Graphical Query Interfaces for Semistructured Data: 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
Department of Informatics
Athens University of Econ. & Business
vassalos@aueb.gr

Abstract

We describe the QURSED system for the declarative specification and automatic generation of web-based query forms and reports (*QFR*s) for semistructured XML data. In QURSED, a *QFR* is formally described by its query set specification (*QSS*), which captures the complex query and reporting *capabilities* of the *QFR,* and the associations of the query set specification with visual elements that implement these capabilities on a web page. The design-time component of QURSED, called QURSED Editor, semi-automates the development of the query set specification and its association with visual elements by translating intuitive visual actions taken by a developer into appropriate specification *fragments*. The run-time component of QURSED produces XQuery statements by synthesizing fragments from the query set specification that have been activated during the interaction of the end-user with the *QFR*, and renders the query results in interactive reports, as specified by the *QSS*. We describe the techniques and algorithms employed by QURSED, with emphasis on how it accommodates the intricacies introduced by the semistructured nature of the underlying data. We present the formal model of the query set specification, as well as its generation via the QURSED Editor, and focus on the techniques and heuristics the Editor employs for translating visual designer input into meaningful specifications. We also present the algorithms QURSED employs for query generation and report generation. 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 is a simple and powerful data exchange and representation language, 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] that evolved into XML-based data models [53]. XML query languages (most notably the emerging XQuery standard [52]), XML databases [40] and mediators [8,13,16,25,41] 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 [16,41], XHTML 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 [5,42] - 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 [27,28,32] 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 whose focus is semistructured XML data.

QURSED produces query form and report pages that are called *QFRs*. A *QFR* is associated with a *Query Set Specification* (*QSS*). A *QSS* describes formally the complex query and reporting capabilities [44] of a *QFR*. These capabilities include the large number of queries that a form can generate to the underlying XML query processor and the different structure and content of the query result. The emitted queries are expressed in XQuery and the query results are expressed directly in XHTML that renders the report page.

1.1 System Overview and Architecture

We discuss next the QURSED system architecture, shown in 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*.
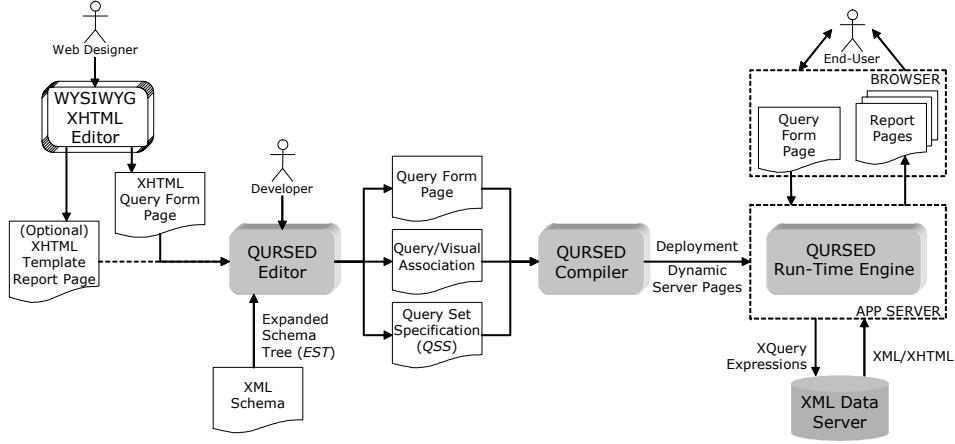

Figure 1 QURSED System Architecture

The Editor inputs the XML Schema [50] that describes the structure of the XML data to be queried and constructs an *Expanded Schema Tree* (*EST*) from 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 static part of the form page, including the XHTML form controls [46], 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 [29]. If a template report page is not provided, the developer can automatically build one using the Editor.

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 *QSS* focuses on the query capabilities 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*

3

Query Language (*TQL*) described in Section 4. The *QSS*'s key components are the parameterized *condition fragments*, the fragment *dependencies* and the *result tree generator* (*RTG*). 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 *XHTML form controls* [46] 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 11 in Section 6.1). Finally, the result tree generator 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 [22], while Active Server Pages [30] 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 primary function of QURSED is to generate and serve form interfaces and reports for semistructured data, and not to offer an API to an XML Data Server, which is a function better provided by the Data Server itself. QURSED of course can be used alongside applications that access the Data Server via such an API, being as it is simply another client connected to the XML Data Server.

Notice also that QURSED displays the XML result as delivered by the underlying XML Data Server. The form and structure of that result is mainly dependent on the result tree generator created by the *QFR* designer, as we will see in Section 6. Given the primary function of QURSED, the *RTG*s created by *QFR* designers generally lead to XQuery results structured as valid XHTML that has the form of a report. These results can be displayed on a browser, which is the intended use given the primary focus of QURSED, or can be fed into an application as XML input.

Finally, the QURSED infrastructure, in particular *QSS* and the QURSED run-time engine, can be used to describe and access powerful data-centric web services. Developing a web services infrastructure around *QSS* is the subject of future work, as is discussed in Section 8.

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 tool for the development of a *QFR* and its query set specification. Section 8 presents some conclusions and discusses future work.

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 [27], ColdFusion [28], and Microsoft Visual Interdev [32]. 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 [37], 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 [56] and Microsoft's Query Builder (part of Visual InterDev [32]), which target relational databases, and XML-GL [10], EquiX [9], BBQ [33], VQBD [7], the Lorel's DataGuide-driven GUI [21], and PESTO [6], which target XML or object-oriented databases.

3. *Schema Mapping Tools*, such as IBM's Clio [38], Microsoft's BizTalk Mapper [31], TIBCO's XML Transform [43] and Enosys's Query Builder [14]. These are graphical user interfaces that facilitate the data transformation from one or more source XML Schemas to a target XML Schema. The user constructs complex XQuery [52] or XSLT [51] expressions through a set of visual actions. These tools are mainly used in integration scenarios.

4. *Data-Intensive Web Site and Application Generators*, such as Autoweb [19], Araneus [4], Strudel [17] and Application Manifold [15]. 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 [18].

Web-based Form and Report Generators create web-based interfaces that access relational databases. The developer uses a set of wizards to visually explore the tables and views defined in a relational database schema and

5

selects the one(s) she wants to query using a query form page. By dragging 'n' dropping the attributes of the desired table to XHTML form controls [46] 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 XHTML 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 XHTML 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 XHTML 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 XHTML. The three most popular custom tag libraries today are Sun's Java Server Pages (JSPs) [22], Microsoft's Active Server Pages (ASPs) [30] and Macromedia ColdFusion Markup Language [28].

These tools are tightly coupled with the relational model and are able to generate automatically forms and reports when flat uniform relational tables need to be displayed. However, these tools cannot be used to generate semistructured query forms and reports; developing form and report pages that query and display semistructured data can be accomplished by writing custom JSPs (or ASPs) that address the semistructured aspects of the data. In contrast, we provide a system that generates such query forms and reports for semistructured data.

Visual Querying Interfaces are applications that allow the exploration of the schema and/or content of the underlying database and the formulation of queries. Unlike the form and report generators, which produce web front-ends for the "general public", visual querying interfaces present the schema of the underlying database to experienced users, who are often developers building a query, help them formulate queries visually, and display the result in a default fashion. The user has to, at the very least, understand what the meaning of "schema" is and what the model of the underlying object structure is, 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, based on a Document Type Definition (DTD) [45], a predecessor of XML Schema. EquiX and BBQ use some form of the Expanded Schema Tree (*EST*) and of the corresponding visual concept, but they still

6

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.

A more powerful visual query language is XML-GL that uniformly expresses XML documents, DTDs and queries as graphs. Queries consist of a set of extraction query graphs, a set of construction query graphs, and a set of bindings from nodes of one side to nodes of the other. In terms of expressiveness, XML-GL is more powerful than BBQ and EquiX, because of its ability to construct complex results using grouping, aggregate and arithmetic functions. It also supports heterogeneous union, in a fashion similar to TQL. XML-GL is less powerful than XQuery though, since recursive queries are not expressible and nested subqueries are partially supported.

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 by an average web developer of a form that represents and can generate a large number of possible queries.

Schema Mapping Tools are graphical user interfaces that declaratively transform data between XML Schemas in the context of integration applications. The transformation is a three-step process that is based on multiple source XML Schemas and a single target XML Schema that are visualized and presented to user. The first step discovers and creates correspondences between one or more elements of the source schemas and a single target element without attaching any specific semantics to them. The second step turns correspondences to mappings by specifying exactly how the source elements are transformed to the target element. Selection predicates, inner and outer joins, arithmetic, string and user defined functions are a few examples of the supported functionality. Clio [38] goes one step further and explains the difference between different mappings interactively by giving examples to the user based on small datasets. The third step of the transformation process generates either an XQuery [52] or an XSLT [51] expression that actually implements the transformation.

Note that the first two steps above are carried out using visual actions only, so the user does not need to be aware of the particular query language used by each tool. These visual actions greatly facilitate data integration by simplifying the transformation process, especially when someone takes into account that the generated query expressions are particularly complex and hard to write by hand.

7

QURSED's Editor adopts part of the functionality provided by the schema mapping tools for a different purpose. More specifically, the Editor creates two types of transformations without making a distinction between correspondences and mappings. First, it creates query/visual associations that map form controls on the XHTML query form page to parameters of selection predicates, in order to generate queries that filter the data. And second, it creates a transformation between a single XML Schema and an XHTML template report page in order to construct the report pages.

Data-Intensive Web Site and Application Generators. Autoweb [19], Araneus [4] and Strudel [17] 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 model for both data and web sites and 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.

Application Manifold [15] is the first attempt to expand a data integration framework to an application integration one. The system is capable of generating web-based e-commerce applications by integrating and customizing existing ones. Applications' flow is modeled and visually represented using UML State Charts that consist of states, corresponding to web pages that provide activities, linked by transitions, corresponding to navigation links that the end user can follow, and containing actions, corresponding to method calls that trigger other

transitions and/or alter the application's state. Application integration and customization is specified using a declarative language that allows for optimization and verification of the generated application.

Related to QURSED is also prior work on capability-description languages and their use in mediator systems [23,44]. The *QSS* formalism of QURSED is essentially a capability description language for query forms and reports over XML data. The prior work on capabilities has focused on describing the capabilities of query *processors* with an underlying relational data model. Instead the *QSS* captures the complex query and reporting capabilities of query *forms* over semistructured data.

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 [37]. The paper describes a software architecture that allows an extensible set of XHTML 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 [48], 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 that 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 sensors 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 XHTML 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 XHTML tables.

Notice that existing web-based form and report generators are tightly coupled to the relational data model, as explained in the previous section, and cannot be used to generate forms and reports on semistructured data.

Loose Coupling of Query and Visual Aspects. QURSED separates the logical aspects of query forms and reports generation, i.e., the query form capabilities, from the presentation aspects, 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 XHTML form controls, the condition fragments are associated with sets of XHTML 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 Description Language for Query Form Capabilities. We provide formal syntax and semantics for the *QFR* query set specifications, which describe query form capabilities by succinctly encoding 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 use 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 overall uniform system, when compared either against relational-based front-end generators or against conventional XML-based front-end architectures, such as Oracle's XSQL [34]. A representation-related uniformity 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   PRELIMINARIES

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 *QFR* interface. It concludes by describing the end-user experience with that interface.

3.1   Data Model, XML Schema and Expanded Schema Tree

QURSED models XML data as labeled ordered tree objects (*loto*s), 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. Empty elements are represented as nodes having a leaf node, labeled with the empty string, as child. If *n* is a node of a *loto*, we denote as *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` node and have `min` and `max` operating temperature.

(a) Data Set (*loto*)

(b) XML Schema

(c) Visual Representation of the *EST*

(d) Internal Representation of the *EST*
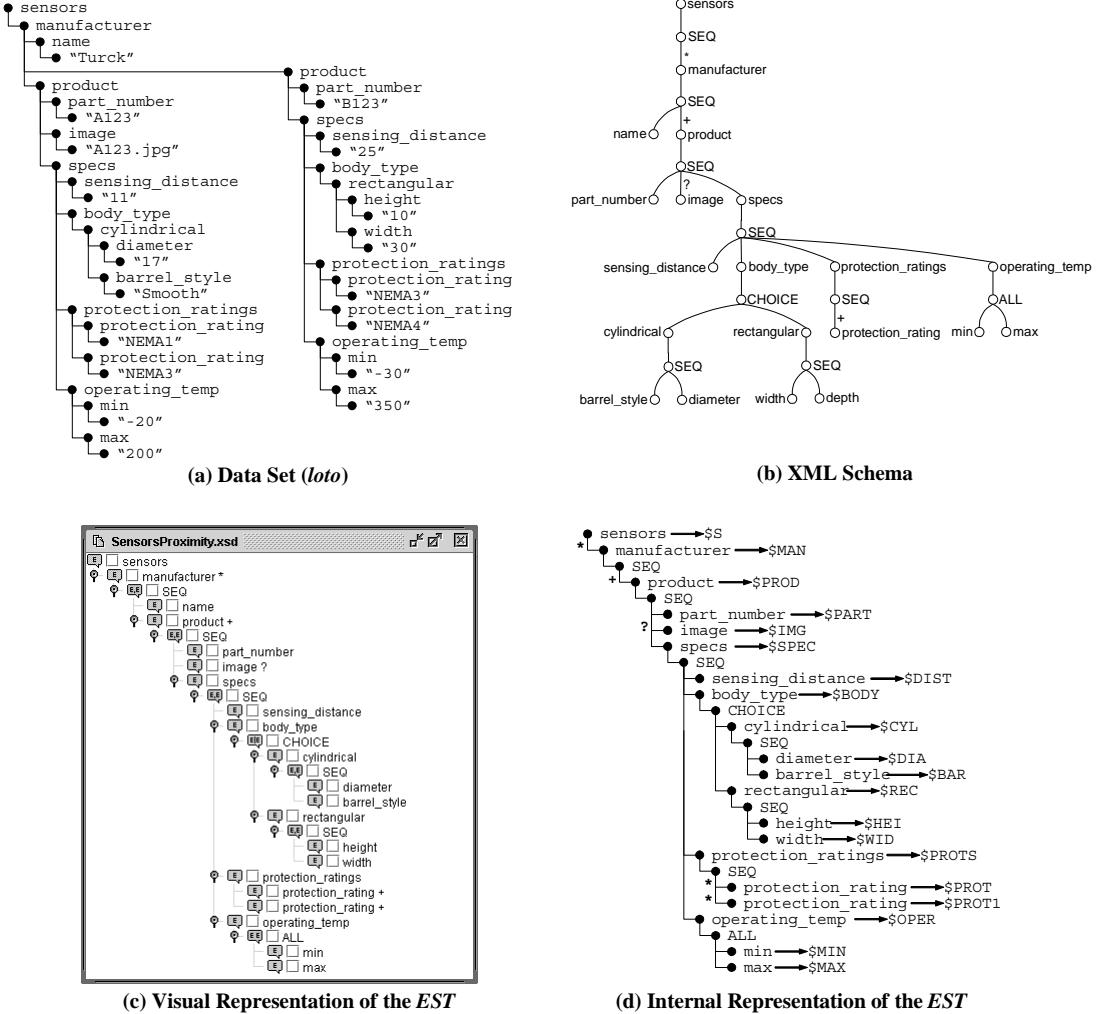
Figure 2 Example Data Set, XML Schema and Expanded Schema Tree

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 in previous work, e.g. [2] and [16]. 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 [49].) The leaf nodes are of primitive type [50]. 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 [49], 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 Editor's view of the *EST* as it is displayed to the developer, and Figure 2d the internal representation used by the Editor. Formally, the *EST* is defined in the following.

12

Definition 1 (Expanded Schema Tree). An Expanded 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 a unique element variable *var*(*n*), which starts with the $ symbol, and an occurrence constraint *occ*(*n*), which can be ? (0-1 occurrence), 1 (only one occurrences), * (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 Boolean property *report*(*n*).

- SEQ nodes that are labeled with an occurrence constraint *occ*(*n*), which can be ?, 1, * or +.

- CHOICE nodes that are labeled with an occurrence constraint *occ*(*n*), which can be ?, 1, * or +.

- ALL nodes that are labeled with an occurrence constraint *occ*(*n*), which can be ? or 1. ∎

The root node of an *EST* is a non-repeatable, non-optional element node.

An element node has a Boolean property *report*, which could be set to *true* by checking the corresponding checkbox that appears next to the element node on the view of the *EST* (Figure 2c). The reason for doing that is to indicate to the Editor which elements to include in the report. Report generation is described in Section 7.3.

Note that, even though element nodes are labeled with variables, the developer does not need to be aware of their existence, as the Editor's view of the *EST* in Figure 2c illustrates. Section 7 explains why the Editor doesn't need to expose the variables.

3.1.1 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 Figure 3, where two "Protection Rating" form controls appear on the query form page. To accomplish this, the developer expands the `protection_rating` element node on the *EST* and creates two copies of it, as shown on Figure 2c. The *EST* of Figure 2d illustrates the internal effect of the two aliases, where the two copies of the `protection_rating` element node have two different and unique element variables, `$PROT1` and `$PROT2`.

An expansion can be applied on a repeatable element node *n*, creating a copy *c* of the subtree rooted at *n* and setting it as the last child of *n*'s parent node. All element nodes of *c* are labeled with new and unique element variables. An expansion can also be applied on repeatable SEQ and CHOICE nodes.

3.2 Example *QFR* and End-User Experience

Using QURSED, a developer can easily generate a *QFR* interface like the one shown in Figure 3 that queries and reports on 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.

Query Form and Report Pages - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Sensors

General

Manufacturer: No preference / Balluff / Baumer / Turck

Sensing Distance [ ] mm

Protection Rating 1 [NEMA3]

Protection Rating 2 [No preference]

Operating Temperature [ ] to [ ] °C

Mechanical

Body Type [No preference]

Dimension X [20] mm

Dimension Y [40] mm

Report

Options

Results/page [10]

Sort By Options

[Body Type] [ASC]

Sort By Selections

DESC-Manufacturer
ASC-Sensing Distance

Customize Presentation

Column Name		P
Image		☑
Manufacturer		☑
Part Number		☑
Protection Ratings		☑
Sensing Distance		☑
Body Type ☑	Diameter	☑
	Barrel Style	☑
	Height	☑
	Width	☑

Reset   Execute

Image	Manufacturer	Part Number	Protection Ratings	Sensing Distance mm	Body Type	
	TURCK	BC 3-M12-AN6X	NEMA1 / NEMA3 / NEMA4	6.0	cylindrical — Diameter mm / Barrel Style	15 / Smooth
	TURCK	BC 3-M12-AP6X	NEMA3	6.0	cylindrical — Diameter mm / Barrel Style	19 / Smooth
	TURCK	BC 5-Q08-AN6X2	NEMA3 / NEMA4	7.0	rectangular — Height mm / Width mm	14 / 9
	TURCK	BC 5-Q08-AP6X2	NEMA3 / NEMA6 / NEMA11	7.5	rectangular — Height mm / Width mm	10 / 35
	TURCK	BC 5-S18-AN4X	NEMA3 / NEMA11	10.0	rectangular — Height mm / Width mm	15 / 10
	TURCK	BC 5-S18-AP4X	NEMA1 / NEMA3 / NEMA13	10.6	rectangular — Height mm / Width mm	2 / 10
	TURCK	BC 5-S18-Y0X	NEMA1 / NEMA3 / NEMA11	12.0	rectangular — Height mm / Width mm	17 / 23
	TURCK	BC 5-S185-AP4X	NEMA3	12.0	cylindrical — Diameter mm / Barrel Style	10 / Threaded
	TURCK	BC10-M30-AZ3X	NEMA3 / NEMA13	15.0	cylindrical — Diameter mm / Barrel Style	11 / Smooth
	TURCK	BC10-M30-RZ3X	NEMA1 / NEMA3 / NEMA6	25.0	cylindrical — Diameter mm / Barrel Style	20 / Threaded

Next 10   Previous 10

Figure 3 Example *QFR* Interface

The browser window displays a query form page on the left and a report page on the right. On the query form page XHTML 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. Some columns, like "Part Number", are always presented and can not be hidden by the end-user. Next to these columns appears a disabled and selected 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 `product` elements 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 [10], XML-GL [10], LOREL [39], XMAS [25] and XQuery [52]. TQL facilitates the development of query set specifications that encode large numbers of queries and the development of a visual interface for the easy construction of those specifications. This section describes the structure and semantics of TQL queries. The structure and semantics of query set specifications are described in the next section.

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

```
AND ──▶ PROT1 = "NEMA3"
sensors ──▶ $S
  manufacturer
    name ──▶ $NAME
    product ──▶ $PROD
    part_number ──▶ $PART
    OR
      AND ──▶ true
        image ──▶ $IMG
      AND ──▶ true
  specs
    sensing_distance ──▶ $DIST
    body_type
      $N_BODY
      OR
        AND ──▶ $DIA <= 20 AND $DIA <= 40
          cylindrical ──▶ $CYL
          diameter ──▶ $DIA
          barrel_style ──▶ $BAR
        AND ──▶ $HEI <= 20 AND $WID <= 40
          rectangular ──▶ $REC
          height ──▶ $HEI
          width ──▶ $WID
  protection_ratings
    protection_rating ──▶ $PROT
    protection_rating ──▶ $PROT1
```

(a) Condition Tree

```
html
body
  table ──▶ GROUPBY ($PROD, $NAME, $DIST)
    tr ──▶ SORTBY ($NAME DESC, $DIST)
      td
        img ──▶ GROUPBY ($IMG)
        $IMG
      td
        img ──▶ $NAME = "Turck"
        "turck.gif"
        img ──▶ $NAME = "Balluff"
        "balluff.gif"
        img ──▶ $NAME = "Baumer"
        "baumer.gif"
      td ──▶ GROUPBY ($PART)
        $PART
      td
        table
          tr ──▶ GROUPBY ($PROT)
            td
              $PROT
      td
        $DIST
      td
        table
          tr
            td ──▶ GROUPBY ($N_BODY)
              $N_BODY
          tr
            td
              table ──▶ GROUPBY ($CYL)
                tr
                  td ──▶ GROUPBY ($DIA)
                    $DIA
                  td ──▶ GROUPBY ($BAR)
                    $BAR
              table ──▶ GROUPBY ($REC)
                tr
                  td ──▶ GROUPBY ($HEI)
                    $HEI
                  td ──▶ GROUPBY ($WID)
                    $WID
```

(b) Result Tree

Figure 4 TQL Query Corresponding to Figure 3

Definition 2 (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 ∧, ∨ and ¬. 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* (*CCTs*) 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. The OR-removal is used to define the semantics of condition trees. It is not part of the QURSED implementation.
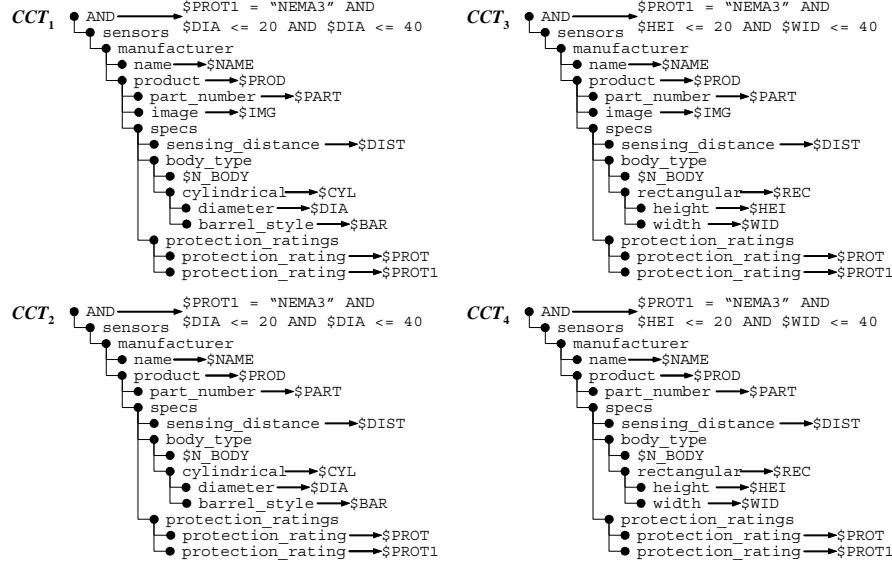

**Figure 5 Conjunctive Condition Trees**
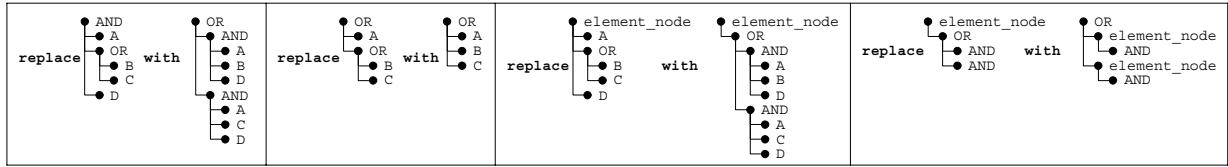
Figure 6 OR-Removal Replacement Rules

Intuitively, OR-removal is analogous to turning a logical expression into disjunctive normal form [20]. In particular, we repeatedly apply the rules shown in Figure 6. Without loss of generality, the subtrees of Figure 6 are presented with 2 or 3 children. 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, as per the earlier definition of condition trees. Finally, notice that in the course of OR-removal "intermediate results" may not be valid condition trees as per Definition 2 (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.

A conjunctive condition tree C produces all bindings for which an input *loto t* "satisfies" C. Formally, a binding is a mapping β from the set of element variables and name variables in C to the nodes and node labels of t, such that the child of the root of C (which is an AND node) matches 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 nodes in C are ignored in the traversal of C. In particular, in the definition above, by "child of the element", we mean either the 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. Notice that it is easy to do AND-removal on conjunctive condition trees. Let $a_1,...,a_n$ be the AND nodes in a conjunctive condition tree with root a, and let $b_1,...b_n$, and b be their Boolean expressions respectively. We can eliminate $a_1,...a_n$, and replace b with b AND b_1 and...and b_n.

The result of C is the set of qualified bindings. For a conjunctive condition tree with element and name variables $\$V_1,...,\$V_k$, a binding is represented as a tuple $[\$V_1:v_1,...,\$V_k:v_k]$ that binds $\$V_i$ to node or value v_i, where $1 \le i \le k$. The tuple variables are ordered according to a top-down, preorder traversal of the conjunctive condition tree. 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 [39], 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 into 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., CCT_2 rows are

18

subsumed by CCT_1 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

$NAME	$PROD	$PART	$IMG	$DIST	$N_BODY	$CYL	$DIA	$BAR	$REC	$HEI	$WID	$PROT	$PROT1	
Turck	product / part_number : "A123"	A123	A123.jpg	11	cylindrical	cylindrical / diameter : "17"	17	Smooth				NEMA1	NEMA3	CCT_1
Turck	product / part_number : "A123"	A123	A123.jpg	11	cylindrical	cylindrical / diameter : "17"	17	Smooth				NEMA3	NEMA3	CCT_1
Turck	product / part_number : "A123"	A123		11	cylindrical	cylindrical / diameter : "17"	17	Smooth				NEMA1	NEMA3	CCT_2
Turck	product / part_number : "A123"	A123		11	cylindrical	cylindrical / diameter : "17"	17	Smooth				NEMA3	NEMA3	CCT_2
Turck	product / part_number : "B123"	B123		25	rectangular				rectangular / height : "10"	10	30	NEMA3	NEMA3	CCT_4
Turck	product / part_number : "B123"	B123		25	rectangular				rectangular / height : "10"	10	30	NEMA4	NEMA3	CCT_4

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.)

Definition 3 (Result Tree). A result tree of a TQL query q is a node-labeled tree that consists of:

- Element nodes n having an element name $name(n)$, which is 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 list g and a sort-by list s on each node. A group-by list g is a (possibly empty) list of variables $[\$V_1,\ldots,\$V_n]$ from the condition tree of q. A sort-by list s is a list of $(\$V_i, O_i)$ pairs, where $\$V_i$ is a variable from the condition tree of q, and O_i is the sorting order determined for $\$V_i$. O_i can take the values "DESC" for descending or "ASC" for ascending order. If O_i is missing, then "ASC" is taken as the default value. 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 lists 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 $\wedge$, $\vee$ 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. Note that the rows of the XHTML tables that contain the static column names are omitted from the result tree for presentation clarity. Group-by and sort-by lists 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*. For example, in result tree of Figure 4b the "turck.gif" img node is added to the result of a query if a qualified binding of the \$NAME variable is equal to "Turck".

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 list $[\$V_1,\ldots,\$V_k]$ and, without loss of generality, sort-by list $[\$V_1,\ldots,\$V_m]$ ($m \le k$), let $\mu=[\$V_{A1}:v_{A1},\ldots,\$V_{An}:v_{An}]$ be a partial binding that instantiates all the group-by and condition variables of the ancestors of *n*, let the Boolean expressions of *n* and its ancestors be *b* and $b_{A1},\ldots,b_{Ah}$, and let the variables in these expressions that do not appear among the $[\$V_{A1},\ldots,\$V_{An}, \$V_1,\ldots,\$V_k]$ 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 $\$V_{A1},\ldots,\$V_{An},\$V_1,\ldots,\$V_k$ 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 list.

Figure 7 shows the resulting *loto* from the TQL query of Figure 4 and the bindings of Table 1. Note, for example, that for each of the two distinct partial bindings of the triple [\$PROD, \$NAME, \$DIST], one tr element node is created, and that, for each such binding, different subtrees rooted at the nested table element nodes are created, corresponding to different π bindings. Finally, out of the three Boolean expressions that label the img elements in Figure 4b, only the first one evaluates to *true*, for both sensors, based on the bindings of variable \$NAME in Table 1.
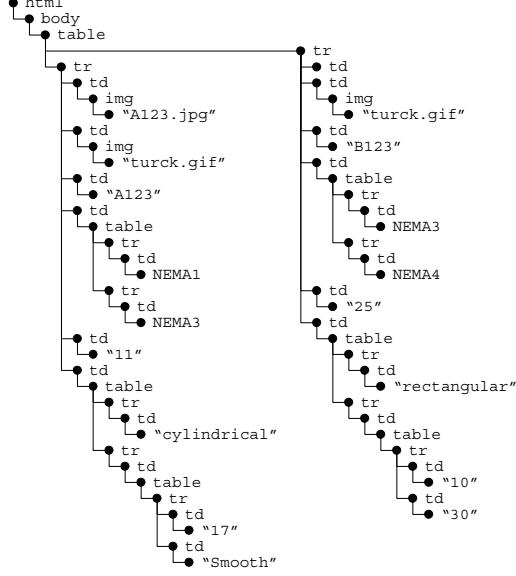
Figure 7 Resulting *loto* for Bindings of Table 1

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 [52] and [55].

In the absence of structural disjunction, the condition tree of a TQL query corresponds to the FOR and WHERE clauses of an XQuery expression that involve path expressions, conjunction and the SOME construct. Whether a path appears in the FOR clause or the WHERE clause depends on whether the corresponding variable appears in a group-by list in the result tree or not. Appendix A describes in detail the conversion of a TQL query to an XQuery expression and illustrates the above points.

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 *QFR*s large numbers of possible queries. In general, the *QSS* 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 *QFR*s.

21
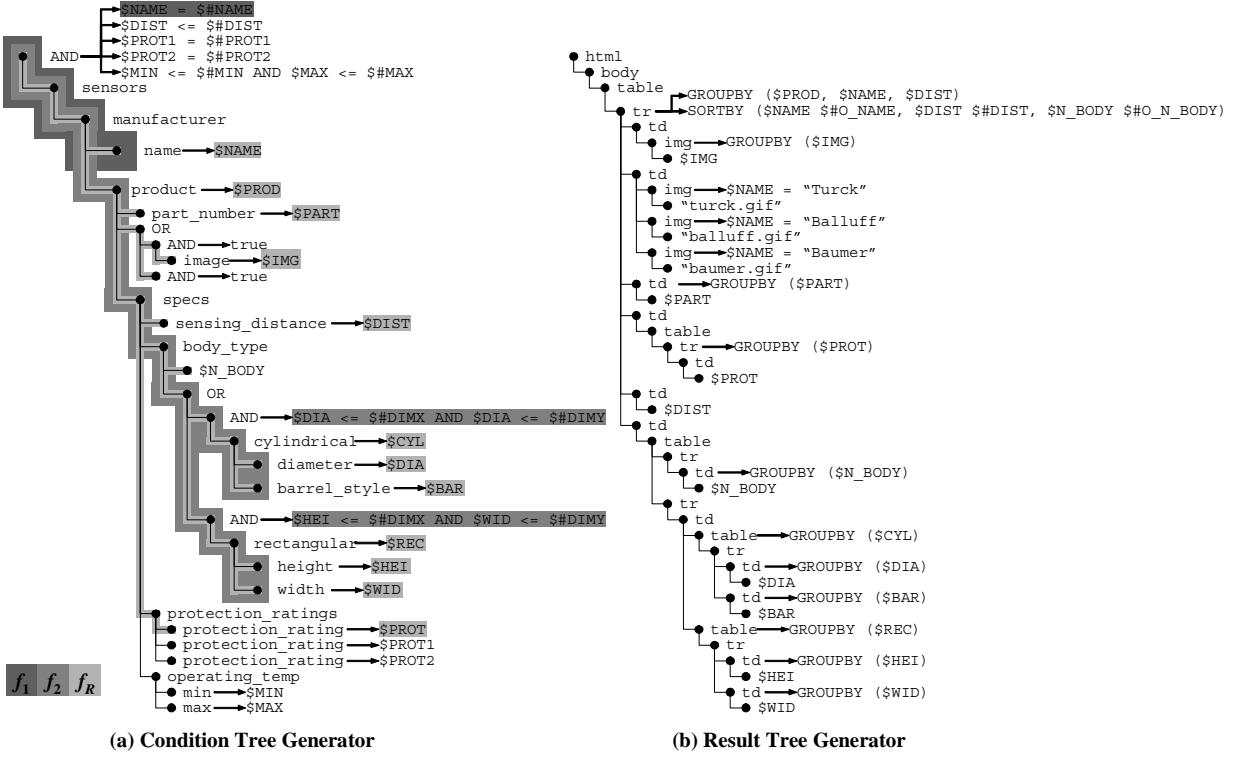
(a) Condition Tree Generator **(b) Result Tree Generator**

Figure 8 Query Set Specification

Definition 4 (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 fragment*.

    - Boolean expressions can use *parameters* (a.k.a. placeholders [24]) as operands of their predicates. Parameters are denoted by the $\$\#$ symbol and must bind to a value [50].

        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 list *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 $\$\#O\_$. 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.

22

- *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 the *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 f_R, called *result fragment*, that includes all the element nodes whose variables appear in the *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 subtee 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 the *RTG* in Figure 8b, and imposes no Boolean conditions.

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* ⟨*CTG,RTG,F,D*⟩ and a query/visual association, 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.
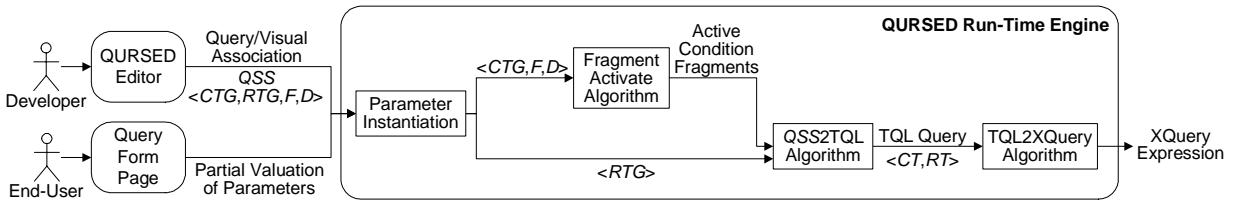

Figure 9 Query Formulation Process

23

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 and on the query/visual association, a partial valuation ν 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:

ν = {$#PROT1:"NEMA3", $#DIMX:"20", $#DIMY:"40", $#O_NAME:"DESC", $#O_DIST:"ASC"}

Based on ν, the run-time engine instantiates the parameters of condition fragments in *F*. For example, the above partial valuation instantiates the parameters $#DIMX and $#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 lists of the *RTG*, and the parameters of Boolean expressions labeling nodes of the *RTG*, are instantiated. The ordering parameters can take the values "DESC" or "ASC", as in the case of $#O_NAME and $#O_DIST in the above partial valuation. An example of an *RTG*, where parameterized Boolean expressions label its nodes, is shown in Section 7.5. Finally, the run-time engine also instantiates the parameters of the set of dependencies *D*. Dependencies are presented in the next section.

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. The algorithm renders a condition fragment *active* if it has all its parameters instantiated by the partial valuation ν. Since the partial valuation ν might not provide values for all the parameters used in the *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 ν. As a special case, the result fragment f_R is always active, since it doesn't have any parameters.

Note that the FragmentActivate algorithm in 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 $f_1,…,f_n$, along with the edges that connect them. Each AND

node n_{AND} 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 n_{AND} in the fragments $f_1,\ldots,f_n$ respectively.

Similarly, in order to convert the *RTG* to the *RT*, the *QSS*2TQL algorithm first eliminates from the *RTG* the subtrees rooted at nodes labeled with a Boolean expression *b* that has uninstantiated parameters or evaluates to *false*, as further explained in Section 7.5. Then for every node that has a sort-by list *s*, we keep in the label only the variables with instantiated ordering parameters.

As an example of the *QSS*2TQL 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.

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 a declarative way to control the appearance of visual fragments.

Definition 5 (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 $\wedge$, $\vee$ and $\neg$. The predicates consist of arithmetic and comparison operators and functions that use parameters from the *CTG* 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 10a, and let us define two dependencies d_1 and d_2 as follows:

$$\langle f_2, \ \texttt{\$\#BODY = "cylindrical"}, \ \{f_1\}\rangle \quad (d_1)$$
$$\langle f_3, \ \texttt{\$\#BODY = "rectangular"}, \ \{f_1\}\rangle \quad (d_2)$$
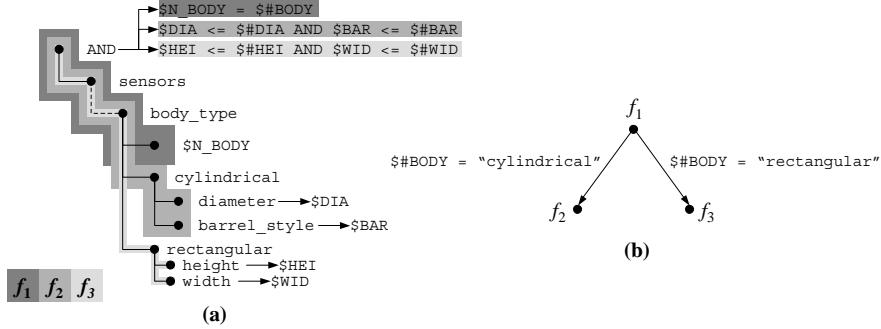
Figure 10 Condition Tree Generator and Dependencies Graph

The condition fragment f_1 uses the parameter $\texttt{\$\#BODY}$ that appears in the condition of both dependencies on f_2 and f_3. If a value is not provided for $\texttt{\$\#BODY}$, then neither dependency holds, and f_2 and f_3 are inactive. If the value "cylindrical" is provided, then f_1 is active, the condition for d_1 is true, and so f_2 is rendered active.
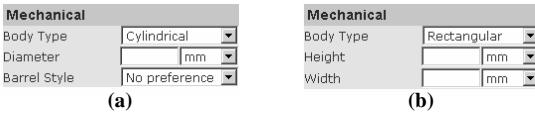
Figure 11 Dependencies on the Query Form Page

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 11 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 10a. For instance, the condition fragment f_1 applies a condition to the element node labeled with $\texttt{\$BODY}$ 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 11a). 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.

Obviously, circular dependencies must be avoided, since the involved dependent fragments can never become active. This restriction is captured by the *dependency graph*:

Definition 6 (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, labeled with the condition B of d. ■

The dependency graph for the dependencies d_1 and d_2 defined above is shown in Figure 10b. 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 [26] (hence of complexity $\Theta(V+E)$) and is outlined below. Note that, when evaluating a condition b of a dependency, any predicates that contain uninstantiated parameters evaluate to *false*.

Algorithm FragmentActivate
Inputs: A dependencies graph $G = <V, E>$, and a partial valuation ν over P, where P is the set of the parameters that appear in the *QSS*.
Output: The set A of active condition fragments.
Method:
1   $A \leftarrow \emptyset$
2   Compute the set of fragments B whose parameters are all instantiated by ν
3   For each edge (n, u) in E
4   Evaluate the condition on edge (n, u)
5   Repeat
6   If node u belongs to B and has no incoming edges
7       $A \leftarrow \{u\}$
8   If node u belongs to B, has an incoming edge (n, u) where n belongs in A, and the condition on (n, u) is true
9       $A \leftarrow \{u\}$
10 Until A reaches fixpoint

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

7   QURSED EDITOR

The QURSED Editor is the tool the developer uses to build *QFR*s. Figure 12 shows the Editor's architecture, how the developer interacts with the graphical user interface, and how the Editor interprets these visual actions in order to construct the *QSS* and the query/visual association of a *QFR*.

The developer builds a condition tree generator by constructing a set of Boolean expressions based on the input XML Schema, in the form of an *EST*, and the input XHTML query form page that are displayed to her. Internally,

the Editor interprets the set of Boolean expressions as the set of condition fragments of the *QSS* and the query/visual association. The Editor constructs the *CTG* by building each condition fragment f, as if f was the only fragment of the condition tree generator, and then merging f with the *CTG*. A key step in that process is that the Editor checks if f is meaningful by considering the presence of CHOICE elements in the *EST* and, if necessary, manipulates f by introducing heuristically structural disjunction operators (OR nodes). The developer also builds the set of dependencies on the set of condition fragments that become part of the *QSS*. These processes are described in Sections 7.1 and 0.
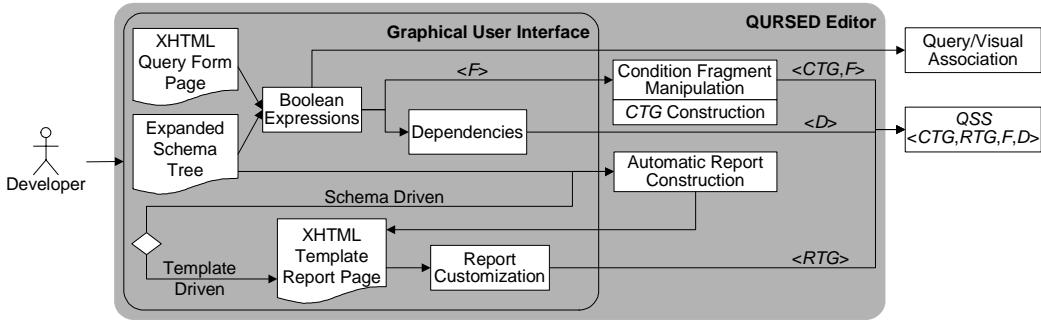

Figure 12 QURSED Editor Architecture

For the construction of the result tree generator, the developer has two choices that are illustrated as a diamond on Figure 12. Either an XTMHL template report page is automatically constructed based on the *EST* (schema-driven), or one is provided as an input (template-driven). Either way, the Editor constructs internally an *RTG* that becomes part of the *QSS*. This process is described in Section 7.3. The developer can also further customize the template report page report by building Boolean expressions and adding dynamic projection functionality, presented in Sections 7.4 and 7.5.

A key benefit of the Editor is that it enables the easy generation of semistructured queries with OR nodes by considering the presence of CHOICE elements in the *EST*. 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

Figure 13a demonstrates how the developer uses the Editor to define the condition fragment f_1 of Figure 8a. The main window of the Editor presents the sample *EST* of Section 3.1 in the left panel, and the query form page in the right panel. The query form page is displayed as an XHTML tree that contains a `form` element node and a set of

form controls, i.e., `select` and `input` element nodes [46]. The XHTML tree corresponds to the page shown on

Figure 13b rendered in the Macromedia HomeSite [29] WYSIWYG XHTML editor. Based on this setting, the

developer defines 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.
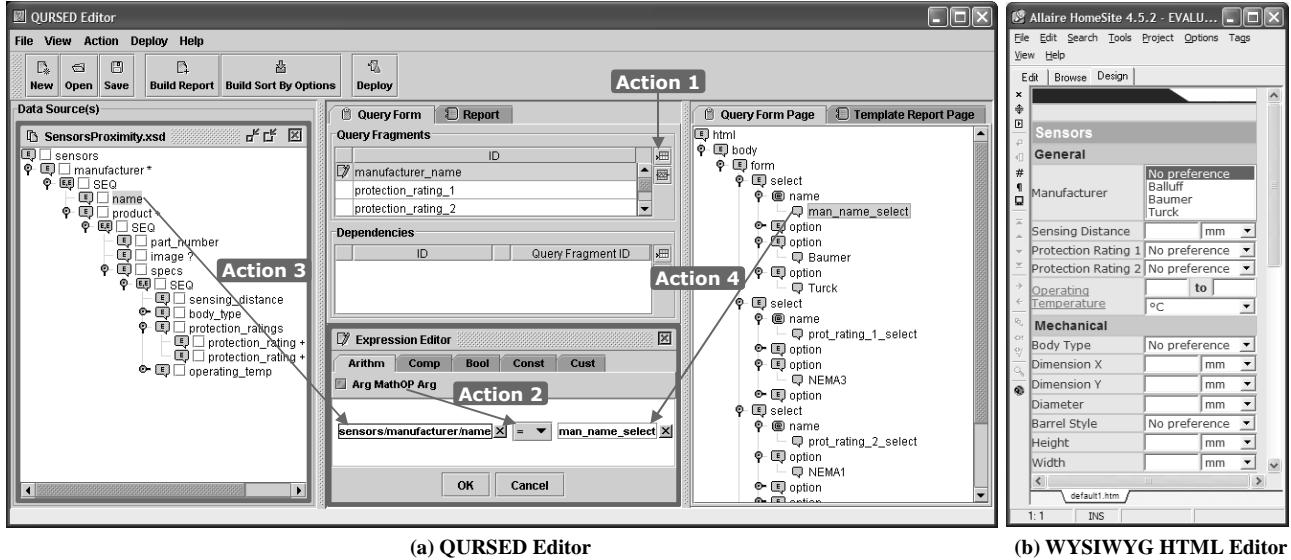
| (a) QURSED Editor | (b) WYSIWYG HTML Editor |

Figure 13 Building a Condition Fragment

The developer starts by clicking on the "New Condition Fragment" button (Action 1 of Figure 13a) and

providing a unique `ID`, which is `manufacturer_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) and setting its left operand to be the element node `name` (Action 3). The full path name of the node

appears in the left operand box and is also indicated by the highlighting of the `name` element node on the left panel.

As a final step, the developer binds the right operand of the equality predicate to the `select` XHTML form control

named `man_name_select` (Action 4) thus establishing a query/visual association and defining as the visual

fragment the "Manufacturer" form control shown in Figure 13b. Internally, the Editor creates the parameter

`$#NAME`, associated with the "Manufacturer" form control of Figure 13b, and sets it as the right operand of the

Boolean expression, as Figure 8a shows.

In order to build more complex condition fragments, Actions 2, 3 and 4 can be repeated multiple times, thus

introducing multiple variable and parameters and including more than one XHTML form control in the

corresponding visual fragment.

29

Note that, even though the visual actions introduce variables and parameters in the condition fragment, the developer does not need to be aware of their names. In effect, variables correspond to path names and parameters to XHTML form control names. The Editor interprets the Boolean expression as a condition fragment that conjunctively combines all paths of the expression.

7.1.1 *Automatic Introduction of Structural Disjunction*

The semistructuredness of the schema (CHOICE nodes and optional elements) may render the Boolean expression meaningless and unsatisfiable. The Editor automatically, and by employing a heuristic, manipulates a condition fragment *f* by introducing structural disjunction operators (OR nodes) that render *f* meaningful.

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. Sensors can be either cylindrical or rectangular. The developer's intention is to specify that either the diameter is less than dimensions X and Y, or the height is less than dimension X and the width less than Y. The developer constructs the following Boolean expression by following the previously described steps:

`($DIA <= $#DIMX ∧ $DIA <= $#DIMY) ∨ ($HEI <= $#DIMX ∧ $WID <= $#DIMY)`

The `$DIA`, `$HEI` and `$WID` variables label the `diameter`, `height` and `width` elements of the *EST*. The `$#DIMX` and `$#DIMY` parameters are associated with the "Dimension X" and "Dimension Y" form controls.

However, the query where the above Boolean expression is interpreted as a condition fragment that conjunctively combines the paths to `diameter`, `height` and `width` elements is unsatisfiable, since no sensor has all of them. The Editor captures the original intention by automatically manipulating the ∨ Boolean connective and treating it as an OR node of TQL, as the condition fragment f_2 in Figure 8a indicates. The OR node corresponds to the CHOICE node in the *EST* of Figure 2c. Two AND nodes are also introduced and are labeled with the conjunctions in the initial Boolean expression, namely: `($DIA <= $#DIMX ∧ $DIA <= $#DIMY)` and `($HEI <= $#DIMX ∧ $WID <= $#DIMY)`. The manipulation of a condition fragment is part of the Construct*CTG* algorithm.

The Construct*CTG* algorithm creates a condition tree generator by merging the condition fragments. It operates incrementally by merging each condition fragment *f* with the condition tree generator already constructed from the previous condition fragments. The main step of the algorithm manipulates *f* by employing a heuristic, such that *f* produces meaningful satisfiable queries given the Boolean expression *b*. In particular, the algorithm introduces structural disjunction operators into *f* by replacing Boolean connectives ∨ in *b* with OR nodes, as illustrated in the

example above. The manipulation is driven by the CHOICE nodes and optional elements in the schema. An initial step of the algorithm checks if f can be manipulated to produce meaningful, satisfiable queries. This is accomplished by bringing b to disjunctive normal form and identifying at least one unsatisfiable conjunction. If there is one, then the algorithm terminates outputting an error. The final step of Construct*CTG* merges f with the input *CTG*. The order that the condition fragments are passed to the algorithm does not matter.

The Construct*CTG* algorithm assumes a function *node*(V) that, given a variable V in b, returns the node n 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) returns the parent of the node that the developer drag 'n' dropped. It also assumes the existence of a function *copy*(n) that, given a node n in the *EST*, returns the copy of it in f, or *null*, if one doesn't exist.

Algorithm Construct*CTG*
Input: A condition fragment f with a Boolean expression b labeling its 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.
Method:
Step 1: Satisfiability Check of f
1   Rewrite b in disjunctive normal form such that $b = c_1 \vee c_2 \ldots \vee c_n$, where c_i is a conjunction of predicates
2   If a conjunction c_i, where $1 \leq i \leq n$, uses two variables V_{ix}, V_{iy} such that the lowest common ancestor of $node(V_{ix})$ and $node(V_{iy})$ in the *EST* is a CHOICE node
3   Output an error indicating the unsatisfiable conjunctions

Step 2: Manipulation of f
     // Introduces OR nodes to f based on CHOICE nodes in the EST
4   For any two variables V_{ix}, V_{jy} used in conjunctions c_i and c_j of b, respectively, where $1 \leq i,j \leq n$ and $i \neq j$
5   If both the paths from $node(V_{ix})$ and $node(V_{jy})$ to their lowest element node common ancestor n_{ANSC} in the *EST* contain either a CHOICE node or an optional element, excluding n_{ANSC}
6   Apply the Rules 1 and 2 of Figure 14
     // Label AND nodes with Boolean expressions
7   For each conjunction c_i of b, $1 \leq i \leq n$
8   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
9   If the AND node is labeled with more than one conjunction
10   Combine them with the $\vee$ Boolean connective

Step 3: Addition of f to the *CTG*
11   Set the children of the root AND node of f as children of the root AND node of the *CTG*
12   Take the union of the sets of Boolean expressions labeling the root AND node of f and the root AND node of the *CTG* and label the root AND node of the latter with it
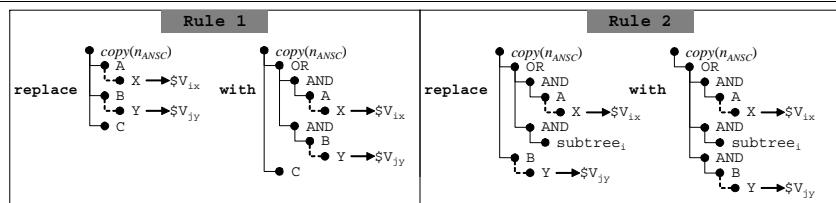

**Figure 14 "OR Node Introduction" Rules**
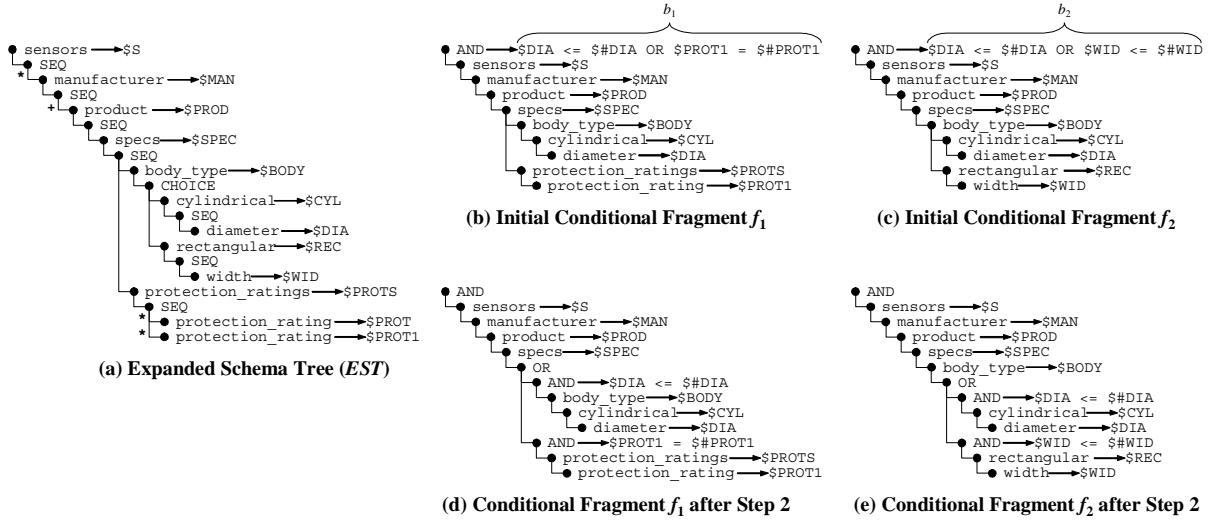
Figure 15 Example of the Construct*CTG* Algorithm

Line 6 of the algorithm covers two cases that are illustrated in Figure 14. In the first case, the node *copy*(n_{ANSC}) does not have an OR child node and Rule 1 shows how the condition fragment *f* is manipulated. In the second case the node *copy*(n_{ANSC}) has an OR child node n_{OR} and the subtree *tree*$_{ix}$ that contains *node*(V_{ix}) is a child of an AND child node n_{AND} of n_{OR}, and *tree*$_{jy}$ that contains *node*(V_{jy}) is a child of *copy*(n_{ANSC}). In this case, Rule 2 does not introduce a new OR node, but places the subtree rooted at B under the existing OR node instead.

Figure 15 illustrates an example of the application of the Construct*CTG* algorithm on the 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 15(b) and (c) respectively. f_1 asks for sensors either having diameter less than the parameter $#DIA or a protection rating equal to the parameter $#PROT1, 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 Construct*CTG* algorithm, since both conjunctions of b_1 and b_2 involve a single variable. In Step 2, structural disjunction operators are 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 the *EST* and element node protection_rating is optional. So an OR node is introduced under their lowest common ancestor node specs. Similarly, in f_2, the nodes diameter and width are both under a CHOICE node in the *EST*, so an OR node is introduced under the node body_type.

Step 3 of the Construct*CTG* algorithm just puts f_1 and f_2 together, thus constructing the merged *CTG* shown in Figure 17a, where the two fragments are indicated in two different tones of gray.

7.1.2 Eliminating Redundancies

The Editor eliminates redundancies in the merged *CTG* in order to improve the performance of the generated TQL queries. As shown in [3], efficiency of tree pattern queries depends on the size of the pattern, so it is essential to identify and eliminate redundant nodes. More specifically, according to the rule of Figure 16, the Editor renders redundant an element node that has a sibling node labeled with the same variable.
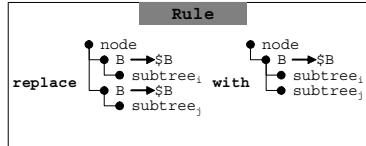
Figure 16 "Node Elimination" Rule

The application of the rule takes time linear in the number of nodes of the *CTG*. The process of eliminating redundant nodes could also be performed on TQL queries, instead of the *CTG*, at run-time. Either way, the final TQL query is the same, so it is preferable to perform the optimization at compile-time.

(a) Initial *CTG*

(b) *CTG* after "Node Elimination" Rules
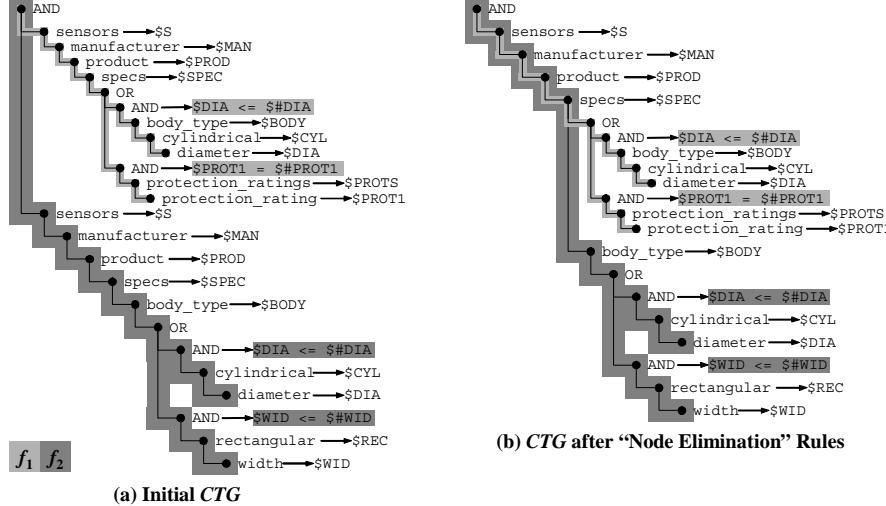
Figure 17 Eliminating Redundant Nodes on the *CTG*

Notice that the rule is not complete for minimization of TQL queries. Indeed, the minimization problem for TQL queries can easily be shown to be NP-hard, while the rule application is polynomial. The rule is introduced specifically to eliminate those redundancies introduced during the construction of the *CTG*, presented in the previous section. For example, the Construct*CTG* algorithm constructs the *CTG* of Figure 17a by merging two fragments. The

path from the `sensors` node to the `specs` node appears in both condition fragments, and every element node along the path is labeled with the same variable in both fragments. One of these paths is eliminated by parsing the *CTG* top-down and iteratively applying the rule of Figure 16. The resulting *CTG* is shown in Figure 17b. Note that the rule preserves the boundaries of the fragments as element nodes are being eliminated.

7.2   Building Dependencies

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. As an example, Figure 18 demonstrates how the developer builds dependency d_1: <f_2, `$#BODY="cylindrical"`, $\{f_1\}$> of Section 6.1 by performing a set of actions indicated by the numbered arrows. Dependency d_1 sets the condition fragment f_2 on the cylindrical dimensions (Figure 10a) active if the parameter `$#BODY` is set to "cylindrical".
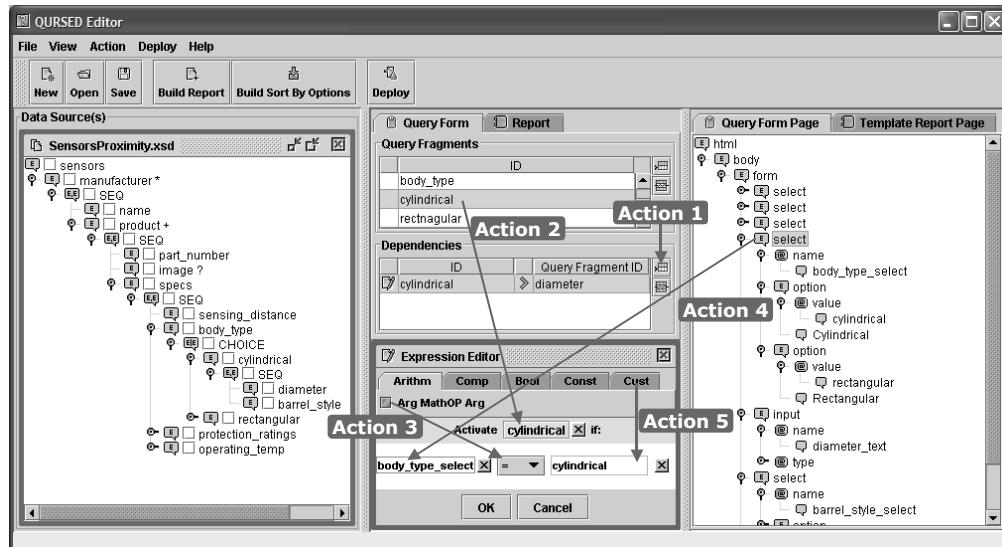
Figure 18 Building Dependencies

First, the developer initiates a dependency (Action 1 of Figure 18) 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 to be the "cylindrical" one (Action 2), and builds the condition of the dependency in the expression editor (Action 3). She specifies that the left operand of the equality predicate is a parameter bound to the "Body Type" `select` form control (Action 4), and the right operand to be the string constant "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 defined in Section 6.1.

34

7.3 Building Result Tree 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, called *schema-driven*, 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 XHTML 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, called *template-driven*, where the developer provides as input a template report page to guide the result tree generator construction. Both options are described next.
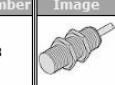
Figure 19 Schema-Driven Constructed Report Page

7.3.1 Schema-Driven Construction of Result Tree Generator

The developer can automatically build a result tree generator based on the nesting of the *EST*. For example, Figure 19 shows a report page created from the result tree generator for the data set and the *EST* of Figure 2. 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 20.

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 20). 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 21a. The variables that will be used in the result tree generator are also indicated. Then, the Editor automatically generates the template report page (Action 2) displayed on the right panel of Figure 20 as a tree of XHTML element nodes. Figure 21c shows how a WYSIWYG XHTML editor renders the template report page. The Editor translates the above actions into a *QSS* as follows.
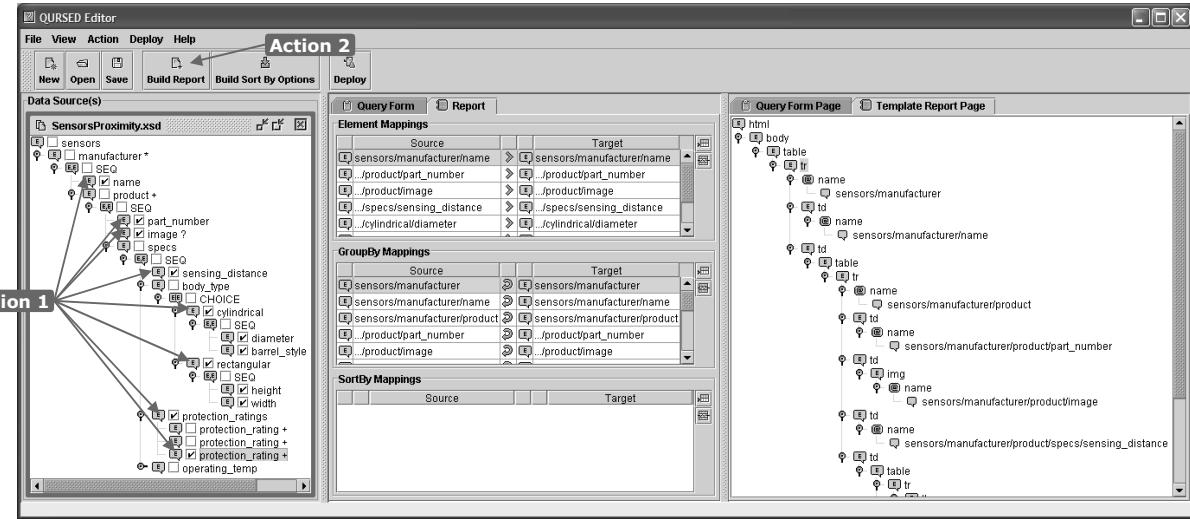
Figure 20 Selecting Elements Nodes and Constructing Template Report Page

In Action 2, the Editor automatically generates the result tree generator of Figure 21b that presents the element

nodes selected in Action 1 using XHTML `table` element nodes that are nested according to the nesting of the *EST*.

For illustration purposes, each `table` element node in Figure 21b 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 21c 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 `td` (table data

cell) element nodes. For example, in the result tree generator of Figure 21b the element variable $NAME appears as

the child of the `td` element node of the "manufacturer" table.

We discuss next how optional and repeatable element nodes and CHOICE nodes in the *EST* are handled by the

Editor on the template report page.

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 present or not (see Figure 19). Figure 21a

demonstrates the effect of the visual action to select the optional element `image` to appear on the report page.

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 21b, and

the group-by list of its `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, as in the case where 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.

(a) Condition Tree Generator

(b) Result Tree Generator

(c) Template Report Page
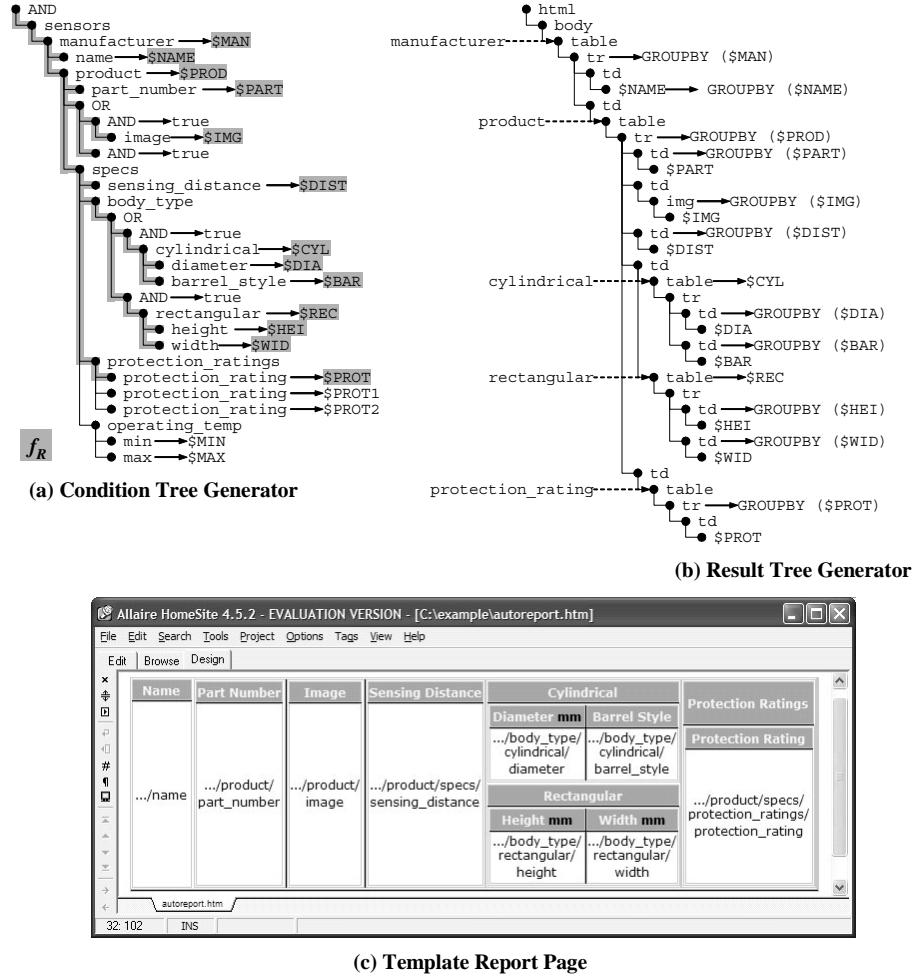
Figure 21 Automatically Generated Result Fragment, Result Tree Generator and Template Report Page

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 manipulates the result fragment f_R by introducing OR nodes based on CHOICE nodes and optional elements in the *EST*. 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*, returns the copy of it in f_R, if one exists, 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 result fragment f_R and the result tree generator *RTG*.
Method:
Step 1: Manipulation of f_R
    // Introduce OR nodes in f_R based on CHOICE nodes and optional elements in the EST
1   Traversing f_R top-down, for an element node n_i
2   If n_i is labeled with a variable $\$V_i$ and $parent(node(\$V_i))$ is a CHOICE node and $parent(n_i)$ isn't an OR node
3   If there is a sibling n_j of n_i labeled with a variable $\$V_j$ such that $node(\$V_j)$ is a sibling of $node(\$V_i)$
4   For all sibling element nodes n_j of n_i labeled with a variable $\$V_j$ such that $node(\$V_j)$ is a sibling of $node(\$V_i)$
5   Apply Rule 1 of Figure 22
6   Else
7   Apply the Rule 2 of Figure 22   *// Treat n_i as optional element*
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
9   Apply the Rules 2 and 3 of Figure 22 correspondingly

Step 2: Construction of the result tree generator *RTG*
10   Create a node n_r named "html", a node n_b named "body", a node n_t named "table", and a node n_{tr} named "tr"
11   Set n_r as the root of the *RTG*, n_b as a child of n_r, n_t as a child of n_b, and n_{tr} as a child of n_t
12   Traversing the *EST* top-down and left to right, ignoring SEQ, CHOICE and ALL nodes, for an element node n_i
13   BuildTable(n_i, n_{tr})

BuildTable (n_i, n_{tr})

14   If n_i is either repeatable or $parent(n_i)$ is a CHOICE node
15   Create a node n_{td} named "td" and a node n_t named "table"
16   Set n_{td} as a child of n_{tr} and n_t as a child of n_{td}
17   Create a node named "tr" and set it as the current n_{tr}
18   If $parent(n_i)$ is a CHOICE node
19   Attach the Boolean expression $var(n_i)$ to n_t
20   If n_i is repeatable
21   Add $var(n_i)$ to the group-by list of n_{tr}
22   If n_i is a selected element node
23   Create a node n_{th} named "th" and add it as a child of n_{tr}
24   Create a node named $name(n_i)$ and add it as a child of n_{th}
25   If n_i is a leaf element node
26   Create a node named "td", add it as a child of n_{tr}, and set it as the current n_{td}
27   Create a node named $var(n_i)$ and add it as a child of n_{td}
28   If $var(n_i)$ is not in any group-by list of an ancestor node
29   Add $var(n_i)$ to the group-by list of n_{td}
30   For every child element node n_c of n_i
31   BuildTable(n_c, n_{tr})
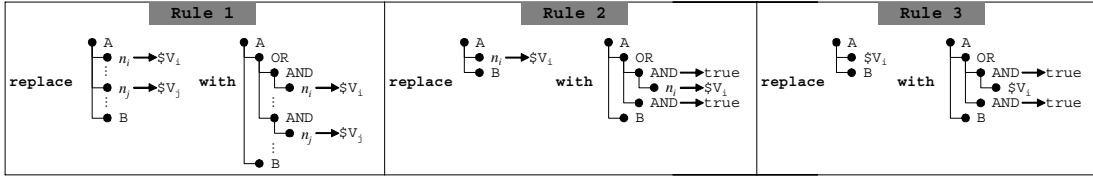
Figure 22 "OR Node Introduction" Rules for Result Fragment f_R

The result fragment f_R that is manipulated 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 7.1.1 and redundant nodes are eliminated using the rule of Figure 16.

7.3.2   Template-Driven Construction of Result Tree Generator

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 XHTML 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 19 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 23 and provides it to the Editor.
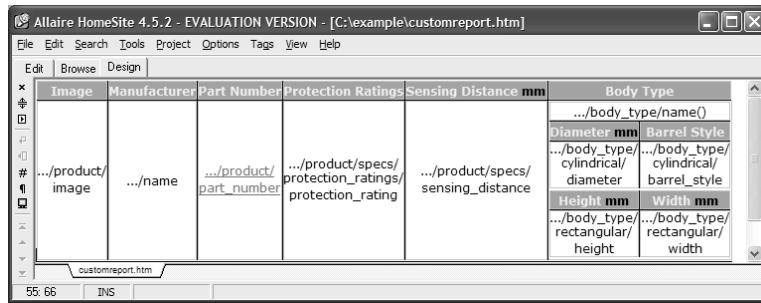
Figure 23 Editing the Template Report Page

On the right panel of Figure 24 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 Figure 24.

First, the developer creates a new element, group-by or sort-by mapping (Action 1). Depending on what mapping was created, one of Actions 2, 3, or 4 is performed.
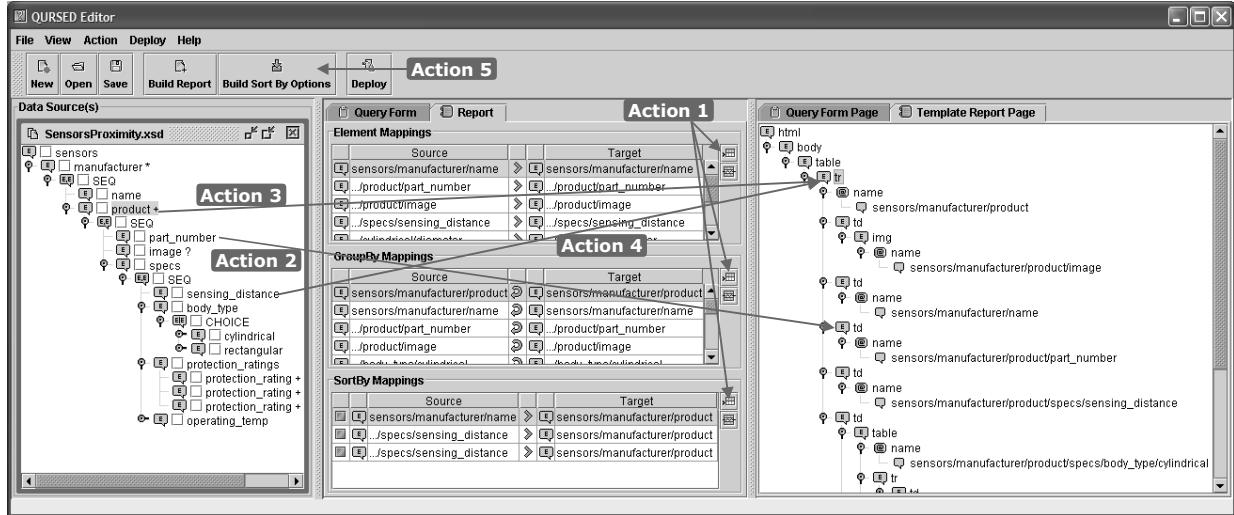
Figure 24 Performing Element and Group-By Mappings on the Template Report Page

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 or 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 mapping the part_number element node to the td element node on the template report page, the $PART variable is implicitly placed in the result tree generator of Figure 8b.

In the case of group-by mapping, the developer maps element nodes from the *EST* to any nodes of the template report page (Action 3). For example, by mapping the product element node to the tr element node of the outermost table in the template report page, the $PROD element variable is added to the group-by list of the tr. This action will result in one tr element node for each binding of the $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 mapping the sensing_distance element node to the tr element node of the outermost table, the sort-by list of that element, shown in Figure 8b, is generated. The Editor defines automatically a group-by mapping for each sort-by mapping, if there isn't one. 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 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 (Action 5). 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

40

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 *QSS*2TQL 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 XHTML editor and further customized visually, even after the mappings have been defined.

Based on the above actions, the result fragment f_R is defined as the set of variables used in the result tree generator that the developer manually constructs. The f_R is constructed by Step 1 of the AutoReport algorithm of Section 7.3.1, merged with the condition tree generator of a *QSS* according to Step 3 of the Construct*CTG* algorithm of Section 7.1.1, and redundant nodes are eliminated using the rule of Figure 16.

7.4   Building Result Boolean Expressions

In Figure 3, the manufacturer's column does not display the name as text, but a corresponding image (logo) is presented instead. This effect is accomplished by the three img elements, corresponding to the three possible manufacturers, shown in the result tree generator *RTG* of the *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 *RTG*.

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 18, 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 18. 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* at run-time.

- In Actions 4 and 5, the developer need not specify only 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.

Note that the Boolean expressions containing variables are translated to XQuery conditional expressions [52], 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, then they are evaluated during the formulation of the TQL query, as the *QSS*2TQL 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 25 shows the *RTG* of the *QSS* of Figure 8, where Boolean expressions controlling the dynamic projection label `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 Presentation" section on the query form page of Figure 3. If a checkbox is checked, then the corresponding Boolean expression evaluates to *true* and the subtree is included in the result tree of the TQL query formulated during run-time. These Boolean expressions are defined by the developer using the actions described in Section 7.4, but instead of nodes from the *EST*, the developer sets as operands of the Boolean expression the checkboxes from the query form page.

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 schema-driven 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 oriented 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 on the report page. For example, on the "Customize Presentation" table on Figure 3, the end-user cannot prevent the projection of "Part Number" and "Sensing Distance".

```
● html
  ● body
    ● table ──►GROUPBY ($PROD, $NAME, $DIST)
      ● tr ──►SORTBY ($NAME $#O_NAME, $DIST $#DIST, $N_BODY $#O_N_BODY)
        ● td ──►$#P_IMG = true
          ● img ──►GROUPBY ($IMG)
            ● $IMG
        ● td ──►$#P_MAN = true
          ● img ──►$NAME = "Turck"
            ● "turck.gif"
          ● img ──►$NAME = "Balluff"
            ● "balluff.gif"
          ● img ──►$NAME = "Baumer"
            ● "baumer.gif"
        ● td ──►GROUPBY ($PART)
          ● $PART
        ● td ──►$#P_PROT = true
          ● table
            ● tr ──►GROUPBY ($PROT)
              ● td
                ● $PROT
        ● td
          ● $DIST
        ● td
          ● table
            ● tr
              ● td ──►GROUPBY ($N_BODY)
                ● $N_BODY
            ● tr
              ● td
                ● table ──►GROUPBY ($CYL)
                  ● tr
                    ● td ──►$#P_DIA = true
                          ──►GROUPBY ($DIA)
                      ● $DIA
                    ● td ──►$#P_IMG = true
                          ──►GROUPBY ($BAR)
                      ● $BAR
                ● table ──►GROUPBY ($REC)
                  ● tr
                    ● td ──►$#P_HEI = true
                          ──►GROUPBY ($HEI)
                      ● $HEI
                    ● td ──►$#P_IMG = true
                          ──►GROUPBY ($WID)
                      ● $WID
```

Result Tree Generator

Figure 25 Boolean Expressions for Dynamic Projection

8   CONCLUSIONS AND FUTURE WORK

We presented QURSED, a system for the generation of powerful 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 of the query forms and reports. An on-line demonstration of the system is available at http://www.db.ucsd.edu/qursed/.

Future work in this area should consider extending the set of queries that can be expressed with TQL to a bigger subset of XQuery and correspondingly increase the power of the query set specification, in order to capture richer

43

form (and source) capabilities. A challenge will be to enhance the querying power while keeping the Editor's interface as intuitive as it is now. Moreover, given that the Editor employs heuristics in translating developer input into query set specifications and (ultimately) *QFR*s, user studies are necessary to evaluate the quality of these decisions as well as the usability of form generation systems and their resulting forms in general.

QURSED is one of the first attempts to describe formally the logical capabilities of query forms and reports and to clearly separate them from the form and report presentation. The approach followed by QURSED, to model form capabilities using query set specifications, is promising for other capability modeling tasks, such as the problem of describing and automatically integrating rich data management-oriented web services. In particular, *QSS* can be the basis for a highly expressive language for the description of data-oriented Web services [36]. A Web service designer would generate a Web service using a graphical web services Editor similar to the one described here. The Web services specification could then be used either as an input to a run-time mediator that would decide whether particular service requests fall within the capabilities of the web service [44], or simply to generate and package a low-level WSDL [47] description for the service.

9   ACKNOWLEDGMENTS

REFERENCES

[1]   S. Abiteboul, P. Buneman, D. Suciu: *Data on the Web*, Morgan Kaufman, California, 2000.
[2]   S. Abiteboul, L. Segoufin, V. Vianu: *Representing and Querying XML with Incomplete Information*, in Principles Of Database Systems (PODS), 2001 pp. 150-161.
[3]   S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, D. Srivastava: *Minimization of Tree Pattern Queries*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 2001, pp. 497-508.
[4]   P. Atzeni, G. Mecca, P. Merialdo: *To Weave the Web*, in proceedings of the 23rd International Conference on Very Large Databases (VLDB), 1997, pp. 206-215.
[5]   P. Bernstein et al.: *The Asilomar report on database research*, SIGMOD Record 27(4), 1998, pp. 74-80.
[6]   M. Carey, L. Haas, V. Maganty, J. Williams: *PESTO: An Integrated Query/Browser for Object Databases*, in proceedings of the 22nd International Conference on Very Large Databases (VLDB), 1996, pp. 203-214.
[7]   S. Chawathe, T. Baby, J Yeo: *VQBD: Exploring Semistructured Data* (demonstration description), in proceedings of the ACM SIGMOD International Conference on Management of Data, page 603, 2001, pp. 603.
[8]   S. Cluet, C. Delobel, J. Siméon, K. Smaga: *Your Mediators Need Data Conversion!*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 1998, pp. 177-188.
[9]   S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, A. Serebrenik: *EquiX – Easy Querying in XML Databases*, in proceedings of the ACM Workshop on The Web and Databases (WebDB), 1999, pp. 43-48.
[10] S. Comai, E. Damiani, P. Fraternali: *Computing graphical queries over XML data*, in the ACM Transactions on Information Systems (TOIS) 19(4), 2001, pp. 371-430.

[11] A. Deutsch, M. Fernandez , D. Florescu, A. Levy , D. Suciu: *XML-QL: A Query Language for XML*, W3C note, 1998.
http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/

[12] A. Deutsch, Y. Papakonstantinou: *Optimization of Nested XQueries Using Minimization*, submitted for publication, 2003.

[13] D. Draper, A. Halevy, D. Weld: *The Nimble Integration Engine*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 2001, pp. 567-568.

[14] Enosys Software
http://www.enosyssoftware.com

[15] A. Eyal, T. Milo: *Integrating and customizing heterogeneous e-commerce applications*, VLDB Journal 10(1), 2001, pp. 16-38.

[16] M. Fernández, A. Morishima, D. Suciu: *Efficient Evaluation of XML Middle-ware Queries*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 2001, pp. 103-114.

[17] M. Fernández, D. Suciu and I. Tatarinov: *Declarative Specification of Data-intensive Web sites*, in proceedings of the Workshop on Domain Specific Languages (DSL), 1999, pp. 135-148.

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

[19] P. Fraternali, P. Paolini: *Model-Driven Development of Web Applications: the Autoweb System*, in the ACM Transactions on Office Information Systems 18 (4), 2000, pp. 323-382.

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

[21] R. Goldman, J. Widom: *Interactive Query and Search in Semistructured Databases*, in proceedings of the ACM Workshop on The Web and Databases (WebDB), 1998, pp. 52-62.

[22] Java Server Pages, White Paper
http://java.sun.com/products/jsp/whitepaper.html

[23] A. Y. Levy, A. Rajaraman, J. J. Ordille: *Querying Heterogeneous Information Sources Using Source Descriptions*, in proceedings of the 22nd International Conference on Very Large Databases (VLDB), 1996, pp. 251-262.

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

[25] B. Ludäscher, Y. Papakonstantinou, P. Velikhov: *Navigation-Driven Evaluation of Virtual Mediated Views*, in Extending Database Technology (EDBT), 2000, pp. 150-165.

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

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

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

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

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

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

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

[33] K. Munroe, Y. Papakonstantinou: *BBQ: A Visual Interface for Browsing and Querying XML*, in VDB5, 2000, pp. 277-296.

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

[35] Y. Papakonstantinou, M. Petropoulos, V. Vassalos: *QURSED: Querying and Reporting Semistructured Data*, in proceedings of the ACM SIGMOD International Conference on Management of Data, 2002, pp. 192-203.

[36] M. Petropoulos, A. Deutsch, Y. Papakonstantinou: *Query Set Specification Language (QSSL)*, in Sixth International Workshop on the Web and Databases (WebDB), 2003, pp. 99-104.

[37] M. Petropoulos, V. Vassalos, Y. Papakonstantinou: *XML Query Forms (XQForms): Declarative Specification of XML Query Interfaces*, in proceedings of WWW10, 2001, pp. 642-651.

[38] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, R. Fagin: *Translating Web Data*, in proceedings of the 28th International Conference on Very Large Databases (VLDB), 2002, pp. 598-609.

[39] D. Quass et al.: *Querying Semistructured Heterogeneous Information*, in proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases (DOOD), 1995, pp. 319-344.

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

[41] J. Shanmugasundaram et al.: *Efficiently Publishing Relational Data as XML Documents*, in proceedings of the 26th International Conference on Very Large Databases (VLDB), 2000, pp. 65-76.

[42] A. Silberschatz, M. Stonebraker, J. D. Ullman: *Database Systems: Achievements and Opportunities - The "Lagunita" Report of the NSF Invitational Workshop on the Future of Database System Research held in Palo Alto, California, February 22-23, 1990*, SIGMOD Record 19(4), 1990, pp. 6-22.

[43] TIBCO XML Transform
http://tibco.com/solutions/products/extensibility/xml_transform.jsp

[44] V. Vassalos, Y. Papakonstantinou: *Expressive Capabilities Description Languages and Query Rewriting Algorithms*, Journal of Logic Programming, 43(1), 2000, pp. 75-122.

[45] W3C: *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation 6 October 2000.
http://www.w3.org/TR/REC-xml

[46] W3C: *HTML 4.01 Specification*, W3C Recommendation 24 December 1999.
http://www.w3.org/TR/html4/

[47] W3C: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C Working Draft 10 November 2003.
http://www.w3.org/TR/wsdl20/

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

[49] W3C: *XML Schema Part 0: Primer*, W3C Recommendation 02 May 2001.
http://www.w3.org/TR/xmlschema-0/

[50] W3C: *XML Schema Part 2: Datatypes*, W3C Recommendation 02 May 2001.
http://www.w3.org/TR/xmlschema-2/

[51] W3C: *XSL Transformations (XSLT) Version 2.0*, W3C Working Draft 12 November 2003.
http://www.w3.org/TR/xslt20/

[52] W3C: *XQuery 1.0: An XML Query Language*, W3C Working Draft 12 November 2003.
http://www.w3.org/TR/xquery/

[53] W3C: *XQuery 1.0 and XPath 2.0 Data Model*, W3C Working Draft 12 November 2003.
http://www.w3.org/TR/query-datamodel/

[54] W3C: *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Working Draft 12 November 2003.
http://www.w3.org/TR/xquery-operators/

[55] W3C: *XQuery 1.0 Formal Semantics*, W3C Working Draft 12 November 2003.
http://www.w3.org/TR/query-semantics/

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

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 [52], but the draft includes an issue regarding an explicit GROUPBY construct. Such a construct is presented in Appendix C. The choice of augmenting XQuery 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 is shown in [12]. As Appendix C shows,

XQuery+GROUPBY expressions can always be translated to XQuery expressions, often of significantly increased complexity: the use of GROUPBY expressions results in cleaner query expressions and more opportunities for optimization, but does not affect the generality of the algorithm.

TQL2XQuery inputs the condition tree *CT* and the result tree *RT* of a TQL query. Condition tree and result tree nodes are denoted as n_{CT} and n_{RT}, respectively. We denote by *CT*[$V] the node n_{CT} such that *var*(*n*)=$V. The function *parent*(n_{CT}) returns the parent node of n_{CT}.

The main routine recursively traverses the result tree top-down and builds an XQuery expression that consists of nested FWOR (FOR-WHERE-ORDER BY-RETURN) expressions (lines 1-4). Every node n_{RT} that is labeled with a group-by list *g* or a Boolean expression *b* generates a FWOR expression that is nested within the RETURN clause of the FWOR expression created by the lowest ancestor of n_{RT} labeled with a group-by list or a Boolean expression.

For each variable $V in a group-by list or a Boolean expression of an n_{RT}, the algorithm proceeds in two steps (lines 7-10). The first step (ReachNode subroutine) declares $V in the FOR clause of the FWOR expression E^1. The second step (ApplyConditionSubTree subroutine) conditions the variable $V by translating the subtree rooted at *CT*[$V] into conditions in the WHERE clause of *E*. The *visibleVars* variable is the set of all variables declared in *E* and in all FWOR expressions that *E* is nested in.

The ReachNode subroutine declares a variable $V by taking as input the n_{CT}=*CT*[$V], the lowest ancestor n_{LA} of n_{CT}, such that *var*(n_{LA}) in *visibleVars*, and the current FWOR expression *E*. The algorithm "reaches" n_{CT} by walking the path from n_{LA} to n_{CT} recursively. For an AND node in the path, ReachNode adds the labeling Boolean expression *b* conjunctively to the WHERE clause of *E* and declares any variables in *b* that are not in *visibleVars*. For an element node in the path, the algorithm declares its labeling variable in the FOR clause of *E*, if not already in *visibleVars*. OR nodes are ignored.

The ApplyConditionSubTree subroutine "applies" to a variable $V declared by ReachNode the subtree of the condition tree rooted at *CT*[$V]. ApplyConditionSubTree takes as input the *CT*[$V] and the current FWOR expression *E*, and traverses the subtree rooted at *CT*[$V] top-down. For an element node in the subtree, the algorithm declares its labeling variable in the FOR clause of *E* (lines 32-35). For an AND node in the subtree, the labeling Boolean expression *b* is added to the WHERE clause of *E* (lines 36-39). For an OR node in the subtree, the algorithm

generates a SOME...SATIFIES expression for each one of its AND child nodes (lines 40-43). Subsequent variable declarations are added to the SOME clause and Boolean expressions are added conjunctively to the SATIFIES clause (lines 44-47). The disjunction of the generated SOME...SATIFIES expressions are added conjunctively to the WHERE clause of *E*. Note that for a nested OR node the algorithm generates nested SOME...SATIFIES expressions, in which case the disjunction of the generated SOME...SATIFIES expressions are added conjunctively to the SATIFIES clause of the nesting expression.

Finally, lines 11-21 of the main routine add to the RETURN clause of *E* a GROUPBY expression, a conditional expression and either a direct element constructor or an enclosed expression, if n_{RT} has a corresponding group-by list, Boolean expression and either a constant or a variable as name. Lines 22-23 add an ordering specification to the ORDER BY clause of *E*, if n_{RT} has a sort-by list.

Initially, the algorithm is called with TQL2XQuery(*CT*, *RT*, *nil*). There isn't an initial FWOR expression.

Algorithm TQL2XQuery
Input: *CT*, *RT*, *nil*
Output: An XQuery expression equivalent to the input TQL query
Method:
Traverse *RT* top-down and left-to-right. For an element node n_{RT} of *RT*:
1   Set *V*←variables in group-by list *g* of n_{RT} ∪ variables in Boolean expression *b* of n_{RT}
2   If there exists a variable V_i in *V* and not in *visibleVars*
3   Create a new FWOR expression *E*
4   If *E* is top-level FWOR expression   *// Implies top-level group-by list in RT*
5   Add to FOR clause of *E* the variable declaration "$doc IN accessSource('XMLDB')[2]"
6   For each variable V_i in *V* and not in *visibleVars*
7   Find the lowest element node ancestor n_{LA} of *CT*[V_i] such that *var*(n_{LA}) in *visibleVars*
8   ReachNode(n_{LA}, *CT*[V_i], *E*)
9   ApplyConditionSubTree(*CT*[V_i], *E*)
10   If group-by list *g* of n_{RT} is not empty
11   Add to RETURN clause of *E* the expression "GROUPBY *g* AS"
12   If n_{RT} has a Boolean expression *b*
13   Add to RETURN clause of *E* the expression "IF *b* THEN"
14   If *name*(n_{RT}) is a constant
15   Add to RETURN clause of *E* the expression <*name*(n_{RT})>
16   For each child n_C of n_{RT}, TQL2XQuery(*CT*, n_C, *E*)
17   Add to RETURN clause of *E* the expression </*name*(n_{RT})>
18   If *name*(n_{RT}) is a variable   *// then the node is guaranteed to be a leaf node, see Definition 3 in Section 4*
19   Add to RETURN clause of *E* the expression "{*name*(n_{RT})}"
20   If the sort-by list *s* of n_{RT} is not empty
21   Add to ORDER BY clause of *E* the *s* list

---

[1] If all the variables in a group-by list and a Boolean expression are declared in a nesting FWOR expression, then a new expression is not created.
[2] accessSource is a custom XPath function [54] that, given a name, retrieves the document element of the corresponding XML data set stored in the XML Data Server.

ReachNode(n_{LA}, n_{CT}, E)

22 If there is a child node n_C of n_{LA} that leads to n_{CT}

23 If n_C is an AND node

24 Add conjunctively to WHERE clause of E the Boolean expression b labeling n_{AND}

25 For every variable V_i in b and not in *visibleVars*, ReachNode(n_C, $CT[\$V_i]$, E)

26 If n_C is an element node and *var*(n_C) not is *visibleVars*

27 GenerateVarDeclaration(n_C, E)

28 ReachNode(n_C, n_{CT}, E)

ApplyConditionSubTree(n_{CT}, E)

29 If n_{CT} is an element node and *var*(n_{CT}) not is *visibleVars*

30 GenerateVarDeclaration(n_{CT}, E)

31 For each child n_C of n_{CT}, ApplyConditionSubTree(n_C, E)

32 If n_{CT} is an AND node

33 For each child n_C of n_{AND}, ApplyConditionSubTree(n_C, E)

34 Add conjunctively to WHERE/SATISFIES clause of E the Boolean expression labeling n_{AND}

35 If n_{CT} is an OR node, not denoting an optional element

36 For each child AND node n_{AND}

37 Create a new SOME...SATISFIES expression E

38 For each child n_C of n_{AND}, ApplyConditionSubTree(n_C, E)

39 Add conjunctively to SATISFIES clause of *e'* the Boolean expression labeling n_{AND}

40 Add conjunctively to WHERE/SATISFIES clause of E the disjunction of SOME...SATISFIES expressions

GenerateVarDeclaration(n_{CT}, E)

41 If n_{CT} is an element node, where *name*(n_{CT}) is a constant

42 Add to FOR/SOME clause of E: "*var*(n_{CT}) IN *var*(*parent*(n_{CT}))/*name*(n_{CT})"

43 If n_{CT} is an element node, where *name*(n_{CT}) is a name variable

44 Add to FOR/SOME clause of E: "*var*(n_{CT}) IN *var*(*parent*(n_{CT}))/name()"   *// XPath's* name() *function* [54]

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.

```
<html>
  <body>
    <table>{
    FOR $doc IN accessSource('XMLDB'),
        $S IN $doc/sensors,
        $MAN IN $S/manufacturer,
        $PROD IN $MAN/product,
        $SPEC IN $PROD/specs,
        $PROTS IN $SPEC/protection_ratings,
        $PROT1 IN $PROTS/protection_rating,
        $PART IN $PROD/part_number,
        $DIST IN $SPEC/sensing_distance,
        $BODY IN $SPEC/body_type,
        $N_BODY IN $BODY/name()
        $PROT IN $PROTS/protection_rating,
        $NAME IN $MAN/name,
    WHERE
        $PROT1 = "NEMA3"
        AND ((SOME $CYL IN $BODY/cylindrical,
                   $DIA IN $CYL/diameter,
                   $BAR IN $CYL/barrel_style
              SATISFIES
                   $DIA <= 20 AND $DIA <= 40)
```

```
            OR
           (SOME $REC IN $BODY/rectangular,
                 $HEI IN $REC/height,
                 $WID IN $REC/width
            SATISFIES
                 $HEI <= 20 AND $WID <= 40))
ORDER BY $NAME DESCENDING, $DIST
RETURN
    GROUPBY $PROD, $NAME, $DIST 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>,
        {
        GROUPBY $PART AS
        <td>{$PART}</td>
        },
        <td>{
            <table>{
                GROUPBY $PROT AS
                <tr>{
                    <td>{$PROT}</td>
                }</tr>
            }</table>
        }</td>,
        <td>{$DIST}</td>,
        <td>{
            <table>{
                <tr>{
                    GROUPBY $N_BODY AS
                    <td>{$N_BODY}</td>
                }</tr>,
                <tr>{
                    <td>{
                    FOR $CYL IN $BODY/cylindrical,
                        $DIA IN $CYL/diameter,
                        $BAR IN $CYL/barrel_style
                    WHERE
                        $DIA <= 20 AND $DIA <= 40
                    RETURN
                        GROUPBY $CYL AS
                        <table>{
                            <tr>{
                                GROUPBY $DIA AS
                                <td>{$DIA}</td>,
                                GROUPBY $BAR AS
                                <td>{$BAR}</td>
                            }</tr>
                        }</table>
                    }</td>,
                    <td>{
                    FOR $REC IN $BODY/rectangular,
                        $HEI IN $REC/height,
                        $WID IN $REC/width
                    WHERE
                        $HEI <= 20 AND $WID <= 40
                    RETURN
                        GROUPBY $REC AS
                        <table>{
                            <tr>{
                                GROUPBY $HEI AS
                                <td>{$HEI}</td>,
                                GROUPBY $WID AS
                                <td>{$WID}</td>
                            }</tr>
                        }</table>
                    }</td>
```

```
                    }</tr>
                }</table>
            }</td>
        }</tr>
    }</table>
  </body>
</html>
```

Appendix B.   Example *QSS* Describing a Join

The syntax and semantics of the Tree Query Language (TQL), presented in Section 4, allow joins and, correspondingly, *QSS* allows the description of queries that involve joins. Such a case is illustrated by the *QSS* of Figure 26 that corresponds to a query form that allows the end-user to find sensors that have "Sensing Distance" that is not the minimum sensing distance offered by a particular manufacturer, i.e., there is at least one sensor of the same manufacturer that has a lower sensing distance.

Condition fragment f_2 joins the two `manufacturer` element nodes on their `name` element node and applies the condition on their corresponding `sensing_distance` element nodes.


(a) Condition Tree Generator   (b) Result Tree Generator
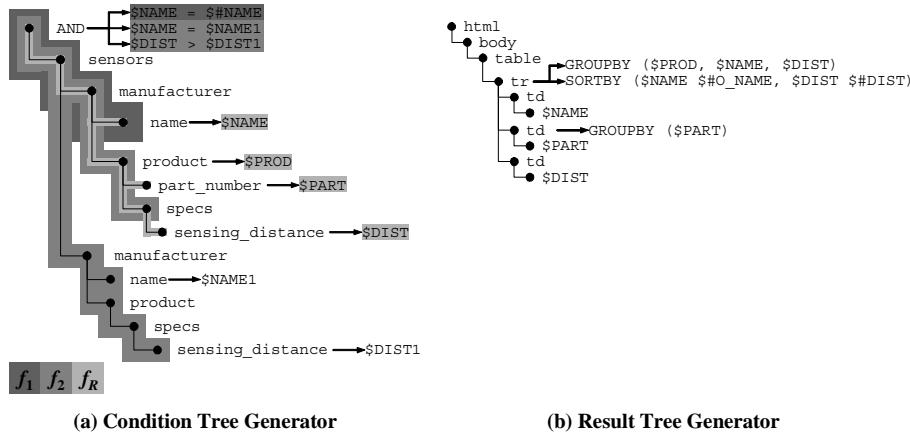
Figure 26 Example *QSS* Describing a Join

Appendix C.   `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):

```
        Expr            ::= Expr 'SORTBY' '(' SortSpecList ')'
                          | UnaryOp Expr
                          | Expr BinaryOp Expr
                          | Variable
                          | Literal
                          | '.'
                          | FunctionName '(' ExprList? ')'
                          | ElementConstructor
                          | '(' Expr ')'
                          | '[' ExprList? ']'
                          | PathExpr
                          | Expr Predicate
                          | FlwrExpr
                          | 'IF' Expr 'THEN' Expr 'ELSE' Expr
                          | ('SOME' | 'EVERY') Variable 'IN' Expr 'SATISFIES' Expr
                          | ('CAST' | 'TREAT') 'AS' Datatype '(' Expr ')'
                          | Expr 'INSTANCEOF' Datatype
                          | GroupBy
```

51

```
                    /*********  new *******/
    GroupBy      ::= 'GROUPBY' VarList?  HavingClause? 'AS' Expr
    /*********  new *******/
    VarList      ::= Variable (',' VarList)?
    /*********  new *******/
    HavingClause ::= 'HAVING' Expr
    /*********  new *******/
```

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
        <titleYear>
            $t,
            <year> $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:

```
FOR $a IN distinct(document("http://www.bn.com")/bib/book/author)
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 </y> </Tup>),
                $t IN $Tup/t/node(),
                $y IN $Tup/y/node()
        RETURN <titleYear>
                $t,
                <year> $y </year>
            </titleYear>
    </result>
```