrates the effect of the two condition fragments, where two copies of the `protection_rating` element node have two different and unique element variables, `$PROT1` and `$PROT2`, which are used in the corresponding boolean expressions on the root AND node. The parameters `$#PROT1` and `$#PROT2` are associated with the corresponding form controls on the query form page of Figure 11b.

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 11b, 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. In this case the developer defines the condition fragment and builds the following boolean expression for it:

```
($DIA <= $#DIMX AND $DIA <= $#DIMY) OR ($HEI <= $#DIMX AND $WID <= $#DIMY)
```

She builds this boolean expression by dragging 'n' dropping one OR connective in the expression editor and then two AND connectives as operands. The `$DIA`, `$HEI` and `$WID` variables are generated by dragging the `*` child nodes of the `diameter`, `height` and `width` elements from the schema and dropping them as operands of the AND connectives. The `$#DIMX` and `$#DIMY` parameters are associated with the "Dimension X" and "Dimension Y" form controls shown on the query form page of Figure 11b.

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 8a 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 conjunctions in the initial boolean expression: `($DIA <= $#DIMX AND $DIA <= $#DIMY)` and `($HEI <= $#DIMX AND $WID <= $#DIMY)`. In general, the Editor brings boolean expressions with disjunction to disjunctive normal form and tries to identify potential unsatisfiable conjuncts: if two element variables in a conjunct 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

[5] The basic visual concept of expandable schema exists in some forms in the BBQ system [20] and the EquiX system [8].

and the condition tree generator to replace the disjunctions with OR nodes in the tree. The checking and rewriting algorithm is given in Appendix A.

6.2 Building Result Trees

The Editor provides two options for the developer to build the result tree component of a query set specification, each one associated with a set of corresponding actions. For the first and simpler option, 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, as shown in Figure 1, to guide the result tree generation. Both options and their actions are described next.

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

Figure 12 Automatically Generated Report Page

6.2.1 Automatic Result Tree Construction

The developer can automatically build a result tree based on the nesting of the input schema without having to input a template report page to the Editor. For example, Figure 12 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 13.

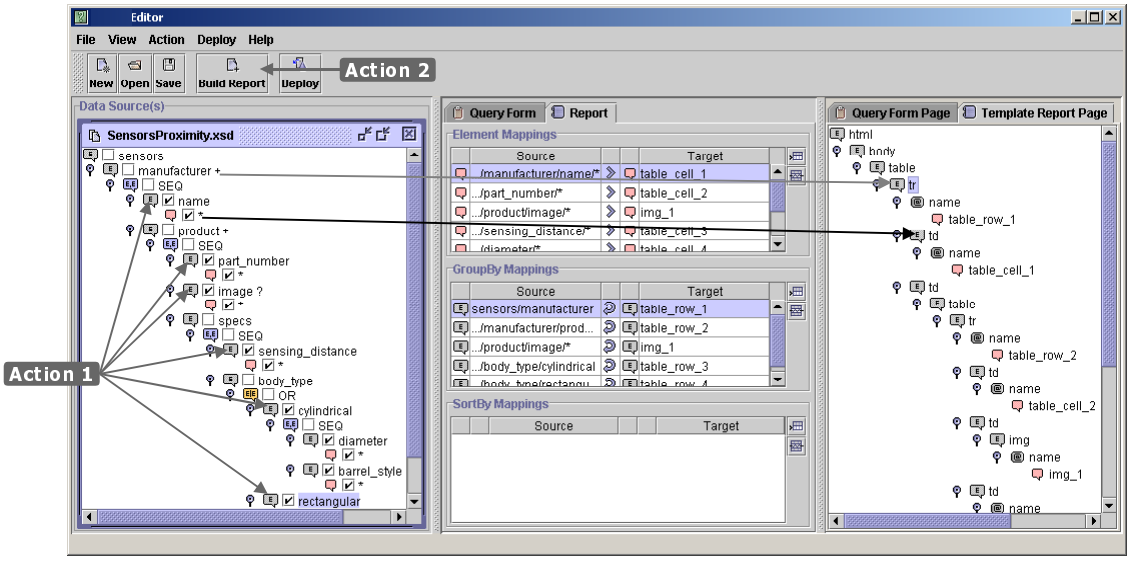


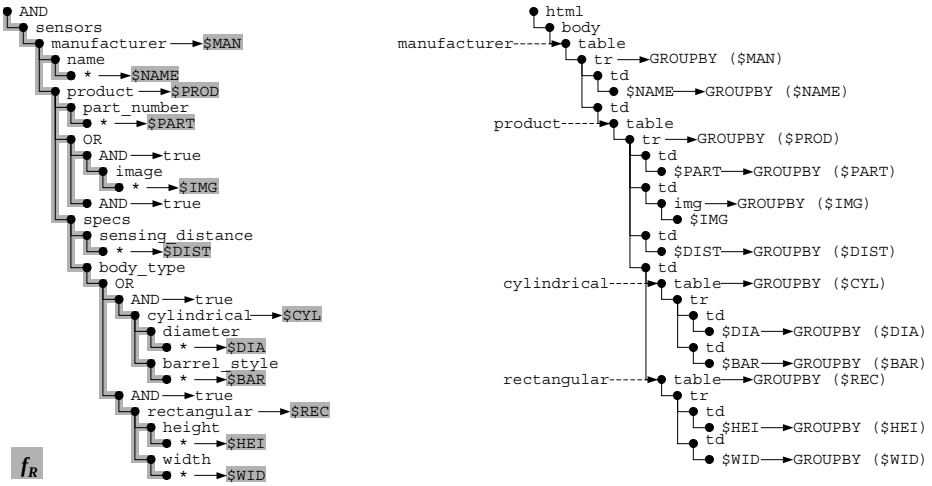
Figure 13 Selecting Elements Nodes and Constructing Template Report Page

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 13, 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 14a. The variables that will be used in the result tree are also indicated.

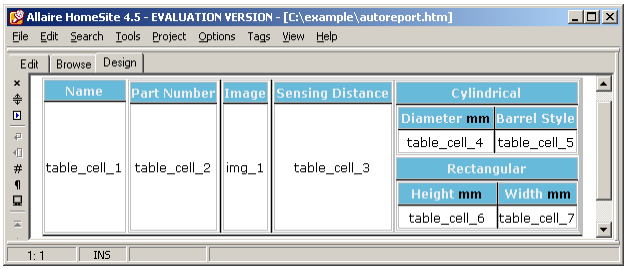
Action 2. *Build the Template Report Page*: The developer clicks on the "Build Report" button and the Editor automatically generates the template report page displayed on the right panel of Figure 13 as a tree of HTML

element nodes. It also automatically generates the *element mappings* and the *group-by mappings* that appear in the tables of the middle panel. These mappings are explained in detail in Section 6.2.2 where the developer, instead of the Editor, constructs the result tree. Figure 14c shows how a WYSIWYG HTML editor renders the template report page.

In Action 2, the Editor automatically generates the result tree of Figure 14b 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 14b 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 14c are also created, from the name labels of the selected element nodes. The headers are omitted from Figure 14b 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 14b the element variable `$NAME` appears as the child of the `td` element node of the "manufacturer" table.



(a) Condition Tree Generator **(b) Result Tree**



(c) Template Report Page

Figure 14 Automatically Generated Result Fragment, Result Tree and Template Report Page

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 13. 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, for element `name` is indicated in Figure 13 with the black arrow from the left to the right panel.

The mappings in Figure 13 correspond to the result tree of Figure 14b and the result fragment of the condition tree generator of Figure 14a.

Figure 14a 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 12. 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 14b 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 corresponding to the `manufacturer` element is indicated with the gray arrow of Figure 13. 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.
- 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 middle panel of Figure 13.

The complete algorithm for generating the `table` element nodes and the group-by lists, including the heuristics employed by the algorithm, are omitted for lack of space.

6.2.2 Result Tree Construction Based On Template Report Page

The developer can create more sophisticated report pages and result trees by providing to the Editor a template report page she has constructed with an HTML editor. For example, on the report page of Figure 3 the developer wants to display the manufacturer's name for each sensor product, unlike the report page on Figure 12, that followed the schema nesting pattern, where the `product` is nested in the `manufacturer` element node. To accomplish that, she constructs the template report page shown in Figure 15 and provides it to the Editor.

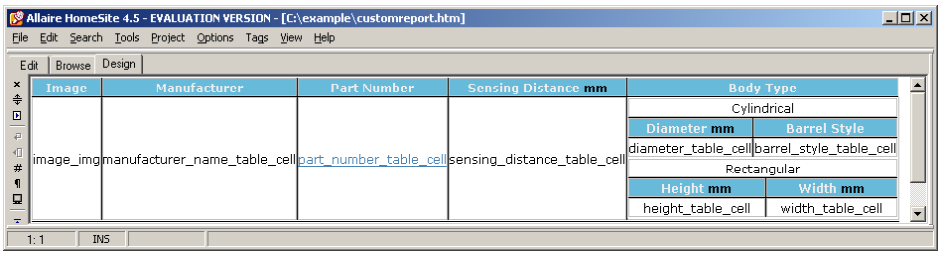


Figure 15 Editing the Template Report Page

Figure 16 displays on the right panel the template report page. Using the schema panel and the template report page panel, the developer constructs the result tree of the query set specification of Figure 8. In particular, the structure of the result tree is the structure of the template report page. The rest of the result tree (element variables, group-by and sort-by lists) is constructed by performing the following four actions:

Action 1. *Create New Mapping*: The developer creates a new element, group-by or sort-by mapping by adding a new row to the corresponding table in the middle panel of Figure 16. Depending on what mapping was created, one of Actions 2, 3, or 4 is performed.

Action 2. *Perform Element Mappings*: The developer drags element nodes from the schema and drops them to leaf nodes of the template report page. The action places the element variable for the selected element node in the result tree, as well as in the result fragment of the *CTG* (if it's not there already via the actions of Section 6.1.) For example, by dragging the `*` child node of the `name` element node and dropping it on the `td` element node in the template report page, the developer implicitly places the `$NAME` variable in the result tree of Figure 8b.

Action 3. *Perform Group-By Mappings*: The developer drags element nodes from the schema and drops them to any nodes of the template report page. For example, by dragging the `product` element node and dropping it on the `tr` element node of the outermost table in the template report page the developer generates the group-by list of that element, shown in Figure 8b. This particular action essentially flattens the `manufacturer` element node by displaying the manufacturer's name (`$NAME` variable) for each product (`$PROD` variable), as Figure 3 shows.

Action 4. *Perform Sort-By Mappings*: Same as Action 3, but the developer additionally specifies an order. For example, by dragging the `sensing_distance` element node, dropping it on the `tr` element node of the outermost table and specifying ascending order the developer generates the sort-by list of that element, shown in Figure 8b. The Editor performs automatically a group-by mapping for each sort-by mapping, enforcing the semantics of Section 4.

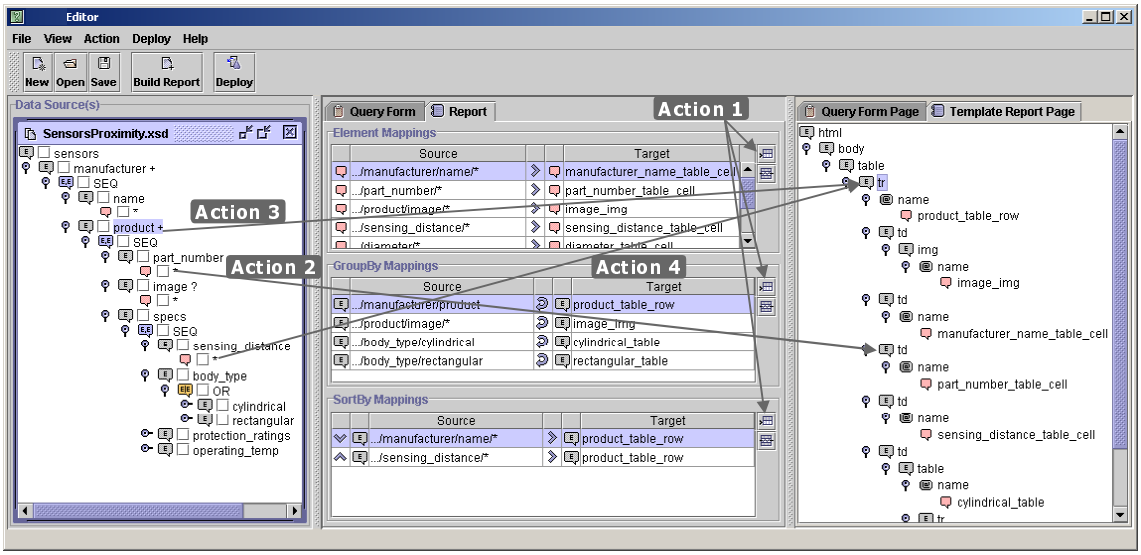


Figure 16 Performing Element and Group-By Mappings on the Template Report Page

An engineering benefit from the way the developer builds the result tree is that the template report page can easily be opened from any external HTML editor and further customized visually, even after the mappings have been performed. The mappings between schema element nodes and template report page element nodes are retained on the template report page through the use of the `name` attribute of the HTML elements. The names for the `td` (table data cell) element nodes are shown on Figure 16.

6.3 Building Dependencies

Figure 10 of Section 5.1 demonstrates the effect of dependencies, where the end-user's option in the "Body Type" `select` form control renders inactive the condition fragments that query the dimensions of cylindrical or rectangular sensors, and hide the corresponding visual fragments on the query form page.

The Editor provides a set of actions to allow the developer to build a dependency, i.e., to select the dependent condition fragment and to construct the condition of the dependency. Figure 17 displays a set of condition fragments as rows of the upper table on the middle panel, and part of the query form of Figure 3 on the right panel. The set of

condition fragments contains the "cylindrical" condition fragment that applies a condition to the `diameter` and `barrel_style` element nodes, and is indicated as condition fragment f_2 in Figure 9a. This condition fragment is associated with the "Diameter" and "Barrel Style" form controls of the query form page of Figure 3 that constitute its visual fragment. The developer builds the dependency d_1 of Section 5.1 on the "cylindrical" condition fragment by performing the following set of actions indicated on Figure 17 by the numbered arrows:

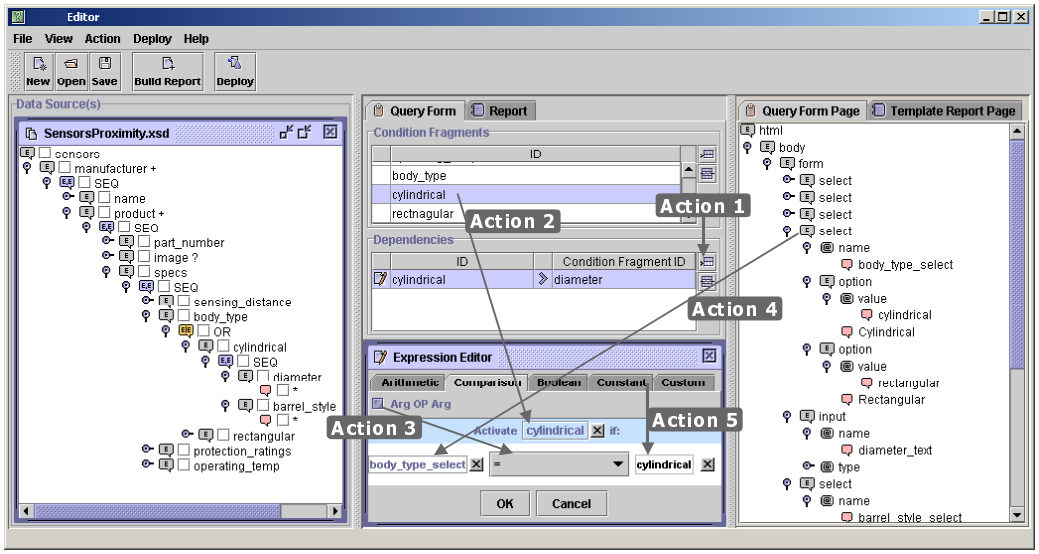


Figure 17 Building Dependencies

Action 1. *Create Dependency*: Click on the "New Dependency" button and enter a descriptive `ID`. On the middle panel, a new row appears in the lower table that lists the dependencies, and the expression editor opens at the bottom.

Action 2. *Select Condition Fragment*: Drag 'n' drop the "cylindrical" condition fragment to the expression editor's "Activate" box. This selected condition fragment will be inactive if the condition defined next evaluates to *false*.

Action 3. *Build Condition*: In the expression editor, drag 'n' drop the equality predicate.

Action 4. *Bind Form Controls to Parameters*: Specify that the left operand of the equality predicate is a parameter by dragging 'n' dropping the "Body Type" `select` form control from the query form page. Thus the left operand takes the value the end-user chooses for body type.

Action 5. *Specify Constant Values as Operands*: Specify the right operand of the equality predicate by dragging 'n' dropping a string constant and typing "cylindrical". Note that only constant values and parameters that bind to form elements can be used in the condition, as described in Section 5.1.

7 CONCLUSIONS

We presented QURSED, a system for the generation of web-based interfaces for querying and reporting semistructured data. We described the system architecture and the formal underpinnings of the system, including the Tree Query Language for representing semistructured queries, and the succinct and powerful query set specification for encoding the large sets of queries that can be generated by a query form. We described how the tree queries and the query set specification accommodate the needs of query interfaces for semistructured information through the use of condition fragments, OR nodes and dependencies. We also presented the QURSED Editor that allows the GUI-based specification of the interface for querying and reporting semistructured data, and described how the intuitive visual actions result in the production of the query set specification and its association with the visual aspects.

In the future we will extend the functionality of the Web-based and visual aspects of the system. The Editor will provide a WYSIWYG HTML view of the right-hand side panels that render the visual component of the forms page and

the report template. From a query functionality point of view, we will extend the set of queries that can be expressed with TQL and we will correspondingly increase the power of the query set specification. A challenge will be to enhance the query power while the Editor's interface is as intuitive as it is now.

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, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, L. Tanca: *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, C. Delobel, J. Siméon, K. Smaga: *Your Mediators Need Data Conversion!*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 1998.

[8] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, A. Serebrenik: *EquiX – Easy Querying in XML Databases*, in proceedings of the ACM Workshop on The Web and Databases (WebDB), 1999.

[9] D. Draper, A. Halevy, D. Weld: *The Nimble Integration Engine*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 2001.

[10] M. Fernandez, A. Morishima, D. Suciu: *Efficient Evaluation of XML Middle-ware Queries*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 2001.

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

[12] P. Fraternali: *Tools and Approaches for Data Intensive Web Application Development: a Survey*, in the ACM Computing Surveys 31(3), 1999, pp. 227-263.

[13] P. Fraternali, P. Paolini: *Model-Driven Development of Web Applications: the Autoweb System*, in the ACM Transactions on Office Information Systems 18 (4), 2000.

[14] M.R. Genesereth and N.J. Nillson: *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann, 1987.

[15] R. Goldman, J. Widom: *Interactive Query and Search in Semistructured Databases*, in proceedings of the ACM Workshop on The Web and Databases (WebDB), 1998.

[16] A. Levy, A. Rajaraman, J. D. Ullman: *Answering Queries Using Limited External Processors*, in Principles Of Database Systems (PODS), 1996, pp. 227-237.

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

[18] D. E. Knuth: *The Art of Computer Programming*, vol. 3: Sorting and Searching, Addison Wesley, 1973.

[19] J. Melton, A. R. Simon: *Understanding the new SQL: A Complete Guide*, Morgan Kaufmann, 1993.

[20] K. Munroe, Y. Papakonstantinou: *BBQ: A Visual Interface for Browsing and Querying XML*, in VDB5, 2000.

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

[22] 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.

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

[24] J. Shanmugasundaram et al.: *Efficiently Publishing Relational Data as XML Documents*, in proceedings of the 26th International Conference on Very Large Databases (VLDB), 2000.

[25] 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.

[26] M. Zloof: *Query By Example*, in proceedings of the National Compute Conference, AFIPS, Vol. 44, 1975, pp. 431-438.

[27] P. V. Biron, A. Malhotra: *XML Schema Part 2: Datatypes*, W3C Recommendation 02 May 2001. http://www.w3.org/TR/xmlschema-2/

[28] D. Chamberlin et al.: *XQuery 1.0: An XML Query Language*, W3C Working Draft 07 June 2001. http://www.w3.org/TR/xquery/
[29] A. Deutsch et al.: *XML-QL: A Query Language for XML*, W3C note, 1998. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/
[30] M. Dubinko et al.: *XForms Requirements*, W3C Working Draft 04 April 2000. http://www.w3.org/TR/xhtml-forms-req
[31] D. C. Fallside: XML Schema Part 0: Primer, W3C Recommendation 02 May 2001. http://www.w3.org/TR/xmlschema-0/
[32] P. Fankhauser et al.: *XQuery 1.0 Formal Semantics*, W3C Working Draft 07 June 2001. http://www.w3.org/TR/query-semantics/
[33] Java Server Pages, White Paper http://java.sun.com/products/jsp/whitepaper.html
[34] Macromedia Dreamweaver UltraDev http://www.macromedia.com/software/ultradev/
[35] Macromedia ColdFusion http://www.macromedia.com/software/ultradev/special/coldfusion/
[36] Macromedia HomeSite http://www.macromedia.com/software/homesite/
[37] Microsoft ASP.NET http://www.asp.net/
[38] Microsoft Visual InterDev http://msdn.microsoft.com/vinterdev/
[39] Oracle XSQL Pages and the XSQL Servlet http://technet.oracle.com/tech/xml/xsql_servlet/htdocs/relnotes.htm
[40] D. Raggett, A. Le Hors, I. Jacobs: *HTML 4.01 Specification*, W3C Recommendation 24 December 1999. http://www.w3.org/TR/html4/

A. ORNodes Algorithm

Algorithm ORNodes
Input: A boolean expression *b* for a condition fragment *f*, and the subtree *t* of the condition tree generator *CTG* constructed from *f*
Output: Error if *f* can produce unsatisfiable queries, or *t* with possibly new OR and AND nodes
Method:

1 Rewrite *b* in disjunctive normal form such that $b = c_1$ OR c_2…OR c_n, where c_i is a conjunction of predicates
2 For each conjunction c_i, where $1 \le i \le n$, that uses variables $\$V_{i1}$, $\$V_{i2}$,…, $\$V_{ik}$
  2.1 If there are two variables $\$V_{ix}$, $\$V_{iy}$, where $1 \le x,y \le k$, that label element nodes in the schema whose nearest common ancestor in the schema is a choice (OR) element, , then output that the condition fragment *f* produces unsatisfiable queries
3 Mark all variables as not visited
4 For each two conjunctions c_i and c_j, where $1 \le i,j \le n$ and $i \ne j$, and for each two variables $\$V_{ix}$, $\$V_{jy}$ used in c_i and c_j respectively such that at least one of them is not visited
  4.1 If both the paths from the elements e_{ix}, e_{jy} labeled with $\$V_{ix}$ and $\$V_{jy}$ in the schema to their nearest element node common ancestor n_{ANSC} contain a choice (OR) element, then
    4.1.1 If n_{ANSC} does not have an OR child node in *t*, , then insert an OR node n_{OR} as its child
    4.1.2 Insert in *t* two AND nodes, a_1 and a_2, as children of the OR node n_{OR}
    4.1.3 Remove from *t* the two children of $n_{ANSC,}$, $tree_{ix}$ and $treet_{jy}$, that contain e_{ix}, e_{jy}
    4.1.4 Insert in *t* the subtrees $tree_{ix}$ and $treet_{jy}$ as children of the AND nodes a_1 and a_2 respectively
    4.1.5 Mark $\$V_{ix}$ and $\$V_{jy}$ as visited
5 For each conjunction c_i, where $1 \le i \le n$
  5.1 Identify the AND node a_i such that $tree(a_i)$ contains all the element nodes labeled with the variables used in c_i and label it with boolean expression c_i
  5.2 If an AND node in *t* is labeled with more than one conjunction c_i, then
    5.2.1 Combine them with the OR boolean connective

B. TQL Query to XQuery Expression Algorithm (TQL2XQ)

The algorithm TQL2XQ works on TQL queries, presented in Section 4. TQL2XQ generates an XQuery expression equivalent to the input TQL query. The XQuery expressions generated by TQL2XQ include `GROUPBY` expressions to perform the groupings. `GROUPBY` expressions are not part of the latest XQuery working draft [28], but the draft includes an issue on an explicit `GROUPBY` construct.

Notice that TQL work on the original TQL query, without performing OR-removal. OR-removal is potentially exponential, but it is used only to define the semantics of TQL.

TQL2XQ makes the following two constraining assumptions for the condition tree *CT* and the result tree *RT* of the input TQL query:

1. AND nodes in *CT* have element nodes as children.
2. Element nodes in *CT* with a name variable as element name are leaf nodes.

The Editor guarantees that the *CTG* and *RT* it produces cannot generate a TQL query that violates these assumptions.

TQL2XQ inputs the root n_{CT} of a condition tree *CT* and the root n_{RT} of a result tree *RT* of a TQL query. An element node *r* of *RT* has a name *name*(*r*), a group-by list *G* of variables *var*(G_r) and a sort-by list *S*.

TQL2XQ outputs an XQuery expression in the form of nested `FWR` (`FOR-WHERE-RETURN`) expressions, where a `FWR` expression *e* is nested in the `RETURN` clause of its parent. The `FOR` clause of *e* introduces a list of variables *var*(*e*) each one associated with a path expression in *CT*. The `WHERE` clause is a condition *cond*(*e*) that filters the bindings of the variables introduced in the `FOR` clause.

We say that an element node *r* of *RT* is in the scope of the `RETURN` clause of a `FWR` expression *e*, e.g., *scope*(*r*)=*e*, if *r* will be constructed in the `RETURN` clause of *e*. Initially, the root n_{RT} of *RT* is in the scope of a `FWR` expression, where the `FOR` clause associates *var*(n_{CT}) with the path expression `document("source.xml")`.

Algorithm TQL2XQ
Input: n_{CT}, n_{RT}, *scope*(n_{RT})
Output: An XQuery expression
Method: Traversing *RT* top-down and left-to-right, an element node *r* of *RT*:

1 If G_r is not empty
  1.1 If there is a variable $V_i in *var*($G_r$) and not in *var*(*scope*(*r*)) that maps to element node *n* in *CT* and a variable $V_j in any of the subtrees rooted at the siblings of *r* that maps to element node *n'* in *CT* and the nearest common ancestor of *n* and *n'* is an OR node
    1.1.1 Create a new `FWR` expression *e'*, nested in *e*, and set *scope*(*r*)← *e'*
  1.2 For every variable $V_i in *var*($G_r$) and not in *var*(*scope*(*r*)) that maps to element node *n* in *CT*
    1.2.1 Append $V_i in the list of variables that the `FOR` clause of *scope*(*r*) introduces and associate $V_i with a path expression *pe* in *CT* that starts from the element variable $V_j of the nearest ancestor of *n*, where $V_j is in *var*(*scope*(*r*)) or in *var*(*ancestor*(*scope*(*r*))) of the nearest ancestor `FWR` expression of *scope*(*r*), and navigates to *n*. Associate $V_i with `name(`*pe*`)` if $V_i is a name variable.
    1.2.2 For every AND node in *CT* with a boolean expression b_i that *pe* "crosses" set *cond*(*scope*(*r*))← $b_i \wedge$ *cond*(*scope*(*r*))
  1.3 Output `GROUPBY` *var*(G_r) `AS`
2 If *name*(*r*) is a constant
  2.1 Output <*name*(*r*)>
  2.2 For each child c_i of *r*
    2.2.1 TQL2XQ(n_{CT}, c_i, *scope*(*r*))
  2.3 Output </*name*(*r*)>
3 Else If *name*(*r*) is a variable
  3.1 Output {*name*(*r*)}
4 If S_r is not empty
  4.1 For each variable $V_i in S_r with order *order*($V_i)
    4.1.1 Replace $V_i in S_r with a path expression from a top constructed element in the `RETURN` clause of *scope*(*r*) to the element that *r* constructed
  4.2 Output `SORTBY` (S_r)

TQL2XQ cannot output an XQuery expression in the following case: there are two element nodes r_1 and r_2 in *RT* having variables $V_1 and $V_2 as element names that map to element nodes n_1 and n_2 in *CT*, r_1 is an ancestor of r_2, and both the paths from the elements n_1, n_2 to their nearest common ancestor contain an OR node.