The QURSED System *

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

Yannis Papakonstantinou
Computer Science and Eng. Dept.
University of California, San Diego
yannis@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, 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. An on-line demonstration of the system is available at http://www.db.ucsd.edu/QURSED/.

Contact Author: Michalis Petropoulos – Phone: +1 858 587 1771, Fax: +1 707 336 5469

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 [30]), XML databases [25] and mediators [8,10,11,19,26] have been designed 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 [11,26], 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 [4,27] - 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 [37,38,42] 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 (QSS), 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.
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 constructs an *Expandable Schema Tree (EST)* out of it. The EST is a structure that serves as the basis for building the query set specification and is a visual abstraction of the XML Schema that the developer interacts with. The Editor also inputs an *XHTML query form page* that provides the (XHTML) static part of the form page, including the HTML form controls [44], such as select ("drop-down menus") and text ("fill-in-the-box") input controls, that the end-user will be interacting with. It may additionally input an optional *template report page* that provides the XHTML structure 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 [39]. If a template report page is not provided, the developer can build one using the Editor.

![Figure 1 QURSED System Architecture](image)

The Editor displays the EST and the XHTML pages to the developer, who uses them to 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, selections and joins) that contain parameters. The query/visual association indicates how each parameter is associated with corresponding HTML form controls [44] 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 submitted and which visual fragments are eligible to appear on the query form page at each point (see Figure 12 in Section 6.1). 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, which control the interaction with the end-user. Dynamic server pages are implemented in QURSED as Java Server Pages [36], while Active Server Pages [41] is another possible option. 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 between the visual fragments 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”, which is sent to the XML Data Server and its XHTML result is displayed on the 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 discusses how a TQL query is formulated from a QSS during run-time and Section 7 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 [37], ColdFusion [38], and Microsoft Visual Interdev [42]. 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 [23], 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 [28] and Microsoft’s Query Builder (part of Visual InterDev [42]), which target the querying of relational databases, and EquiX [9], BBQ [21], VQBD [7], the Lorel’s DataGuide-driven GUI [17], and PESTO [5], which target the querying of XML or object-oriented databases.

3. **Data-Intensive Web Site Generators**, such as Autoweb [14], Araneus [2] and Strudel [12]. 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 [13].

**Web-based Form and Report Generators** create web-based interfaces that access relational databases. Popular examples are Macromedia Dreamweaver UltraDev [37], ColdFusion [38], and Microsoft Visual InterDev [42]. 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 [44] 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 [36], Microsoft’s Active Server Pages [40] and Macromedia ColdFusion Markup Language [38].

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) [28]
interface and Microsoft’s Query Builder [42], which target the querying of relational databases. Recent tools such as EquiX [9], BBQ [21], VQBD [7], the Lorel’s DataGuide-driven GUI [17], and PESTO [5] 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. EquiX and BBQ use some form of the EST and of the corresponding visual concept, but they still require basic knowledge of query language primitives. 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.

It is important to note that the described visual query formulation tools and the Editor have very different goals: The goal of the former is the development of a query or a query template by a database programmer, who is familiar with database models and languages. 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 [14], Araneus [3] and Strudel [12] 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.

There is also the prior work of the authors on the XQForms system that declaratively generates Web-based query forms and reports that construct XQuery expressions [22]. 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 end-user actions.

Finally, there is the emerging XForms W3C standard [32], 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 end-user to check whether the sensor fits within an envelope with length $X$ and width $Y$, where $X$ and $Y$ are end-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 end-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 [43]. 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. The elimination of internal model mismatches yields significant advantages in
   the engineering and maintainability of the system.

3 **EXAMPLE**

   This section describes an example XML Schema, the corresponding EST 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 Data Set, XML Schema and Expandable Schema Tree**

   QURSED models XML data as labeled ordered tree objects (*lotos*), such as the sample data set shown in Figure
   2a that describes two proximity sensor products. 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. If \( n \) is a node
   of a *loto*, we denote as \( \text{tree}(n) \) the subtree rooted at \( n \).

   In the sample data set of Figure 2a, the top *sensors* node contains a *manufacturer* node, whose name is
   “Turck”. This manufacturer contains a list of two *product* nodes, whose direct subelements contain the basic
   information of each sensor. The first sensor’s *part_number* is “A123” and has an *image*, while the second’s one
   is “B123” and has no image. The technical specification of each sensor is modeled by the *specs* node, whose
   content is quite irregular. For example, the *body_type* of the first sensor is *cylindrical*, and has *diameter
   and barrel_style*, while the second one is *rectangular* and has *height and width*. Also, both sensors
   have more than one *protection_rating* nodes and have *min* and *max* operating temperature.
Figure 2 Example Data Set, XML Schema and Expandable Schema Tree

The XML Schema that describes the structure of the sample data set of Figure 2a is shown as a tree structure in Figure 2b. Similar conventions for representing XML Schemas and DTDs have been used by previous works, e.g. [2] and [11]. Indicated are the optional (? and * labeled edges) and repeatable (* and + labeled edges) elements and the types of groups of elements (SEQ, CHOICE and ALL nodes.) The leaf nodes are of primitive type [29]. Like many XML Schemas, it has nesting and many “irregular” structures such as choice groups, e.g. the body_type may be rectangular or cylindrical, and optional elements [34], e.g. each sensor can optionally have an image element.

Based on the XML Schema in Figure 2b, the Editor constructs the corresponding EST that serves as the basis for building the query set specification. Figure 2c shows the EST as it is displayed to the developer. The EST is a structure that supports two operations. First, multiple copies of repeatable elements can be created, as in the case of
the `protection_rating` element in Figure 2c. A typical reason for doing so is the need to apply different conditions on each copy. Second, nodes of the `EST` can be set to `true` or `false`. This is done via the Editor by checking or unchecking the checkboxes that appear next to the element nodes of the `EST` in Figure 2c. The reason for doing that is to indicate to the Editor which elements to include on the report. Report generation is described in Section 7.3. The `EST` is presented in more detail in Section 7.1.

### 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](image-url)
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 and customize the report page. 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.
- Selected from the “Sort By Options” list to sort the results first by “Manufacturer” (descending) and then by “Sensing Distance” (ascending). The selections appear in the “Sort By Selections” list.
- In the “Customize Presentation” section, selected to present (“P” column) all columns that she has control over, e.g., “Part Number” is, by default, always presented (disabled checkbox).

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. By default, QURSED organizes the presentation of the qualifying XML elements in a way that corresponds to the nesting suggested by their XML Schema. Notice, for example, that each product display has nested tables for rectangular and cylindrical values. Also notice that instead of the text of the manufacturer’s name, a corresponding image (logo) is presented.

The following section illustrates the query model QURSED uses to represent the possible queries. Section 7 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)

End-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 [31], XML-GL [6], LOREL [24], XMAS [19] and XQuery [30]. 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.
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 or a name variable, and an element variable $var(n)$. In a condition tree, there can be multiple nodes with the same constant element name, but element and name variables must be unique. Element variables start with the $\$ symbol and name variables start with the $\$N_$. 

- AND nodes, which are labeled with a Boolean expression $b$ consisting of predicates combined with the Boolean connectives $\land$, $\lor$ and $\neg$. 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 at least one 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.

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.

Intuitively, OR-removal is analogous to turning a logical expression to disjunctive normal form [16]. 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

![Figure 5 Conjunctive Condition Trees](image)

Intuitively, OR-removal is analogous to turning a logical expression to disjunctive normal form [16]. 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](image)

A conjunctive condition tree \( C \) produces all bindings for which an input \( loto t \) “satisfies” \( C \). Formally, a binding is a mapping \( \beta \) 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 an AND node) matches with the root of \( t \), i.e., \( \beta(var(child(root(C)))) = root(t) \), 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 \( C \).¹

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 or value \( v_i \), where \( 1 \leq i \leq k \). A binding of some of the variables in a (conjunctive) condition tree is called a partial binding. Note that the semantics of a binding requires total tuple assignment [24], 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 shown in Figure 5 on the source loto of Figure 2a 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 rows are subsumed by CCT1 rows in Table 1.) The necessary duplicate elimination is performed during construction. Notice that three of the four conjunctive condition trees generate two bindings 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.

### Table 1 Bindings for Conjunctive Condition Trees of Figure 5

<table>
<thead>
<tr>
<th>$NAME$</th>
<th>$PROD$</th>
<th>$PART$</th>
<th>$IMG$</th>
<th>$DIST$</th>
<th>$N_BODY$</th>
<th>$CYL$</th>
<th>$DIA$</th>
<th>$BAR$</th>
<th>$PROT$</th>
<th>$PROT1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turck</td>
<td>product</td>
<td>A123</td>
<td>A123.jpg</td>
<td>11</td>
<td>cylindrical</td>
<td>17</td>
<td>Smooth</td>
<td></td>
<td></td>
<td>NEMA1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NEMA3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NEMA3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NEMA3</td>
</tr>
</tbody>
</table>

The above shows that 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 necessary for queries over semistructured data sources (e.g., sources whose XML Schema makes use of choice groups and optional elements.)

The condition tree corresponds intuitively to the WHERE part of XML query languages such as XML-QL [31], LOREL [24] and XMAS [19], to the extract and match parts of XML-QL [6], and to the FOR and WHERE clauses of a FLWR expression of the upcoming XQuery standard [30]. 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:

---

1 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 \( b_1, \ldots, b_m \) and \( b \) be their boolean expressions. We can eliminate \( a_1, \ldots, a_n \) from the tree, and replace \( b \) with \( b \land b_1 \land \ldots \land b_m \).
• Element nodes \( n \) having an element name \( name(n) \), which is a constant if \( n \) is an internal node, and a constant or a variable that appears in the condition tree of \( q \), if \( n \) is a leaf node.

• 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 \( [\$V_1, ..., \$V_n] \) from the condition tree of \( q \). A sort-by label \( S \) is a list of \( (\$V_1, O_i) \) pairs, where \( \$V_1 \) is a variable from the condition tree of \( q \), and \( O_i \) is the sorting order determined for \( \$V_1 \). \( O_i \) can take the values “DESC” for descending or “ASC” for ascending order. Each variable in the sort-by list of a node must appear in the group-by list of the same node. Empty group-by and sort-by labels are omitted from figures in the remainder of the paper.

• A Boolean expression \( b \) on each node consisting of predicates combined with the Boolean connectives \( \land \), \( \lor \) and \( \neg \). The predicates consist of arithmetic and comparison operators and functions that use element and name variables appearing in the condition tree of \( q \), and constant values as operands.

Every element or name variable must be in the scope of some group-by list or Boolean condition. Similar to logical quantification, the scope of a group-by list or a Boolean condition of a node is the subtree rooted at that node.

Figure 4b shows the result tree for the example of Figure 3. Group-by and sort-by labels are the TQL means of performing grouping and sorting. The intuition behind Boolean expressions on nodes is that they provide control on the construction of nodes in the result of a query: A node (and its subtree) is only added to the result of the query if there is at least one qualified binding of the variables in the condition for that node that renders it \( true \). Note also that the rows of the HTML tables that contain the static column names are omitted from the result tree for presentation clarity.

Given a TQL query with condition tree and result tree, the answer of the query on given input is constructed from the set of qualified bindings of the condition tree. In what follows, binding refers to qualified binding. The result is a \( loto \) constructed by structural recursion on the result tree as formally described below. The recursion uses partial bindings to instantiate the group-by variables and condition variables of element nodes.

Traversing the result tree top-down, for each subtree \( tree(n) \) rooted at element node \( n \) with group-by label \( [\$V_1, ..., \$V_k] \) and, without loss of generality, sort-by label \( [\$V_1, ..., \$V_m] \) (\( m \leq k \)), let \( \mu = [\$V_{A1}, \$V_{A2}, ..., \$V_{An}, \$V_{At}] \) be a partial binding that instantiates all the group-by and condition variables of the ancestors of \( n \), let the Boolean expressions of \( n \) and its ancestors be \( b \) and \( b_{A1}, ..., b_{At} \), and let the variables in these expressions that do not appear
among the \[$V_{A1}, \ldots, V_{An}, V_1, \ldots, V_m\]$ be \[$B_1, \ldots, B_j\] . Recursively replace the subtree \(tree(n)\) in place with a list of subtrees, one for each qualified 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 \(V_{A1}: v_{A1}, \ldots, V_{An}: v_{An}, V_1: v_1, \ldots, V_k: v_k\) with \(v_{A1}, \ldots, v_{An}, v_1, \ldots, v_k\), if and only if \(b, b_{A1}, \ldots, b_{Ah}\) all evaluate to \(true\) for some qualified binding \(\pi' = [V_{A1}: v_{A1}, \ldots, V_{An}: v_{An}, V_1: v_1, \ldots, V_k: v_k, B_1: b_1, \ldots, B_j: b_j]\) (otherwise the subtree is not included in the list of subtrees produced.) The list of instantiated subtrees is ordered according to the conditions in the sort-by label.

Figure 7 shows the resulting \textit{loto}\textsuperscript{2} 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 \texttt{tr} element node is created, and that, for each such binding, different subtrees rooted at the nested \texttt{table} element nodes are created, corresponding to different \(\pi\) bindings. Finally, out of the three Boolean expressions that label the \texttt{img} elements in Figure 4b, only the first one evaluates to \texttt{true}, for both sensors, based on the bindings of variable \$NAME in Table 1.

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 A. 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 [30] and [35].

\footnotesize
\begin{verbatim}
.html body table tr td img "Al123.jpg" td img "Al123.jpg" td "Al123" tr td Table tr td NEMA1 tr td "10" tr td "cylindrical" tr td Table tr td NEMA3 tr td "25" tr td "NEMA4" tr td "rectangular" tr td Table tr td NEMA3 tr td "30" tr td "Smooth"
.html body table tr td img "turck.gif" td img "turck.gif" td "B123" tr td Table tr td NEMA1 tr td "11" tr td "Smakelbox" tr td Table tr td NEMA3 tr td "25" tr td "NEMA4" tr td "rectangular" tr td Table tr td NEMA3 tr td "10" tr td "30"
\end{verbatim}

\textbf{Figure 7 Resulting \textit{loto} for Bindings of Table 1}
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 8. This section formally presents the query set specification that is the logical underpinning of QFRs, including the visual interfaces and interactions described in Section 7.

Definition 3 (Query Set Specification). A query set specification QSS is a 4-tuple \(<CTG, RTG, F, D>\), where:

- **CTG**, the condition tree generator, is a condition tree with three modifications:
  - AND nodes \(a_i\) can be labeled with a set of Boolean expressions \(B(a_i)\).
  - The same element or name variable can appear in more than one condition fragments.
  - Boolean expressions can use parameters (a.k.a. placeholders [18]) as operands of their predicates.
    Parameters are denoted by the \# symbol and must bind to a value [29].

The same constraints apply to a CTG as to a condition tree.

- **RTG**, the result tree generator, is a result tree with two modifications. First, the variables that appear in the sort-by label \(S\) on a node do not have a specified order (ascending or descending,) as in the case of a result tree, but they have a parameter instead, called ordering parameter that starts with the \# symbol. Second, the Boolean expressions on nodes can use parameters as operands of their predicates. Boolean expressions on nodes involving only parameters and constants as operands (no variables) are a special case since they can be evaluated as soon as the parameters are instantiated. Their use is described later in Section 7.5.

- **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 \(b\) must belong to a node included in \(f\). F always contains a special condition fragment \(fR\), called result fragment, that includes all the element nodes whose variables appear in RTG, 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. Dependencies are defined in Section 6.1.

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:

1. condition fragment $f_1$ is defined by the dark grey subtree and the Boolean expression on the root AND node of the $CTG$ that applies a condition to the name element node;

2. condition fragment $f_2$ is defined by the medium gray subtree and the Boolean expressions that apply a condition to the dimensions of cylindrical and rectangular sensors; and

3. condition fragment $f_R$ (the result fragment) is defined by the light grey subtree that includes all the element nodes whose variables appear in $RTG$ in Figure 8b, and imposes no Boolean conditions.

Figure 8 Query Set Specification

How the developer produces a query set specification via the Editor is described in Section 7.
6 QUERY FORMULATION PROCESS

Figure 9 summarizes the query formulation process of the QURSED run-time engine. The process starts by accepting a QSS \( \langle \text{CTG,RTG,F,D} \rangle \) (provided by the interaction of the developer with the Editor) and a partial valuation of its parameters (provided by the end-user’s interaction with the query form page). The process terminates by outputting an XQuery expression.

**Figure 9 Query Formulation Process**

*Parameter Instantiation.* The run-time engine first instantiates the parameters of the condition tree generator CTG and the result tree generator RTG. In particular, during the end-user’s interaction with the query form page, and based on which form controls she fills out, a partial valuation \( \nu \) over \( P \), where \( P \) is the set of the parameters that appear in the QSS, is generated. As an example partial valuation, consider the one generated by the query form page of Figure 3 from the constant values the end-user provides:

\[
\nu = \{ \$\#\text{PROT1} : \text{"NEMA3"}, \$\#\text{DIMX} : \text{"20"}, \$\#\text{DIMY} : \text{"40"}, \$\#\text{O_NAME} : \text{"DESC"}, \$\#\text{O_DIST} : \text{"ASC"} \}
\]

Based on \( \nu \), the run-time engine instantiates the parameters of condition fragments in \( F \). For example, the above partial valuation instantiates the parameters \$\#\text{DIMX} \) and \$\#\text{DIMY} \) of condition fragment \( f_2 \) of Figure 8a, which imposes a condition on the dimensions of the sensor’s body type. Similarly, the ordering parameters of the sort-by labels of RTG are instantiated. The ordering parameters can take the values “DESC” or “ASC”, as in the case of \$\#\text{O_NAME} \) and \$\#\text{O_DIST} \) in the above partial valuation. Finally, the run-time engine also instantiates the parameters of the set of dependencies D and the parameters of Boolean expressions labeling nodes of the RTG. Dependencies are presented in the next section and an example of an RTG, where parameterized Boolean expression are labeling its nodes, is shown in Section 7.5.

*FragmentActivate Algorithm.* As a second step on Figure 9, the FragmentActivate algorithm inputs the instantiated CTG and the set of condition fragments \( F \), and outputs the set of active condition fragments, which is also instantiated. The algorithm renders a condition fragment active if it has all its parameters instantiated by the
partial valuation $\nu$. Since the partial valuation $\nu$ might not provide values for all the parameters used in $CTG$, some condition fragments are rendered inactive. Based on the above example partial valuation, condition fragment $f_2$ of Figure 8a and the condition fragment that imposes a condition on protection rating (not indicated in Figure 8a) are rendered active, while condition fragment $f_1$ on manufacturer’s name is inactive, since parameter $\$\#NAME$ is not instantiated by $\nu$. As a special case, the result fragment $f_R$ is always active, since it doesn’t have any parameters.

Note that the FragmentActivate algorithm on Figure 9 also inputs the set of dependencies $D$, which further complicate the algorithm. Both the dependencies and the revised version of the FragmentActivate algorithm are presented in the next section.

**QSS2TQL Algorithm.** The set of active condition fragments and the instantiated $RTG$ are passed to the $QSS2TQL$ algorithm, which outputs a TQL query by formulating its condition tree $CT$ and its result tree $RT$. The $CT$ consists of the union of the nodes of the active condition fragments $f_1, \ldots, f_n$, along with the edges that connect them. Each AND node $\alpha$ in the $CT$ is annotated with the conjunction $c_1 \wedge \ldots \wedge c_n$ of the Boolean expressions $c_1, \ldots, c_n$ that annotate the node $\alpha$ in the fragments $f_1, \ldots, f_n$ respectively.

Similarly, in order to convert the $RTG$ to the $RT$, the $QSS2TQL$ algorithm first eliminates from the $RTG$ the subtrees rooted at nodes labeled with a Boolean expression $b$ that has un-instantiated parameters or evaluates to false, as further explained in Section 7.5. Then for every node that has a sort-by label $S$, we keep in the label only the variables with instantiated ordering parameters.

As an example of the $QSS2TQL$ algorithm, consider the $CT$ of Figure 4a, which is formulated based on the active condition fragments of Figure 8a, i.e., $f_2$, the condition fragment that imposes a condition on protection rating, and the result fragment $f_R$. Accordingly, the $RT$ of Figure 4b is formulated from the $RTG$ of Figure 8b, where the variable $\$\#N\_BODY$ is excluded from the top sort-by list, since its ordering parameter $\$\#O\_N\_BODY$ is not instantiated by the example partial valuation above.

The performance of the resulting TQL query benefits by having its $CT$ rewritten into a smaller equivalent $CT$, by applying the following two rewriting rules:

1. The pattern of Figure 10a happens when the $B$ element node (say operating_temp) is optional. In particular, the OR node guarantees that when the $B$ element node is present it will be bound to the $B$ variable while its absence will not disqualify the binding. However, if a condition fragment places a
condition on $B$ then the lower $B$ node of Figure 10a also appears. In this case, the OR node is redundant and can be removed, as Figure 10b shows, since the extra condition on $B$ makes its presence necessary.

2. OR nodes with a single AND node as child, shown on Figure 10c, are removed. If the removed AND node is labeled with a Boolean expression $b$, then $b$ is conjunctively connected to the Boolean expression $b'$ of closest ancestor AND node, as Figure 10d shows.

![Figure 10 “Remove Optional Copy” Rule](image)

**TQL2XQuery Algorithm.** The final step of the query formulation process on Figure 9 passes the TQL query as input to the TQL2XQuery algorithm, presented in Appendix A. The TQL2XQuery algorithm outputs the final XQuery expression, which is sent to the underlying XQuery processor.

### 6.1 Dependencies

Dependencies 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 declarative way to control the appearance of visual fragments.

**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 $\land$, $\lor$ and $\neg$. The predicates consist of arithmetic and comparison operators and functions that use parameters and constant values as operands. The set $H \subseteq F$, called the head of the dependency, contains the condition fragments that use at least one parameter that appears in $B$.

A dependency $d$ holds if each parameter $p_i$ in $B$ is instantiated in a condition fragment in $H$ that is active, and $B$ evaluates to true. In the presence of dependencies, a fragment $f$ is active if all its parameters are instantiated and at least one of the dependencies, where $f$ is the dependent condition fragment, holds. Intuitively, a set of dependencies constrains the set of queries a query set specification can generate by rendering inactive the dependent condition...
fragments when none of their dependencies hold. For example, consider the condition tree generator and condition
fragments of Figure 11a, and let us define two dependencies $d_1$ and $d_2$ as follows:

\[
\begin{align*}
&<f_1, \#\text{BODY} = \text{"cylindrical"}, \{f_1\}> \quad (d_1) \\
&<f_2, \#\text{BODY} = \text{"rectangular"}, \{f_2\}> \quad (d_2)
\end{align*}
\]

The condition fragment $f_1$ uses the parameter $\#\text{BODY}$ that appears in the condition of both dependencies on $f_2$
and $f_3$. If a value is not provided for $\#\text{BODY}$, 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.

![Condition Tree Generator and Dependencies Graph](image)

**Figure 11 Condition Tree Generator and Dependencies Graph**

Dependencies affect the appearance of a query form. 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
12 demonstrates the effect of dependencies $d_1$ and $d_2$ on the query form page of Figure 3. The two shown sets of
form controls are the visual fragments of the condition fragments shown in Figure 11a. For instance, the condition
fragment $f_1$ applies a condition to the element node labeled with $\$\text{BODY}$ 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 12a.) 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 in QURSED: its visual fragment is hidden, so there is no
way for the end-user to provide values for the parameters of the condition fragment.

![Dependencies on the Query Form Page](image)

**Figure 12 Dependencies on the Query Form Page**
Obviously, circular dependencies must be avoided, since the involved dependent fragments can never become active. This restriction is captured by the dependency graph:

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

The dependency graph for the dependencies $d_1$ and $d_2$ defined above is shown in Figure 11b. QURSED enforces that the dependency graph is *acyclic*.

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 algorithm for "resolving" the dependencies to decide which fragments are active, called *FragmentActivate*, is based on topological sort [20] (hence of complexity $\Theta(V+E)$) and is outlined below. If during the application of the algorithm a condition fragment node has all its parameters instantiated and has incoming edges labeled with conditions that are not evaluated yet, we will say that the node is *provisionally active*. In lines 3-4, the algorithm decides if they are rendered active. Note also that if a node has all its parameters instantiated and has no incoming edges, then it is automatically set as active.

---

**Algorithm FragmentActivate**

**Input:** A dependencies graph $G = <V, E>$, and a partial valuation $\nu$ over $P$, where $P$ is the set of the parameters that appear in the QSS, and each parameter $p_i$ that appears in $G$ and is a member of $P$ is instantiated with the constant value $\nu(p_i)$.

**Output:** The set $A$ of active condition fragments.

**Method:**

1. $A \leftarrow \emptyset$
2. Repeat
3. If there is an edge $(n, u)$ in $E$, where $n$ is active, $u$ is provisionally active and the condition on edge $(n, u)$ has all its parameters instantiated and evaluates to $true$
4. Add $u$ to $A$
5. Until no more nodes become active

---

Section 0 describes how the developer can define dependencies using the Editor.

### 7 QURSED EDITOR

The QURSED Editor is the interface the developer uses to build $QFR$s. The Editor takes as input an XML Schema and allows a set of visual actions that enable the development of a $QSS$. The Editor also takes as input two HTML pages, the query form page and the template report page. The query form page is connected with the
specification of the condition tree generator, the condition fragments and the dependencies, while the report page is connected 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 EST, namely CHOICE nodes. 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.

7.1 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 EST. Figure 13a shows the main window of the Editor, where the left panel presents the EST, described in the example of 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 [44]. The XHTML tree corresponds to the page shown on Figure 13b rendered in the Macromedia HomeSite [39] WYSIWYG HTML editor. In the future HomeSite’s extensible WYSIWYG editor can replace the Editor’s XHTML tree representation.

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 13a.

![Figure 13 Building a Condition Fragment](image)

**Figure 13 Building a Condition Fragment**

Converting the Condition Fragment. First, the developer initiates a condition fragment by clicking on the “New Condition Fragment” button (Action 1 of Figure 13a) and providing a unique ID, which is \( \text{manufacturer}_\text{name} \).
in this case. The middle panel lists the condition fragments defined so far, and the expression editor at the bottom allows their definition, inspection and revision. Then, the developer builds a Boolean expression in the expression editor, by drag ‘n’ dropping the equality predicate (Action 2) that has two, initially unspecified, operands. She sets the left operand of the equality to be an element node by dragging ‘n’ dropping the name element node from the EST (Action 3). The path from the root of the EST to the dragged element node appears in the left operand box and is also indicated by the highlighting of the name element node on the left panel.

*Query/Visual Association.* As a final step, the developer binds the right operand of the equality predicate to an HTML form control, which will provide the value for the operand at run-time. She performs the binding by dragging ‘n’ dropping the select element node named man_name_select from the query form page (Action 4). The name of the form control appears in the right operand box.

In the above actions the developer builds a Boolean expression that is translated by the Editor to the condition fragment $f_i$ of the condition tree generator of Figure 8a. In order to strictly define this translation we provide a definition of the EST.

**Definition 6 (Expandable Schema Tree).** An expandable schema tree EST is a labeled tree that consists of:

- Element nodes $n$ having an element name $name(n)$, which is a constant. Element nodes are labeled with an element variable $var(n)$ and an occurrence constraining $occ(n)$, which can be ? (0-1 occurrences), 1 (only one occurrence), * (any number of occurrences) or + (one or more occurrences). An element node $n$ is optional if $occ(n)$ is either ? or *. If $occ(n)$ is either + or *, then $n$ is repeatable. Element nodes have a property $report(n)$ that is true if the corresponding checkbox on the view of EST, like the ones shown in Figure 13a, is checked (This property is used in Section 7.3.1, where result tree generators are automatically constructed.)

- SEQ nodes.

- CHOICE nodes.

- ALL nodes.

The following constraints apply to ESTs:

1. Element variables in an EST are unique and are denoted by the $\$ symbol.

2. The root node of an EST is an element node.
An expansion operation can modify the EST. An expansion can be applied only on a repeatable element node $n$, i.e. $\text{occ}(n)$ is either + or *. When applied on a repeatable element node $n$, creates a copy $c$ of the subtree rooted at $n$ and sets it as the last sibling of $n$. Finally, it labels all the element nodes of $c$ with new and unique element variables.

Figure 13a illustrates the visual representation of the EST, where the protection_rating element node has been expanded. Figure 15a shows the corresponding EST (partially) according to the above definition. Based on the EST, the Editor carries out the following tasks on the condition tree generator.

**Constructing Condition Fragment.** The actions of Figure 13a generate condition fragment $f_i$ of Figure 8a from the elements nodes in the EST. More specifically, in Action 3, the selection of an EST element $e$ as an operand to a predicate adds the path from the EST root to $e$ to the corresponding fragment $f$, if it’s not already in $f$, ignoring SEQ, CHOICE and ALL nodes, and sets the variable that labels $e$ as the operand of the predicate. For example, the developer’s action to drag the name element node from the EST and drop it to the left operand of the equality predicate (Action 3) results in the construction of the path of the condition fragment $f_i$ of Figure 8a and sets the $\$\text{NAME}$ element variable as the left operand of the equality predicate. The actions of Figure 13a also label the root AND node with the Boolean expression that the developer builds. For example, the Boolean expression built in the above actions labels the root AND node of Figure 8a and is indicated as part of the condition fragment $f_i$. The construction of the condition tree generator is accomplished by constructing each condition fragment $f_i$, as if $f_i$ was the only fragment of the condition tree generator, and by merging $f_i$ with the condition tree generator. The algorithm that merges $f_i$ with the condition tree generator is called ConstructCTG and is given in detail in Section 7.1.3.

**Query/Visual Association.** Action 4 of Figure 13a 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 $f_i$ of Figure 8a is the “Manufacturer” form control shown in Figure 13b. Action 4 also results in a parameter being created and bound to an HTML form control. In our example, the parameter $\$\#\text{NAME}$ that appears in the Boolean expression of $f_i$ in Figure 8a is generated and is associated with the form control named $\text{man\_name\_select}$ on the query form page. Note that, even though the visual actions introduce variables and parameters in the condition tree generator, the developer does not need to be aware of their existence or semantics.
7.1.1 Aliasing and EST Expansion

There are cases where the developer needs to create “aliases” of element nodes. For example, assume that the developer wants to give the end-user the ability to specify two desirable protection ratings, out of the multiple that a single sensor might have. This case is depicted on the query form page of Figure 13b. To accomplish that, the developer expands the protection_rating element node, using the ExpandNode operation of EST, and creates two copies of it, as shown on Figure 13a. Then she builds two condition fragments, where each one is using a different copy of protection_rating element node. The condition tree generator in Figure 8a illustrates the effect of the two condition fragments, where the two copies of the protection_rating element node have two different and unique element variables, $\text{PROT1}$ and $\text{PROT2}$, which are used in the corresponding Boolean expressions on the root AND node. The parameters $#\text{PROT1}$ and $#\text{PROT2}$ are associated with the corresponding form controls on the query form page of Figure 13b.

7.1.2 Automatic Introduction of Structural Disjunction

The Editor provides a simple interface for the developer to construct a set of Boolean expressions. By default each Boolean expression is transformed by the Editor into disjunctive normal form and labels the root AND node of each condition fragment. However the semistructureness of the schema (CHOICE nodes and optional elements) may render the Boolean expression unsatisfiable. The Editor automatically and by employing heuristics constructs a meaningful satisfiable Boolean expression by introducing the minimal structural disjunction in the form of structural disjunction operators and assign to each term of the structural disjunction one or more conjuncts of the initial Boolean expression.

For example, consider the query form page of Figure 13b, where 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 EST of Figure 2a shows that sensors can be either cylindrical or rectangular, denoted by the CHOICE node 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 constructs a condition fragments by building the following Boolean expression:

\[(\text{DIA} \leq \#\text{DIMX} \land \text{DIA} \leq \#\text{DIMY}) \lor (\text{HEI} \leq \#\text{DIMX} \land \text{WID} \leq \#\text{DIMY})\]
The $DIA$, $HEI$ and $WID$ variables are labeling the diameter, height and width elements of the $EST$. The $#DIMX$ and $#DIMY$ parameters are associated with the “Dimension X” and “Dimension Y” form controls shown on the query form page of Figure 13b.

The Editor detects that the above Boolean expression is unsatisfiable, since no sensor has both diameter and height. The Editor resolves this problem by automatically transforming the $\lor$ Boolean connective of the above Boolean expression to an OR node, as the 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 node in the $EST$ of Figure 2a. 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: $(DIA <= #DIMX \land DIA <= #DIMY)$ and $(HEI <= #DIMX \land WID <= #DIMY)$. The transformation of a condition fragment is part of the ConstructCTG algorithm that is presented in the next section.

7.1.3 ConstructCTG Algorithm

The ConstructCTG algorithm runs once for each condition fragment $f$ and incrementally creates a condition tree generator by merging the prior condition tree generator with the input condition fragment $f$. The order that the condition fragments are passed to the algorithm doesn't matter.

The ConstructCTG algorithm inputs a condition fragment $f$, where the root AND node is labeled with the Boolean expression $b$ the developer built using the actions of Figure 13a. The algorithm also inputs a condition tree generator $CTG$ already constructed from a set of condition fragments $F$, and the $EST$ of the source. The algorithm achieves three goals. The first goal is to check if $f$ can produce satisfiable queries. This is accomplished by bringing $b$ to disjunctive normal form and identifying at least one satisfiable conjunct. If there isn’t one, then the algorithm terminates outputting an error and does not merge $f$ with the $CTG$. The second goal is to transform $f$ by employing heuristics, such that $f$ produces satisfiable queries given the Boolean expression $b$. Accordingly, this goal is accomplished by the ConstructCTG algorithm by introducing structural disjunction that transforms $f$ by replacing Boolean connectives $\lor$ in $b$ with OR nodes in $f$, as explained in the condition fragment transformation example in the previous section. The transformation is based on the disjunctions in the Boolean expression $b$ and on two types of elements in the $EST$: (i) CHOICE nodes, and (ii) optional elements (either ? or * occurrence constraint). Finally, the
third goal is to output the merged CTG, and it is served by the third step of the ConstructCTG algorithm that adds (merges) f to the input CTG.

The ConstructCTG algorithm assumes the existence of a function node($V_i$) that given a variable name $V_i$ in $b$ returns the node $n_i$ of the EST that the variable corresponds to, i.e., the node of the EST that the developer drag ‘n’ dropped. In the case of name variables, node($V_i$) returns the parent of the node that the developer drag ‘n’ dropped. It also assumes the existence of a function copy($n_i$) that given a node $n_i$ in the EST it returns the copy of it in $f$, if there exists one, or null, otherwise.

Algorithm ConstructCTG

Input: A condition fragment $f$ with a Boolean expression $b$ labeling the root AND node, a condition tree generator CTG, and an EST.

Output: The condition tree generator CTG where $f$ has been added, or an error if $f$ cannot produce satisfiable queries, otherwise.

Method:

Step 1: Satisfiability Check of $f$

1. Rewrite $b$ in disjunctive normal form such that $b = c_1 \lor c_2 \ldots \lor c_n$, where $c_i$ is a conjunction of predicates.
2. For each conjunct $c_i$, where $1 \leq i \leq n$, that uses variables $V_{i1}, V_{i2}, \ldots, V_{ik}$
3. If there exist two variables $V_{ix}, V_{iy}, 1 \leq x,y \leq k$, such that the lowest common ancestor of node($V_{ix}$) and node($V_{iy}$) in the EST is a CHOICE node
4. Mark conjunct $c_i$ as unsatisfiable
5. If all conjuncts are marked as unsatisfiable
6. Output an error indicating that $f$ cannot produce satisfiable queries

Step 2: Transformation of $f$

// Introduces OR nodes to $f$ based on CHOICE nodes in the EST

7. Mark all distinct variables in $b$ as not visited
8. For each two conjuncts $c_i$ and $c_j$ in $b$, $1 \leq i,j \leq n$ and $i \neq j$, and for each two variables $V_{ix}, V_{iy}$ used in $c_i$ and $c_j$, respectively, such that at least one of them is not visited
9. If both the paths from node($V_{ix}$) and node($V_{iy}$) to their lowest element node common ancestor $n_{ANSC}$ in the EST contain either a CHOICE node or an optional element, excluding $n_{ANSC}$
10. If copy($n_{ANSC}$) in $f$ does not have an OR child node
11. Apply Rule 1 of Figure 14
12. Else If copy($n_{ANSC}$) in $f$ has an OR child node $n_{OR}$ and the subtree tree$_a$ that contains node($V_{ix}$) is a child of an AND child node $n_{AND}$ of $n_{OR}$, and tree$_b$ that contains node($V_{iy}$) is a child of copy($n_{ANSC}$)
13. Apply Rule 2 of Figure 14
14. Mark $V_{ix}$ and $V_{iy}$ in $b$ as visited

// Label AND nodes with Boolean expressions

15. For each conjunction $c_i$, $1 \leq i \leq n$
16. In $f$, identify the lowest AND node $a_i$ that is the common ancestor of all the element nodes labeled with the variables used in $c_i$ and label it with Boolean expression $c_i$
17. If the AND node $n_{AND}$ is labeled with more than one conjunctions
18. Combine them with the $\lor$ Boolean connective

Step 3: Addition of $f$ to CTG

19. Remove from $f$ all children $n_i$ of the root AND node
20. Insert in CTG all nodes $n_i$ as children of its root AND node
21. Add the set of Boolean expressions $B_f$, labeling the root AND node of $f$, to the set of Boolean expressions $B$ labeling the root AND node of CTG
Figure 14 “OR Node Introduction” Rules

Figure 15 illustrates an example of the application of the ConstructCTG algorithm based on condition fragments defined on the EST of Figure 15a. Assume the developer has built two Boolean expressions $b_1$ and $b_2$, and the Editor has created the corresponding condition fragments $f_1$ and $f_2$, shown in Figure 15b and c respectively, where $b_1$ and $b_2$ are labeling the root AND nodes. $f_1$ asks for sensors either having diameter less than the parameter $\#DIA$ or a protection rating equal to the parameter $\#PROT$, while $f_2$ asks for sensors having either diameter less than the parameter $\#DIA$ or width less than the parameter $\#WID$ so that they fit in a given space. Both condition fragments pass the check of Step 1 of the ConstructCTG algorithm, since both conjuncts of $b_1$ and $b_2$ involve a single variable.

In Step 2, structural disjunction is introduced to both fragments, shown in Figure 15d and e, according to the rules of Figure 14. In $f_1$, element node diameter is under a CHOICE node in EST and element node protection_rating is optional. So an OR node is introduced under element node specs, i.e., their lowest common ancestor. Similarly, in $f_2$, element nodes diameter and width are both under a CHOICE node in EST, so an OR node is introduced under element node body_type.

Step 3 of the ConstructCTG algorithm just puts $f_1$ and $f_2$ together, thus constructing the merged CTG shown in Figure 18a.
### Minimization of CTG

As an extra step, the Editor minimizes the size of the merged CTG using a set of rewriting rules shown in Figure 16 and Figure 17. More specifically, the Editor first pulls up the OR nodes of the CTG by applying the rules of Figure 16 until the OR nodes become children of the root AND node. Then, by traversing the CTG top-down and left to right, the Editor recursively applies Rule 1, Rule 2 and Rule 3 of Figure 17 for every node.

#### Figure 16 “Pull Up OR Nodes” Rules

```plaintext
Rule 1
- replace node A with node A OR subtree_i AND subtree_j

Rule 2
- replace OR AND subtree_i AND subtree_j with OR AND subtree_i OR subtree_j

Rule 3
- replace OR AND subtree_i AND subtree_j OR subtree_k with OR AND subtree_i AND subtree_j AND subtree_k
```

#### Figure 17 “Push Down OR Nodes” Rules

For example, consider the CTG of Figure 18a constructed by the ConstructCTG algorithm in the previous section. The rules of Figure 17 merge common paths in CTG. For example, the path from the sensors element node to the diameter element node appears in both condition fragments of Figure 18a, and every element node along the path is labeled with the same variable in both fragments. These paths though cannot be merged by directly applying the rules of Figure 17, because in $f_1$ the OR node is located above the body_type element node, while in the $f_2$ the OR node is located below. By applying the “Pull Up OR Nodes” rules of Figure 16 first, then it is possible to merge the two paths. Figure 18b shows the CTG after the rules of Figure 16 have been applied, where all OR nodes are pulled up resulting in a single OR node that appears as the only child of the root AND node. The final form of the CTG, shown in Figure 18c, is accomplished by applying the rules of Figure 17. Notice that the two aforementioned paths have been merged.

In Rule 3 of Figure 17, the common prefix $p$, on the right, refers to prefixes of subtree_j and subtree_k, on the left, consisting only of element nodes labeled or named with the same variable, and can be either simple paths or trees. Subtrees subtree(s)_j' and subtree(s)_k', on the right, refer to subtrees of subtree_j and subtree_k.
respectively. Note that in order to apply Rule 3; the tree on the right side of Rule 3 should give the tree on the left side if the rules of Figure 16 are applied on it.

Figure 18 Size Minimization of the CTG

7.2 Building Dependencies

Figure 12 of Section 6.1 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 19 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 11a. 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 6.1 on the “cylindrical” condition fragment by performing a set of actions indicated on Figure 19 by the numbered arrows.
First, the developer initiates a dependency by clicking on the “New Dependency” button (Action 1 of Figure 19) and enters 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. She sets the dependent condition fragment by dragging ‘n’ dropping the “cylindrical” condition fragment to the expression editor’s “Activate” box (Action 2). Then, she builds the condition of the dependency by dragging ‘n’ dropping the equality predicate in the expression editor (Action 3). She specifies 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 (Action 4). Thus the left operand takes the value the end-user chooses for body type. Finally, she specifies the right operand of the equality predicate by dragging ‘n’ dropping a string constant and typing “cylindrical” (Action 5). Note that only constant values and parameters that bind to form elements can be used in the condition of the dependency, as Section 6.1 describes.

7.3 Building Result Trees Generators

The Editor provides two options for the developer to build the result tree generator $RTG$ 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 $EST$ she wants to present on the report page. Then, the Editor automatically builds a result tree generator that creates report pages presenting the source data in the form of HTML tables that are nested according to the nesting of the $EST$. If the developer wants to structure the report page in a
different way than the one the EST dictates, the Editor provides a second option, where the developer provides as input a template report page to guide the result tree generator construction. Both options are described next.

![Figure 20 Automatically Constructed Report Page](image)

### 7.3.1 Automatic Result Tree Generator Construction

The developer can automatically build a result tree generator based on the nesting of the EST. For example, Figure 20 shows a report page created from the result tree generator for the data set and the EST of Figure 2a. The creation of the result tree generator and the template report page is accomplished by performing the two actions that are indicated by the numbered arrows on the Editor’s window of Figure 21.

![Figure 21 Selecting Elements Nodes and Constructing Template Report Page](image)

First, the developer uses the checkboxes that appear next to the element nodes of the EST to select the ones she wants to present on the report page (Action 1 of Figure 21). The element nodes `name`, `part_number`, `image`, `sensing_distance`, `cylindrical`, `rectangular` and the third alias of `protection_rating` are selected. This action sets the `report()` property of the selected element nodes in the EST to `true` and constructs the
result fragment $f_R$ indicated in the condition tree generator of Figure 22a. The variables that will be used in the result tree generator are also indicated. Then, the developer clicks on the “Build Report” (Action 2) and the Editor automatically generates the template report page displayed on the right panel of Figure 21 as a tree of HTML element nodes. Figure 22c shows how a WYSIWYG HTML editor renders the template report page. The Editor translates the above actions into a QSS as follows.

In Action 2, the Editor automatically generates the result tree generator of Figure 22b that presents the element nodes selected in Action 1 using HTML table element nodes that are nested according to the nesting of the EST. For illustration purposes, each table element node in Figure 22b is annotated with the EST element node that it corresponds to. Notice, for example, that the “product” table is nested in the “manufacturer” table, as is the case in the EST. The table headers in Figure 22c are created from the name labels of the selected element nodes. In the tables, the Editor places the element variables of the element nodes selected in Action 1 as children of $\text{td}$ (table data cell) element nodes. For example, in the result tree generator of Figure 22b the element variable $\$NAME$ appears as the child of the $\text{td}$ element node of the “manufacturer” table.

We discuss next how each type of structure of the EST is managed.

Optional Element Nodes: When the developer includes an optional element node in the result, the corresponding result fragment will produce results whether this optional element is or is not present. Figure 22a demonstrates the effect of the visual action to select an optional element node to appear on the report page. For example, for the optional image element node selected in Figure 21, the Editor introduces an OR node to the result fragment with two AND nodes as children. One of them is labeled with $true$ and has no children. This tree will allow bindings for the sensors that don’t have an image element node, as in the case of sensor “B123” in Figure 20.

Repeatable Element Nodes: The Editor handles the repeatable element nodes in the EST by automatically generating corresponding table elements and group-by lists in the result tree generator. For example, the path from the root of the EST to the name element node that is selected in Action 1 contains the manufacturer repeatable element node, which results in the generation of the “manufacturer” table element node, shown in Figure 22b, and the group-by list of its $\text{tr}$ (table row) child element node. This group-by list will generate one table row for each binding of the $\$MAN$ element variable.
**CHOICE Nodes:** CHOICE nodes in the EST require the Editor to automatically generate OR nodes in the result fragment $f_R$. For example, the CHOICE node above the cylindrical and rectangular element nodes in the EST is translated to an OR node in the result fragment $f_R$ in order to generate the bindings for sensors of either body type, and 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.

![Condition Tree Generator](image1)

![Result Tree Generator](image2)

![Template Report Page](image3)

**Figure 22 Automatically Generated Result Fragment, Result Tree Generator and Template Report Page**

The complete algorithm, called AutoReport, for constructing the result fragment and the result tree generator, is presented below. The AutoReport algorithm inputs the EST, where some or all of the element nodes are selected for presentation on the report page, i.e., their $report()$ property is set to $true$, the result fragment $f_R$, and proceeds in two steps. The first step transforms the result fragment $f_R$ by introducing OR nodes the same way Step 2 of the ConstructCTG algorithm of Section 7.1.3 does. The only addition is the optional elements that can be introduced, as
in the case of the image element node described above. The second step automatically constructs the result tree generator.

The AutoReport algorithm assumes the existence of a function $node(\$V_i)$ that given a variable name $\$V_i$ in $f_R$ returns the node $n_i$ of the EST that the variable corresponds to. In the case of name variables, $node(\$V_i)$ returns the parent of the node(s) that the name variable corresponds to. It also assumes the existence of a function $copy(n_i)$ that given a node $n_i$ in the EST it returns the copy of it in $f_R$, if there exists one, or null, otherwise.

**Algorithm AutoReport**

**Input:** The EST where some or all of the nodes are selected for presentation on the report page, and the result fragment $f_R$.

**Output:** The transformed result fragment $f_R$ and the result tree generator $RTG$.

**Method:**

**Step 1: Transformation of $f_R$**

// Introduce OR nodes to $f_R$ based on CHOICE nodes in the EST
1. Mark all distinct variables in $f_R$ as not visited
2. For each two variables $\$V_i$ and $\$V_j$ in $f_R$, $i \neq j$, such that at least one of them is not visited
3. If both the paths from node($\$V_i$) and node($\$V_j$) to their lowest element node common ancestor $n_{ANSC}$ in the EST contain a CHOICE node, excluding $n_{ANSC}$
4. If $copy(n_{ANSC})$ in $f_R$ does not have an OR child node
   - Apply Rule 1 of Figure 14
5. Else if $copy(n_{ANSC})$ in $f_R$ has an OR child node $n_{OR}$ and the subtree $tree_i$ that contains node($\$V_i$) is a child of an AND child node $n_{AND}$ of $n_{OR}$ and $tree_j$ that contains node($\$V_j$) is a child of $copy(n_{ANSC})$
   - Apply Rule 2 of Figure 14
6. Mark $\$V_i$ and $\$V_j$ in $f_R$ as visited

// Introduce OR nodes based on optional elements in $f_R$
7. Traversing $f_R$ top-down and left to right, for a node $n_i$
8. If $n_i$ is labeled with a variable $\$V_i$ and $node(\$V_i)$ is optional, or $n_i$ is named with a variable $\$V_i$ and at least one child of $node(\$V_i)$ is optional
   - Insert an OR node $n_{OR}$ as a child of $parent(n_i)$
9. Insert two AND nodes, $n_{AND1}$ and $n_{AND2}$, labeled with the Boolean expression true as children of $n_{OR}$
10. Remove $n_i$ and insert it as a child of $n_{AND1}$

**Step 2: Construction of the result tree generator $RTG$**

11. Create a node $n_r$ named “html” and set it as the root of $RTG$
12. Create a node $n_b$ named “body” and add it as a child of $n_r$
13. Create a node $n_t$ named “table”
14. Create a node $n_{tr}$ named “tr” and add it as a child of $n_t$
15. Traversing the EST top-down and left to right, ignoring SEQ, CHOICE and ALL nodes, for an element node $n_i$
   - BuildTable($n_i$, $n_{tr}$)

**BuildTable ($n_i$, $n_{tr}$)**

16. If $n_i$ is either repeatable or $parent(n_i)$ is a CHOICE node
   - Create a node $n_{td}$ named “td” and add it as a child of $n_{tr}$
17. Create a node $n_{table}$ named “table”
18. Create a node named “tr” and set it as the current $n_{tr}$
19. If $parent(n_i)$ is a CHOICE node
   - Attach the Boolean expression $var(n_i)$ to $n_i$
20. If $n_i$ is repeatable
   - Add $var(n_i)$ to the group-by list of $n_{tr}$
If \( n_i \) is a selected element node

Create a node \( n_{th} \) named “th” and add it as a child of \( n_tr \).

Create a node named \( name(n_i) \) and add it as a child of \( n_{th} \).

If \( n_i \) is a leaf element node

Create a node named “td”, add it as a child of \( n_tr \), and set it as the current \( n_{td} \).

Create a node named \( var(n_i) \) and add it as a child of \( n_{td} \).

If \( var(n_i) \) is not in any group-by list of an ancestor node

Add \( var(n_i) \) to the group-by list of \( n_{td} \).

For every child element node \( n_c \) of \( n_i \)

BuildTable(\( n_c, n_{tr} \)).

The result fragment \( f_R \) that is transformed during Step 1 of the AutoReport algorithm is merged with the condition tree generator \( CTG \) of a QSS according to Step 3 of the Construct\( CTG \) algorithm of Section 7.1.3 and the rules of Figure 16 and Figure 17. The only difference is that the OR nodes that denote optional elements do not participate in the rules of Figure 16 and Figure 17, i.e., OR nodes that always have exactly two AND nodes as children and exactly one of them is labeled with the Boolean expression \( true \) and has no children, are not pulled up or pushed down. An explanatory example is presented next.

(a) Expandable Schema Tree (\( EST \))

(b) Initial Conditional Fragment \( f_1 \)

(c) Initial Conditional Fragment \( f_R \)

(d) Conditional Fragment \( f_R \) after Step 1

(e) \( CTG \) after “Push Down OR Nodes” Rules

Figure 23 Example of the AutoReport and Construct\( CTG \) Algorithms

Figure 23 shows an example of how the AutoReport algorithm transforms a result fragment \( f_R \) and, subsequently, how the Construct\( CTG \) algorithm merges \( f_R \) with a condition fragment \( f_1 \). Figure 23a shows the \( EST \) that partially corresponds to the visual representation of the \( EST \) of Figure 21. Figure 23b shows a condition fragment \( f_1 \) that asks for sensors having a protection rating equal to the parameter \( \#PROT \). Assuming that the developer has selected the \( part_number \) and \( protection_rating \) element nodes to be presented on the report page, i.e., their report() property is set to \( true \), then the initial \( f_R \) constructed by the Editor is shown in Figure 23c. After Step 1 of the
AutoReport algorithm, an OR node is introduced to $f_k$, as shown in Figure 23d, indicating that the protection\_rating element is optional. The ConstructCTG algorithm constructs the merged CTG of Figure 23e, where the OR node has neither been pulled up nor pushed down.

If the OR nodes indicating optional elements were participating in the rules of Figure 17, then the branches of the CTG in Figure 23e leading to protection\_rating element node would also be merged, according to Rule 1 of Figure 17. Such a CTG though would always treat protection\_rating as required, which is not the case if the condition fragment $f_1$ become inactive during run-time. In contrast, the merged CTG of Figure 23e will retain the protection\_rating element as optional in the case the condition fragment $f_1$ is not activated. On the other hand, if the condition fragment $f_1$ is activated, then the merged CTG of Figure 23e will produce a condition tree that treats the protection\_rating element node as required, according to the simplification rule of Figure 10 of the QSS2TQL algorithm of Section 6.

7.3.2 Result Tree Generator Construction Based On Template Report Page

The developer can create more sophisticated report pages and result tree generators 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 20, that followed the nesting pattern of the EST, where the product is nested in the manufacturer element node. To accomplish that, she constructs the template report page shown in Figure 24 and provides it to the Editor.

![Figure 24 Editing the Template Report Page](image)

On the right panel of Figure 25 the template report page is displayed. Using the EST panel and the template report page panel, the developer constructs the result tree generator of the query set specification of Figure 8. In particular, the structure of the result tree generator is the structure of the template report page. The rest of the result
tree generator (element variables, group-by and sort-by lists) is constructed by performing the actions that are indicated by the numbered arrows on the Editor’s window of Figure 25.

Figure 25 Performing Element and Group-By Mappings on the Template Report Page

First, 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 25 (Action 1). Depending on what mapping was created, one of Actions 2, 3, or 4 is performed.

In the case of element mapping, the developer drags element nodes from the EST and drops them to leaf nodes of the template report page (Action 2). This action places the variable labeling of naming the dragged element node in the result tree generator, and adds the path from the root of the EST to the dragged element node to the result fragment $f_R$. For example, by dragging the part_number element node and dropping it on the td element node in the template report page, the developer implicitly places the $\text{PART}$ variable in the result tree generator of Figure 8b.

In the case of group-by mapping, the developer drags element nodes from the EST and drops them to any nodes of the template report page (Action 3). 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 adds the $\text{PROD}$ element variable to the group-by list of the tr, as shown in Figure 8b. According to the semantics of Section 4, one tr element node will be generated for each binding of the $\text{PROD}$ element variable.

The case of sort-by mapping is the same as the group-by mapping, but the developer additionally specifies an optional order. For example, by dragging the sensing_distance element node and dropping it on the tr
element node of the outermost table the developer generates the sort-by list of that element, shown in Figure 8b. The Editor defines automatically a group-by mapping for each sort-by mapping. Note though that the developer did not specify a fixed order, ascending or descending, thus generating the ordering parameter $\#O\_DIST$. This choice allows the end-user to choose the order or exclude sensing distance from the sort-by list altogether.

Finally, the developer clicks on the “Build Sort By Options” (Action 5), and the Editor automatically generates and appends the XHTML representation of the “Sort by Options” and “Sort By Selections” drop-down lists to the query form page of Figure 3. The “Sort by Options” list contains the sort-by mappings defined in Action 4 for which a fixed order has not been specified. The “Sort By Selections” list is initially empty. During run-time, the end-user can select any item from the “Sort by Options”, select “ASC” or “DESC” order, and, using the “+” button, add it to the “Sort By Selections” list. When the end-user submits the query form, the corresponding ordering parameters are instantiated with the order the end-user selected, as explained in the QSS2TQL algorithm in Section 6.

An engineering benefit from the way the developer builds the result tree generator 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 defined².

Based on the above actions, the result fragment $f_k$ is defined as the set of variables used in the result tree generator that the developer manually constructs. The $f_k$ is constructed by Step 1 of the AutoReport algorithm of Section 7.3.1 and is merged with the condition tree generator of a QSS according to Step 3 of the ConstructCTG algorithm of Section 7.1.3 and the rules of Figure 16 and Figure 17.

### 7.4 Building Result Boolean Expressions

In Figure 3, the manufacturer’s column does not display his name as text, but a corresponding image (logo) is presented instead. This effect is accomplished by the three $\texttt{img}$ elements, corresponding to the three possible manufacturers, shown in the result tree generator $\texttt{RTG}$ of the $\texttt{QSS}$ in Figure 8 and the Boolean expressions that label them. These expressions are visually defined by the developer on the template report page and are translated by the Editor to Boolean expressions labeling nodes of the $\texttt{RTG}$.

---

² The mappings between the $\texttt{EST}$ element nodes and template report page element nodes are retained through the use of the name attribute of the HTML elements.
In order to build these Boolean expressions, the Editor provides to the developer a set of actions that is similar to the actions provided for the specification of dependencies as it is presented in Section 0. The setting of the Editor is the same with the one in Figure 19, except that the “Report” tab is selected in the middle panel and the “Template Report Page” tab is selected in the right panel. The developer builds the Boolean expressions by performing the same set of actions as the ones described in Section 0 with two differences:

- In Action 2, the developer selects a node from the template report page from the right panel, instead of a condition fragment, to the expression editor’s “Activate” box in Figure 19. The subtree rooted at the selected node will be included in the report if the Boolean expression defined in the expression editor evaluates to true during run-time.
- In Actions 4 and 5, the developer cannot only specify parameters and constants as operands of the predicates in the Boolean expression, but also any variable, by dragging any element node from the EST on the left panel.

The Boolean expressions that the developer defines on the template report page are listed in the “Boolean Expressions” table of the middle panel of Figure 25.

Note that the Boolean expressions containing variables are translated to XQuery conditional expressions [30], according to TQL2XQuery algorithm in Appendix A. For example, the three Boolean expressions that label the img elements in Figure 4b are translated to three conditional expressions, as the XQuery expression in Appendix A shows. If the Boolean expressions contain parameters only then they are evaluated during the formulation of the TQL query, as the QSS2TQL algorithm shows in Section 6. An example of Boolean expressions containing parameters is given in the next section.

### 7.5 Dynamic Projection Functionality

On the query form page of Figure 3, the “Customize Presentation” section allows the end-user to control which columns she wants to project on the report page by selecting the corresponding checkboxes in the “P” column. This dynamic projection functionality is provided through the use of Boolean expressions in the result tree generator RTG of a QSS. Figure 26 shows the RTG of the QSS of Figure 8, where Boolean expressions controlling the dynamic projection are labeling td (table data cell) element nodes and are indicated with gray shade. These Boolean expressions contain projection parameters that start with $\#P_-$ and correspond to the checkboxes of the “Customize
The above described process assumes that the developer manually constructs the “Customize Presentation” table of Figure 3. The Editor though has the ability to construct this table automatically as part of the automatic construction of the RTG described in Section 7.3.1. In this case, the “Customize Presentation” table is constructed according to the nesting of the EST just as the template report page is, and is structurally the same as the header row of the template report page. For example, observe that the “Customize Presentation” table on Figure 3 is structurally the same with the header row of the report page, the only difference being that it is displayed vertically.

More specifically, during Action 2 of Section 7.3.1, the Editor asks the developer if she wants to construct a “Customize Presentation” table. If so, the Editor constructs a table based on the element nodes selected during Action 1 of Section 7.3.1 and lets the developer specify which of them she wants the end-user to be able to include or exclude.
exclude on the report page. For example, on the “Customize Presentation” table on Figure 3, the end-user cannot
determine the projection of “Part Number” and “Sensing Distance”. As a final step, the Editor appends the XHTML
tree representation of the “Customize Presentation” table on the query form page and labels the corresponding td (table data cell) element nodes of the RTG with Boolean expressions that contain the presentation parameters.

8 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. An on-
line demonstration of the system is available at http://www.db.ucsd.edu/QURSED/.

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.

9 ACKNOWLEDGMENTS

Our thanks to Clifton McLellan, Spyros Magiatis, Panagiotis Reveliotis, and Angus Wong for their contribution
to the design and implementation of Web Application Builder, a preliminary version of QURSED, which is part of
Enosys Design Suite [33].
REFERENCES

http://www.w3.org/TR/xmlschema-2/

http://www.w3.org/TR/xquery/

http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/

http://www.w3.org/TR/xhtml-forms-req

[33] Enosys Design Suite  
http://www.enosysoftware.com/design.html

http://www.w3.org/TR/xmlschema-0/

http://www.w3.org/TR/query-semantics/


[37] Macromedia Dreamweaver UltraDev  
http://www.macromedia.com/software/ultradev/

[38] Macromedia ColdFusion  
http://www.macromedia.com/software/ultradev/special/coldfusion/

[39] Macromedia HomeSite  
http://www.macromedia.com/software/homesite/

http://www.w3.org/TR/xquery-operators/

[41] Microsoft ASP.NET  
http://www.asp.net/

[42] Microsoft Visual InterDev  
http://msdn.microsoft.com/vinterdev/

[43] Oracle XSQL Pages and the XSQL Servlet  
http://technet.oracle.com/tech/xml/xsql_servlet/htdocs/relnotes.htm

http://www.w3.org/TR/html4/
APPENDIX

A. TQL2XQuery Algorithm

The algorithm TQL2XQuery works on TQL queries, presented in Section 4. TQL2XQuery generates an XQuery expression equivalent to the input TQL query. The XQuery expressions generated by TQL2XQuery include GROUPBY expressions to efficiently perform the groupings. GROUPBY expressions are not part of the latest XQuery working draft [30], but the draft includes an issue regarding an explicit GROUPBY construct. Such a construct is presented in Appendix B. The choice of XQuery augmented with GROUPBY expressions has been made because of the importance of grouping operations for producing nested XML and XHTML output. Explicit GROUPBY expressions enable easier optimization of such grouping operations. As Appendix B shows, XQuery+GROUPBY expressions can always be translated to XQuery expressions, often of significantly increased complexity: their use results in cleaner query expressions and more opportunities for optimization, but does not affect the generality of the algorithm.

TQL2XQuery inputs a condition (sub)tree \( C_T \), rooted at \( n_{C_T} \), and a result (sub)tree \( R_T \), rooted at \( n_{R_T} \), of a TQL query. The algorithm outputs an XQuery expression using nested FWOR (FOR-WHERE-SORTBY-RETURN) expressions and element constructors, where FWOR expressions are always nested in the RETURN clause of their parents, and element constructor expressions only appear in RETURN clauses of FWOR expressions. An FWOR expression \( e \) defines a scope \( s_e \). It follows that scopes are nested. The set of variables in \( s_e \) is \( \text{var}(s_e) \). Every variable in the FOR clause of an FWOR expression \( e \) has a 1-1 correspondence to a variable \( \$V \) in \( R_T \) (and hence also in \( C_T \)).

As discussed in Section 4, \( \$V \) is associated with a node \( n \) in \( C_T \). We write \( n=\text{node}(\$V) \) and \( \$V=\text{var}(n) \). We also write \( \text{scope}(\$V)\text{=}\text{scope}(n)\text{=}s_e \) to denote the expression \( e \) that \( \$V \) is in the scope of. In the algorithm, we represent by \( S \) the current scope and by \( E \) the current FWOR expression. We also define allvars(\( S \)) and allnodes(\( S \)) to be the set of all the variables and their associated nodes in \( C_T \) that are in \( S \) or in any scope that \( S \) is nested in.

Initially, the algorithm is called with TQL2XQuery(\( N_{C_T}, N_{R_T}, \emptyset, \text{nil} \)). \( N_{R_T} \) is the root of \( R_T \) and \( N_{C_T} \) is the root of the \( C_T \) for the TQL query under translation. The initial scope is empty, as is the initial FWOR expression.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>TQL2XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td>( n_{C_T}, n_{R_T}, S, E )</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
<td>An XQuery expression equivalent to the input TQL query</td>
</tr>
<tr>
<td><strong>Method:</strong></td>
<td></td>
</tr>
</tbody>
</table>

48
Traverse $RT$ top-down and left-to-right. For an element node $r$ of $RT$:

1. Set $V \leftarrow$ variables in the group-by list $G_r$ of $r \cup$ variables in the attached Boolean expression $b_r$ of $r$
2. If $S$ is empty and $V$ is not empty then
   3. Create a new FWOR expression $e$, with FOR clause: “FOR var($n_{CT}$) IN document('source.xml')”
   4. Set $E \leftarrow e$, $S \leftarrow s_e$
5. If there exists a variable $S V_i$ in $V$ such that $S V_i$ does not belong to allvars($S$), and the path (in $CT$) from $node(S V_i)$ to $n_{CT}$ contains an OR node
   6. Create a new FWOR expression $e$, set $S \leftarrow s_e$
7. For every distinct variable $S V_i$ in $V$ and not in allvars($S$)
   8. GenerateVariable($S V_i$, $S$, $E$)
9. If the group-by list $G_r$ of $r$ is not empty
   10. Add to RETURN clause of $E$ the expression “GROUPBY $G_r$ AS”
11. If $r$ has an attached Boolean expression $b_r$
   12. Add to RETURN clause of $E$ the expression “IF $b_r$ THEN”
13. If $name(r)$ is a constant
   14. Add to RETURN clause of $E$ the expression <$name(r)$>
   15. For each child $c_i$ of $r$
   16. TQL2XQuery($n_{CT}$, $c_i$, $S$, $E$)
   17. Add to RETURN clause of $E$ the expression </$name(r)$>
18. Else if $name(r)$ is a variable // then the node is guaranteed to be a leaf node, see Definition 2 in Section 4
   19. Add to RETURN clause of $E$ the expression “($name(r)$)”
20. If the sort-by list $S_r$ of $r$ is not empty
21. Add to ORDER BY clause of $E$ the $S_r$ list

GenerateVariable($SV$, $S$, $E$)

22. $B \leftarrow \emptyset$
23. Find in $CT$ the lowest element node ancestor $a$ of $node(SV)$ such that, in $RT$, $var(a)$ is in allvars($S$)
24. Construct a relative path expression $pe$ initially consisting of $var(a)$
25. Walk down the tree path from $a$ to $node(SV)$. For a node $n$ of $CT$ on that path:
   26. If $n$ is an element node
   27. Add to FOR clause of $E$ the variable declaration “var($n$) IN $pe$”
   28. Construct a new relative path expression $pe$ initially consisting of $var(n)$
   29. If $n$ is an AND node with a Boolean expression $b_n$
   30. Add $b_n$ to $B$
31. If $n$ has a name variable
   32. Add to FOR clause of $E$ the variable declaration “var($n$) IN $pe$/name ( )”
33. For every Boolean expression $b_f$ in $B$
34. For every variable $SV_i$ used in $b_f$ and not in allvars($S$)
35. GenerateVariable($SV_i$, $S$, $E$)
36. Add to WHERE clause of $E$ the conjunction of the expressions in $B$

Intuitively, the algorithm traverses the result tree depth-first and produces a FWOR expression, nested in the RETURN clause of the enclosing FWOR expression, when it encounters a node with a group-by list (lines 1-6). The FOR clause of the FWOR expression declares the variables in the group-by list by traversing the condition tree (lines 22-35). When the algorithm traverses the condition tree, it keeps track of the Boolean expressions (lines 29-30) and declares any variables needed in the FOR clause (lines 33-35). It also conjunctively connects each Boolean

---

3 Refers to the XPath’s name () function [40]. Not to be confused with the name() used in Definition 1.
expression with the WHERE clause of the F沃R expression (line 36.) If the nodes of the result tree have an attached Boolean expression, then a condition expression is added to the RETURN clause of the F沃R expression (lines 11-12). Each node of the result tree either constructs an element or generates element content in the RETURN clause (lines 13-19). Finally, if a node in the result tree has a sort-by list, then an ORDER BY clause is added to the F沃R expression (lines 20-21.) The complexity of the TQL2XQuery algorithm is polynomial in the size of the input CT and RT.

The following XQuery expression is generated from the TQL2XQuery algorithm for the TQL query in Figure 4. Notice that the algorithm can be enhanced easily to add a name attribute to all constructed nodes (on line 14), with the value of the attribute being the complete path of the node. That would allow us, for example, to name the different <tr>, <td> and <table> elements.

```xml
<html>
<body>
<table>
  <tr>
    <td>
      FOR $root IN document('source.xml'),
          $SENS IN $root/sensors,
          $MANU IN $SENS/manufacturer,
          $PROD IN $MANU/product,
          $NAME IN $MANU/name,
          $SPEC IN $PROD/specs,
          $DIST IN $SPEC/sensing_distance,
          $PROTS IN $SPEC/protection_ratings,
          $PROT1 IN $PROTS/protection_rating,
          $PART IN $PROD/part_number,
          $BODY IN $SPEC/body_type,
          $N_BODY IN $BODY/name()
    WHERE $PROT1 = "NEMA3"
    ORDER BY $NAME DESCENDING, $DIST
    RETURN
    GROUPBY $PROD AS
      <tr>
        <td>
          FOR $IMG IN $PROD/image
          RETURN
          GROUPBY $IMG AS
            <img>{$IMG}</img>
        </td><td>
          IF ($NAME = "Turck") THEN <img>"turck.gif"</img>
          IF ($NAME = "Balluff") THEN <img>"balluff.gif"</img>
          IF ($NAME = "Baumer") THEN <img>"baumer.gif"</img>
        </td>
        <td>{</td>
        <td>
          GROUPBY $PART AS
            <td>{$PART}</td>
          </td>
        </td>
      </tr>
      </tr>
    </table>
</body>
</html>
```
The proposal extends the XQuery syntax with the following GroupBy expressions (productions below extend those in http://www.w3.org/TR/xquery/#section-XQuery-Grammar):

```
```

B. **GROUPBY Proposal**

The proposal extends the XQuery syntax with the following GroupBy expressions (productions below extend those in http://www.w3.org/TR/xquery/#section-XQuery-Grammar):
The rest of the grammar remains unchanged. A GroupBy expression returns an unordered collection. The example below refers to the "Use Case XMP" DTD and data (in http://www.w3.org/TR/xmlquery-use-cases).

**EXAMPLE** Grouping elements in the returned document. "For each author, return the number of book titles she published, as well as the list of those titles and their year of publication".

```
FOR $b IN document("http://www.bn.com")/bib/book,
    $a IN $b/author,
    $t IN $b/title,
    $y IN $b/@year
RETURN
    GROUPBY $a AS
    <result> $a,
        <number> count(distinct($t)) </number>,
    GROUPBY $t, $y AS
        <$t>,
        <$y> </year>
    </titleYear>
</result>
```

Notice how the same variable $t can be used both outside a GROUPBY and inside a GROUPBY. Outside the GROUPBY its value is a collection, inside the GROUPBY its value is a node. The same query can be expressed without GROUPBY as follows. Here we have to construct an intermediate collection only to apply 'distinct' to it and then to iterate over it:

```
LET $t = document("http://www.bn.com")/bib/book[author=$a]/title
RETURN
    <result> $a
        <number> count(distinct($t)) </number>
        FOR $Tup IN distinct(
            FOR $b IN document("http://www.bn.com")/bib/book[author=$a],
                $t IN $b/title,
                $y IN $b/@year
            RETURN <Tup> <t> $t </t> <y> $y </year> </Tup>),
            $t IN $Tup/t/node(),
            $y IN $Tup/y/node()
        RETURN <titleYear>
            <$t>,
            <$y> </year>
        </titleYear>
</result>
```