Graphical Query Interfaces for Semistructured Data: The QURSED System

MICHALIS PETROPOULOS, YANNIS PAPAKONSTANTINOU
University of California, San Diego
and
VASILIS VASSALOS
Athens University of Economics and Business, Athens, Greece

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 online demonstration of the system is available at http://www.db.ucsd.edu/qursed/.

Categories and Subject Descriptors: D.1.7 [**Programming Techniques**]: Visual Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; *Tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*; H.2.3 [**Database Management**]: Languages—*Query languages*; *Report writers*; H.2.5 [**Database Management**]: Heterogeneous Databases; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*; *Query formulation*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces (GUI)*; *Interaction styles*

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, that is, 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 [Abiteboul et al. 2000] that evolved into XML-based data models [Fernandez et al. 2005]. XML query languages (with the most notable the emerging XQuery standard [Boag et al. 2005]), XML databases [Schöning and Wäsch 2000], and mediators [Cluet et al. 1998; Draper et al. 2001; Fernandez et al. 2001; Ludäscher et al. 2000; Shanmugasundaram et al. 2000] 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 [Fernandez et al. 2001; Shanmugasundaram et al. 2000], 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 [Bernstein et al. 1998; Silberschatz et al. 1990]—albeit semineglected by the database research community. They allow millions of Web users to selectively view the information of underlying sources. A number of tools (Dreamweaver UltraDev [Macromedia 2003]; ColdFusion [Macromedia 2003]; Visual InterDev [Microsoft 2003]) facilitate the development of Web-based query forms and reports that access relational databases. However, these tools are tied to the relational model which limits the resulting user experience and impedes the developer in his efforts to quickly and cleanly produce Web-based query forms and reports. QURSED is, to the best of our knowledge, the first Web-based query forms and reports generator with focus on semistructured XML data.

QURSED produces query form and report pages that are called *QFR*s. A *QFR* is associated with a *Query Set Specification* (*QSS*). A *QSS* describes formally the complex query and reporting capabilities [Vassalos and Papakonstantinou 2000] 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
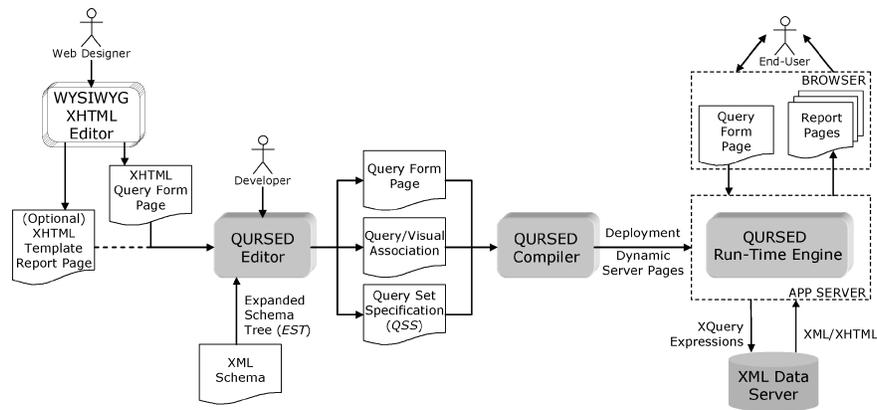
Fig. 1.   The QURSED system architecture.

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 next discuss 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 article. QURSED consists of the *QURSED Editor*, which is the design-time component, the *QURSED Compiler*, and the *QURSED Run-Time Engine*.

The Editor inputs the XML Schema that describes the structure of the XML data to be queried and constructs an *Expanded Schema Tree* (*EST*) out of it. The *EST* is a structure that serves as the basis for building the query set specifica-tion 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 [Raggett et al. 1999], 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 ap-pear 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 HomeSite [Macromedia 2003]. 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 asso-ciation. 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 for-malism of the *Tree 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*. 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* [Raggett et al. 1999] 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 JavaServer Pages [Sun Microsystems 2005], while Active Server Pages such as ASP.NET [Microsoft 2005] are another possible option. The dynamic server pages, the query set specification, and the query/visual association are inputs to the *QURSED Runtime Engine*. In particular, the dynamic server pages enforce the dependencies between the visual fragments on the query form page and handle the navigation on the report page. The engine, based on the query set specification and the query/visual association, generates an XQuery expression when the end-user clicks "Execute", which is sent to the XML Data Server, and its XHTML result is displayed on the report page.

The rest of the article 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.

2. RELATED WORK AND NOVEL CONTRIBUTIONS OF QURSED

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

(1) *Web-Based Form and Report Generators*, such as Dreamweaver UltraDev [Macromedia 2003] ColdFusion [Macromedia 2003], and]. Visual Inter-Dev [Microsoft 2003]. All of the these enable the development of Web-based applications that create form and report pages that access relational databases with the exception of XQForms [Petropoulos et al. 2001] 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 [Zloof 1975] and Microsoft's Query Builder (part of Visual InterDev [Microsoft 2003]) which target relational databases, and XML-GL [Comai et al. 2001], EquiX [Cohen et al. 1999], BBQ [Munroe and Papakonstantinou 2000], VQBD [Chawathe et al.

2001], the Lorel's DataGuide-driven GUI [Goldman and Widom 1998], and PESTO [Carey et al. 1996] which target XML or object-oriented databases.

(3) *Schema Mapping Tools* such as IBM's Clio [Popa et al. 2002], Microsoft's BizTalk Mapper [Microsoft 2004], XML Transform [TIBCO 2005], and BEA's Data View Builder [BEA 2005]. 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 [Boag et al. 2005] or XSLT [Kay 2005] 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 [Fraternali and Paolini 2000], Araneus [Atzeni et al. 1997], Strudel [Fernandez et al. 2000], and Application Manifold [Eyal and Milo 2001]. 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 Fraternali [1999].

Web-based form and report generators create Web-based interfaces that access relational databases. Popular examples are Dreamweaver UltraDev [Macromedia 2003], ColdFusion [Macromedia 2003], and Visual InterDev [Microsoft 2003]. The developer uses a set of wizards to visually explore the tables and views defined in a relational database schema and selects the one(s) she wants to query using a query form page. By dragging and dropping the attributes of the desired table to XHTML form controls [Raggett et al. 1999] 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 and dropping the desired attributes to XHTML elements on the page, for example, 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, for example, 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 JavaServer Pages [Sun Microsystems 2005], Active Server Pages [Microsoft 2005] and ColdFusion Markup Language [Macromedia 2003].

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

Visual querying interfaces are applications that allow the exploration of the schema and/or content of the underlying database and the formulation of queries. Typical examples are the Query-By-Example (QBE) [Zloof 1975] interface and Microsoft's Query Builder [Microsoft 2003] which target the querying of relational databases. Recent visual frontends such as XML-GL

[Comai et al. 2001], EquiX [Cohen et al. 1999], BBQ [Munroe and Papakonstantinou 2000], VQBD [Chawathe et al. 2001], the Lorel's DataGuide-driven GUI [Goldman and Widom 1998], and PESTO [Carey et al. 1996] target the querying of XML and object-oriented databases. Unlike the form and report generators which produce Web frontends for the general public, visual querying interfaces present the schema of the underlying database to experienced users who are often developers building a query, helping them formulate queries visually and displaying the results 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 [Cohen et al. 1999] allows the visual development of complex XML queries that include quantification, negation, and aggregation. EquiX and BBQ [Munroe and Papakonstantinou 2000] use some form of the *EST* and of the corresponding visual concept, but they still require basic knowledge of query language primitives. Simple predicates, Boolean expressions, and variables can be typed at terminal nodes and quantifiers can be applied to nonterminal 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, that is, there is no restructuring ability.

A more powerful visual query language is XML-GL [Comai et al. 2001] 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 [Boag et al. 2005] though since recursive queries are not expressible and nested subqueries are partially supported. An advantage of XML-GL is that it can be implemented as a visual frontend to an XQuery processor since the correspondence between their semantics is straightforward. The disadvantage of XML-GL is that it doesn't make the common case easy. The interface is not intuitive for simple queries until the developer gets familiar with the visual semantics of the language.

It is important to note that the described visual query formulation tools and the Editor have very different goals. The goal of the former is the development of a query or a query template by a database programmer who is familiar with database models and languages. The goal of the latter is the construction from an average Web developer of a form that represents and can generate a large number of possible queries.

Schema mapping tools are graphical user interfaces that declaratively transform data between XML Schemas in the context of integration applications.

IBM's Clio [Popa et al. 2002], Microsoft's BizTalk Mapper [Microsoft 2004], XML Transform [TIBCO 2005], and BEA's Data View Builder [BEA 2005] are representative examples. 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 the 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 into 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 [Popa et al. 2002] 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 [Boag et al. 2005] or an XSLT [Kay 2005] 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 the fact that the generated query expressions are particularly complex and hard to write by hand is taken into account.

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 such as Autoweb [Fraternali and Paolini 2000], Araneus [Atzeni et al. 1997], and Strudel [Fernandez et al. 2000] are excellent examples of the ongoing research on how to design and develop Web sites that are 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 the 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 [Fernandez et al. 2000], 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, Fraternali and Paolini [2000] Autoweb and Araneus [Atzeni et al. 1997] 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 changed, 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 [Eyal and Milo 2001] 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.

Also related to QURSED is prior work on capability-description languages and their use in mediator systems [Levy et al. 1996; Vassalos and Papakonstantinou 2000]. 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. The *QSS* instead 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 [Petropoulos et al. 2001]. 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 XForms W3C standard [Dubinko et al. 2003] 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. QURSED can 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 need for interacting with and presenting semistructured data. Multiple features contribute to this need.

(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.

Loose Coupling of Query and Visual Aspects. QURSED separates the logical aspects of query forms and reports generation, that is, 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: What 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 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; and

(2) provide a tree-based query model that easily captures 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 [Boag et al. 2005] and the interplay of XML/XHTML. The result is an efficient overall system when compared to either relational-based frontend generators or to conventional to XML-based frontend architectures such as XSQL [Oracle 2004]. An XML-related efficiency is derived by the fact that XML is used throughout QURSED: XML is the data model of the source on which XML queries, in XQuery syntax, are evaluated, and it 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 (*lotos*) such as the sample data set shown in Figure 2(a) that describes two proximity sensor products. Each internal node of the labeled ordered tree represents an XML element and is labeled with the element's tag name. The list of children of a node represents the sequence of elements that make up the content of the element. A leaf node holds the string value of its parent node. If n is a node of a *loto*, we denote as *tree(n)* the subtree rooted at n.

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

(a) Data Set (*loto*)

(b) XML Schema

(c) Expanded Schema Tree (*EST*)

(d) Expanded Schema Tree (*EST*)
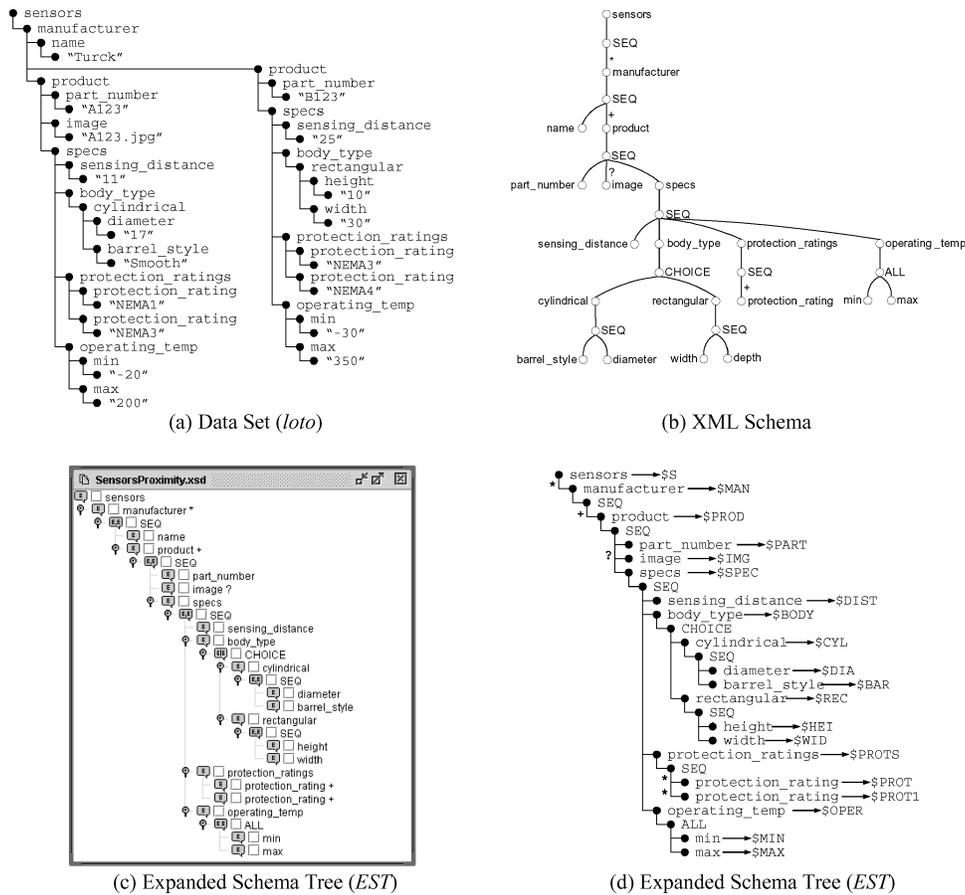
Fig. 2.   Example data set, XML Schema and expanded schema tree.

The XML Schema that describes the structure of the sample data set of Figure 2(a) is shown as a tree structure in Figure 2(b). Similar conventions for representing XML Schemas and DTDs have been used in previous works, for example Abiteboul et al. [2001] and Fernandez et al. [2001]. The optional (? and ∗ labeled edges) and repeatable (∗ and + labeled edges) elements and the types of groups of elements (SEQ, CHOICE, and ALL nodes [Fallside and Walmsley 2004]) are indicated. The leaf nodes are of primitive type [Biron and Malhotra 2004]. Like many XML Schemas, it has nesting and many "irregular" structures such as choice groups, for example the `body_type` may be `rectangular` or `cylindrical`, and optional elements [Fallside and Walmsley 2004], for example each sensor can optionally have an `image` element.

Based on the XML Schema in Figure 2(b), the Editor constructs the corresponding *EST* that serves as the basis for building the query set specification. Figure 2(c) shows the Editor's view of the *EST* as it is displayed to the developer and Figure 2(d) the internal representation used by the Editor. Formally, the *EST* is defined in the following.

Definition 3.1. 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 constrain *occ(n)* which can be ? (0–1 occurrences), 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;

—CHOICE nodes;

—ALL nodes.

   The root node of an *EST* is a nonrepeatable element node.

   The Boolean property *report* of an element node is true if the corresponding checkbox that appears next to the element node on the view of the *EST* (Figure 2(c)) is checked. The reason for doing that is to indicate to the Editor which elements to include on the report. Report generation is described in Section 7.3.

3.2 Example QFR and End-User Experience

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 2(c). The *EST* of Figure 2(d) 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 only 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.

3.3 *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 proximity sensor products. This interface will be the running example and will illustrate the basic points of the functionality and the experience that QURSED delivers to the end-user of the interface.

   The browser window displays a query form page and a report page. On the query form page, 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:

Fig. 3.   Example *QFR* interface.

—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" as being less than 20mm and on "Dimension Y" as less than 40mm. 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, for example, "Part Number" is, by default, always presented (disabled checkbox).

After the end-user submits the form, she receives the report of Figure 3. The results depict the information of product elements: the developer had decided earlier that 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
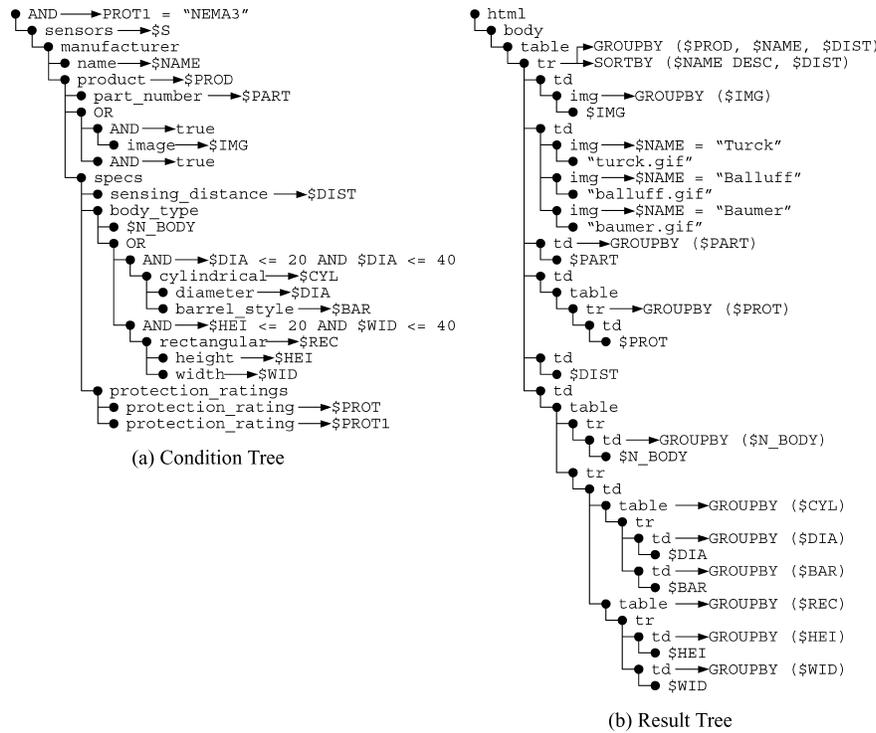
```
● AND ──►PROT1 = "NEMA3"
  ● sensors ──►$S
    ● manufacturer
      ● name──►$NAME
      ● product ──►$PROD
      ● part_number ──►$PART
      ● OR
        ● AND──►true
          ● image──►$IMG
        ● AND──►true
    ● specs
      ● sensing_distance ──►$DIST
      ● body_type
        ● $N_BODY
        ● OR
          ● AND──►$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

Fig. 4.   TQL query corresponding to Figure 3.

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 [Deutsch et al. 1998], XML-GL [Comai et al. 2001], LOREL [Quass et al. 1995], XMAS [Ludäscher et al. 2000] and XQuery [Boag et al. 2005]. TQL facilitates the development of query set specifications that encode large numbers of queries and the development of a visual interface for the easy construction of those specifications. This section describes the structure and semantics of TQL queries. The structure and semantics of query set specifications are described in the next section.

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

Definition 4.1. (Condition Tree). The condition tree of a TQL query q is a labeled tree that consists of:

—element nodes n having an element name *name(n)* which is a constant or a name variable, and an element variable *var(n)*. In a condition tree, there can be multiple nodes with the same constant element name but element and name variables must be unique. Element variables start with the $ symbol and name variables start with the $N_;

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

—OR nodes.

The following constraints apply to condition trees:

(1) The root element node of a condition tree is an AND node.
(2) OR nodes have AND nodes as children.

Figure 4 shows the TQL query for the example of Figure 3. Note that two conditions are placed on diameter of cylindrical sensors corresponding to the height and width of rectangular sensors. Omitted are the variables that are not used in the condition or the result tree.

The semantics of condition trees is defined in two steps, OR-removal and binding generation. OR-removal is the process of transforming a condition tree with OR nodes into a forest of condition trees without OR nodes, called *conjunctive condition trees* in the remainder of the article. OR-removal for the condition tree of Figure 4(a) results in the set of the four condition trees shown in Figure 5.

Intuitively, OR-removal is analogous to turning a logical expression to disjunctive normal form [Genesereth and Nilsson 1987]. 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 where we can no longer apply the rules, we have produced a tree with an OR root node which we replace with the forest of conjunctive condition trees, consisting of all the children of the root OR node. Notice that wherever this process generates AND nodes as children of AND nodes, these can merged, and the Boolean expression of the merged node is the conjunction of the Boolean expressions of the original AND nodes. Also notice that the Boolean expression of the root AND node in the first rule cannot contain any variables in subtrees B or C per the earlier definition of condition trees. Finally, notice that in the course of OR-removal, intermediate results may not be valid condition trees by Definition 4 (in particular, constraint 2 can be violated), but the final results obviously are. The
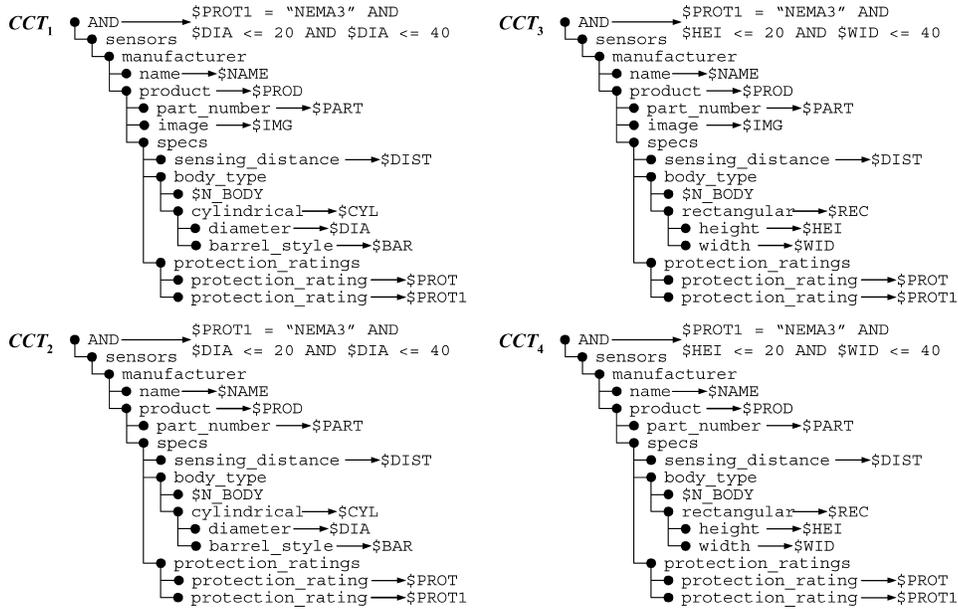
Fig. 5.　Conjunctive condition trees.
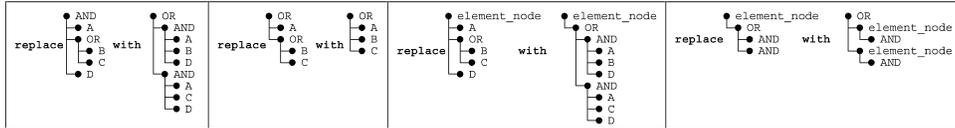
Fig. 6.　OR-removal replacement rules.

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 with the root of t, that is, $\beta(var(child(root(C)))) = root(t)$, and recursively, traversing the two trees top-down, for each child n_i of an element node n in C, assuming $var(n)$ is mapped to a node x in t, there exists a child x_i of x, such that $\beta(var(n_i)) = x_i$ and, if x_i is not a leaf node:

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

Importantly, AND notes in C are ignored in the traversal of C. In particular, in the previous definition, by "child of the element", we mean either element child of the element, or the child of an AND node that is the child of the element. A binding is *qualified* if it makes *true* the Boolean expressions that label the AND nodes of C. Notice that it is easy to do AND-removal on conjunctive condition

Table I. 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

trees. Let $a_1, \ldots, a_n$ be the AND nodes in a *CCT* with root a, and let $b_1, \ldots, b_n$, and b be their Boolean expressions. We can eliminate $a_1, \ldots, a_n$, and replace b with b *AND* b_1 *and* $\ldots$ *and* b_n.

The result of C is the set of qualified bindings. For a conjunctive condition tree with element and name variables $\$V_1, \ldots, \$V_k$, a binding is represented as a tuple $[\$V_1 : v_1, \ldots, \$V_k : v_k]$ that binds $\$V_i$ to node or value v_i, where $1 \le i \le k$. A binding of some of the variables in a (conjunctive) condition tree is called a *partial* binding. Note that the semantics of a binding requires total tuple assignment [Quass et al. 1995], that is, every variable binds to a node or a string value.

The semantics of a condition tree is defined as the union of the bindings returned from each of the conjunctive condition trees in which it is transformed by OR-removal. For example, the result of the four conjunctive condition trees shown in Figure 5 on the source *loto* of Figure 2(a) is shown in Table I. 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 subsumed by CCT_1 rows in Table I.) 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.

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

The condition tree corresponds intuitively to the WHERE part of XML query languages such as [Deutsch et al. 1998] XML-QL, LOREL, [Quass et al. 1995] and XMAS, [Boag et al. 2005] to the *extract* and *match* parts of XML-GL [Comai et al. 2001], and to the FOR and WHERE clauses of a FLWOR expression of XQuery [Boag et al. 2005]. The result tree correspondingly maps to the CONSTRUCT clause of XML-QL and XMAS, the SELECT clause of LOREL, the *clip* and *construct* parts of XML-GL, and the RETURN clause of a FLWOR expression of XQuery. A result tree specifies how to build new XML elements using the bindings provided by the condition tree.

Definition 4.2.   (Result Tree). A result tree of a TQL query q is a node-labeled tree that consists of:

—element nodes n having an element name *name*(n) which is a constant if n is an internal node, and a constant or a variable that appears in the condition tree of q f n is a leaf node;

—a group-by label G and a sort-by label S on each node. A group-by label G is a (possibly empty) list of variables $[\$V_1, \ldots, \$V_n]$ from the condition tree of q,, A sort-by label 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 zis the sorting order determined for $\$V_i$. O_i can take the values "DESC" for descending or "ASC" for ascending order. Each variable in the sort-by list of a node must appear in the group-by list of the same node. Empty group-by and sort-by labels are omitted from figures in the remainder of the article;

—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 4(b) 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 labels are the TQL means of performing grouping and sorting. The intuition behind Boolean expressions on nodes is that they provide control on the construction of nodes in the result of a query: A node (and its subtree) is only added to the result of the query if there is at least one qualified binding of the variables in the condition for that node that renders it true.

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, formally described in the following. The recursion uses partial bindings to instantiate the group-by variables and condition variables of element nodes.

Traversing the result tree top-down, for each subtree *tree*(n) rooted at element node n with group-by label $[\$V_1, \ldots, \$V_k]$ and, without loss of generality, sort-by label $[\$V_1, \ldots, \$V_m]$ $(m \leq k)$, 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
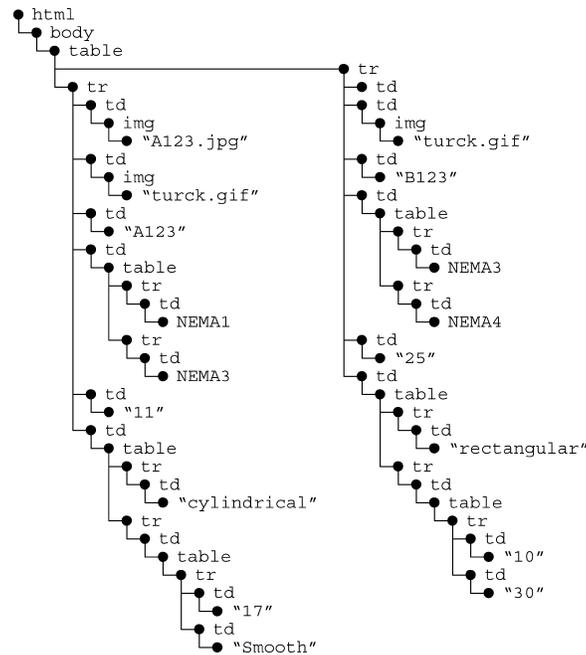
Fig. 7.   Resulting *loto* for bindings of Table I.

for some qualified binding $\pi' = [\$V_{A1} : v_{A1}, \ldots, \$V_{An} : v_{An}, \$V_1 : v_1, \ldots, \$V_k : v_k, \$B_1 : b_1, \ldots, \$B_j : b_j]$ (otherwise the subtree is not included in the list of subtrees produced.) The list of instantiated subtrees is ordered according to the conditions in the sort-by label.

Figure 7 shows the resulting *loto* from the TQL query of Figure 4 and the bindings of Table I. 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 4(b), only the first one evaluates to true for both sensors, based on the bindings of variable $NAME in Table I.

The QURSED system uses the TQL queries internally but issues queries in the 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 Boag et al. [2005] and Draper et al. [2005].

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.
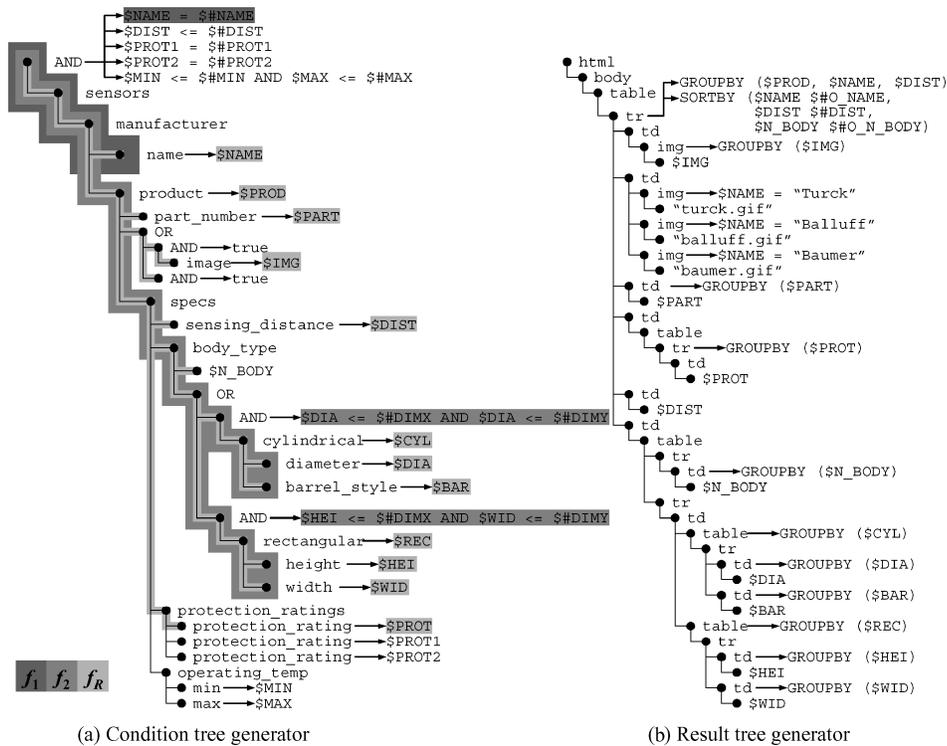
```
$NAME = $#NAME
$DIST <= $#DIST
$PROT1 = $#PROT1
AND      $PROT2 = $#PROT2
$MIN <= $#MIN AND $MAX <= $#MAX
sensors
   manufacturer
      name      $NAME
      product      $PROD
      part_number      $PART
      OR
         AND      true
            image      $IMG
         AND      true
      specs
         sensing_distance      $DIST
         body_type
            $N_BODY
            OR
               AND      $DIA <= $#DIMX AND $DIA <= $#DIMY
                  cylindrical      $CYL
                  diameter      $DIA
                  barrel_style      $BAR
               AND      $HEI <= $#DIMX AND $WID <= $#DIMY
                  rectangular      $REC
                  height      $HEI
                  width      $WID
      protection_ratings
         protection_rating      $PROT
         protection_rating      $PROT1
         protection_rating      $PROT2
      operating_temp
         min      $MIN
         max      $MAX

f₁ f₂ f_R
```

(a) Condition tree generator

```
html
  body
    table      GROUPBY ($PROD, $NAME, $DIST)
               SORTBY ($NAME $#O_NAME,
                       $DIST $#DIST,
                       $N_BODY $#O_N_BODY)
      tr
        td
          img      GROUPBY ($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 generator

Fig. 8.   Query set specification.

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 query set specification can describe a number of queries that is exponential in the size of the specification. The specification also includes a set of dependencies that constrain the set of queries that can be produced.

The developer uses the Editor to visually create a query set specification like the one in Figure 8. This section formally presents the query set specification that is the logical underpinning of *QFR*s.

Definition 5.1.   (Query Set Specification). A query set specification, *QSS*, is a 4-tuple ⟨ *CTG*, *RTG*, *F*, *D* ⟩, where:

—*CTG*, the *condition tree generator*, is a condition tree with three modifications:

• AND nodes a_i can be labeled with a set of Boolean expressions $B(a_i)$,

• the same element or name variable can appear in more than one *condition fragments*,

• Boolean expressions can use *parameters*, also known as placeholders [Levy et al. 1996], as operands of their predicates. Parameters are denoted by the $# symbol and must bind to a value [Biron and Malhotra 2004].

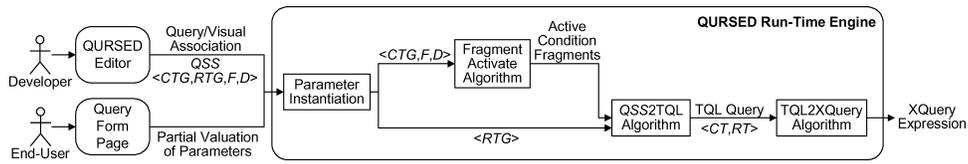The same constraints apply to a *CTG* as to a condition tree:

Fig. 9.   Query formulation process.

—*RTG*, the *result tree generator*, is a result tree with two modifications. First, the variables that appear in the sort-by label S on a node do not have a specified order (ascending or descending,) as in the case of a result tree but they have a parameter instead, called *ordering parameter*, that starts with the $\$\#0\_$. Second, the Boolean expressions on nodes can use parameters as operands of their predicates. Boolean expressions on nodes involving only parameters and constants as operands (no variables) are a special case since they can be evaluated as soon as the parameters are instantiated. Their use is described later in Section 7.5.

—*F* is a nonempty 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 8(a) 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 8(b), 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 runtime 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.

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 v 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:

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

Based on v, 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 8(a) which imposes a condition on the dimensions of the sensor's body type. Similarly, the ordering parameters of the sort-by labels 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 partial valuation just presented. An example of an RTG where parameterized Boolean expressions are labeling 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 v. Since the partial valuation v might not provide values for all the parameters used in the CTG, some condition fragments are rendered *inactive*. Based on the previous example partial valuation, condition fragment f_2 of Figure 8(a) and the condition fragment that imposes a condition on protection rating (not indicated in Figure 8(a) are rendered active, while condition fragment f_1 on the manufacturer's name is inactive, since parameter \$#NAME is not instantiated by v. As a special case, the result fragment f_R is always active since it does not 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 $f_1, \ldots, f_n$, 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 label 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 4(a) which is formulated based on the active condition fragments of Figure 8(a), that is, f_2, the condition fragment that imposes a condition on protection rating, and the result fragment f_R. Accordingly, the *RT* of Figure 4(b) is formulated from the *RTG* of Figure 8(b) 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 6.1. (Dependency). A dependency d is defined as a 3-tuple $\langle f, B, H \rangle$ 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 10(a), and let us define two dependencies d_1 and d_2 as follows:

$$\langle f_2, \text{\$\#BODY} = \text{``cylindrical''}, \{f_1\} \rangle \tag{d_1}$$
$$\langle f_3, \text{\$\#BODY} = \text{``rectangular''}, \{f_1\} \rangle. \tag{d_2}$$
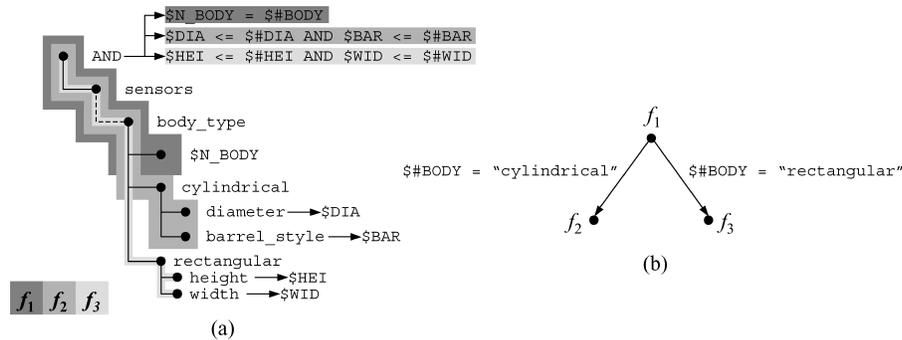
Fig. 10.   Condition tree generator and dependencies graph.

Fig. 11.   Dependencies on the query form page.

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

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 10(a). For instance, the condition fragment f_1 applies a condition to the element node labeled with $\$BODY$ and its visual fragment consists of the "Body Type" form control. End-user selection of the "Cylindrical" option in the "Body Type" form control results in having d_1 hold which makes the visual fragment for f_2 visible (Figure 11(a)). 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 6.2.   (Dependency Graph). A dependency graph for a set of dependencies D and a set of condition fragments F is a directed labeled graph $G = \langle V, E \rangle$, 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
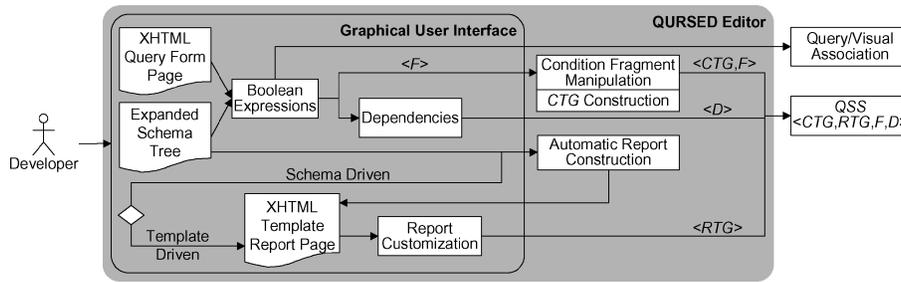
Fig. 12.   QURSED editor architecture.

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 10(b). 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 [Knuth 1973] (hence of complexity $\Theta(V + E)$) and is outlined in the following. 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 $G = \langle V, E \rangle$, and a partial valuation v over P,
where P is the set of the parameters that appear in the *QSS*.
Output: The set A of active condition fragments.
Method:
$A \leftarrow \oslash$	1
Compute the set of fragments B, whose parameters are all instantiated by v	2
For each edge (n, u) in E	3
Evaluate the condition on edge (n, u)	4
Repeat	5
If node u belongs to B and has no incoming edges	6
$A \leftarrow \{u\}$	7
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	8
$A \leftarrow \{u\}$	9
Until A reaches fixpoint	10

Section 7.2 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*.

(a) QURSED editor   (b) WYSIWYG HTML editor
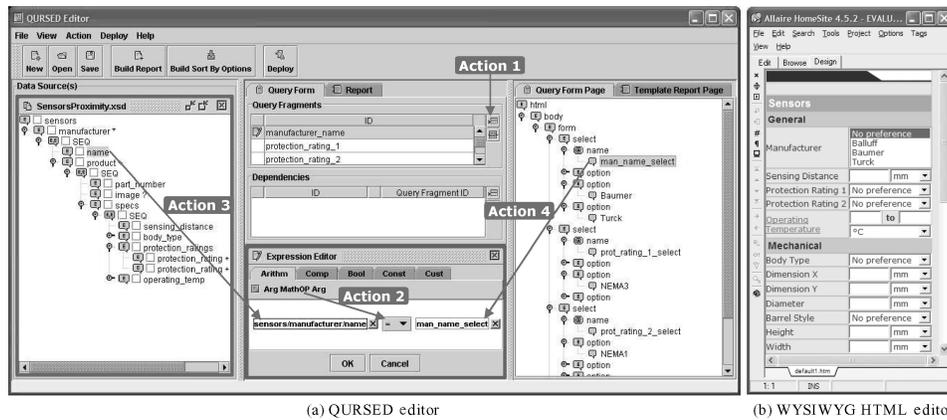
Fig. 13.   Building a condition fragment.

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 must check if f is meaningful by considering the presence of CHOICE elements in the *EST* and, if necessary, manipulating 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 7.2.

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 sections 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 13(a) demonstrates how the developer uses the Editor to define the condition fragment f_1 of Figure 8(a). The main window of the Editor presents the sample *EST* of Section 3.1 on the left panel and the query form page on 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, that is, `select` and `input` element

nodes [Raggett et al. 1999]. The XHTML tree corresponds to the page shown on Figure 13(b) rendered in the Site WYSIWYG XHTML editor [Macromedia 2003]. Based on this setting, the developer defines the condition fragment f_1 of Figure 8(a) that imposes an equality condition on the manufacturer's name by performing the four actions indicated by the arrows on Figure 13(a).

The developer starts by clicking on the "New Condition Fragment" button (Action 1 of Figure 13(a) 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 dragging and 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 13(b). Internally, the Editor creates the parameter `$#NAME`, associated with the "Manufacturer" form control of Figure 13(b), and sets it as the right operand of the Boolean expression as Figure 8(a) shows.

In order to build more complex condition fragments, Actions 2, 3, and 4 can be repeated multiple times, thus introducing multiple variables and parameters and including more than one XHTML form controls in the corresponding visual fragment.

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 contains 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 13(b), 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:

$(\$DIA \leq \$\#DIMX \wedge \$DIA \leq \$\#DIMY) \vee (\$HEI \leq \$\#DIMX \wedge \$WID \leq \$\#DIMY).$

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 consisting of 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 $\vee$ Boolean connective and treating it as an OR node of TQL as the condition fragment f_2 in Figure 8(a) indicates. The OR node corresponds to the CHOICE node in the *EST* of Figure 2(c). Two AND nodes are also introduced and are labeled with the conjunctions in the initial Boolean expression: ($\$DIA \leq \$\#DIMX \wedge \$DIA \leq \$\#DIMY$) and ($\$HEI \leq \$\#DIMX \wedge \$WID \leq \$\#DIMY$). 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 to f by replacing Boolean connectives $\vee$ in b with OR nodes as illustrated in the previously mentioned example. The manipulation is driven by the CHOICE nodes and optional elements. 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 $node(\$V_i)$ that, given a variable name $\$V_i$ in b, returns the node n_i of the *EST* that the variable corresponds to, that is, the node of the *EST* that the developer dragged and dropped. In the case of name variables, $node(\$V_i)$ returns the parent of the node that the developer dragged and dropped. 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, if there exists one, or *null*, otherwise.

Algorithm Construct*CTG*
Inputs: 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
Rewrite b in disjunctive normal form such that $b = c_1 \vee c_2 \ldots \vee c_n$, 1
where c_i is a conjunction of predicates
If a conjunction c_i, where $1 \leq i \leq n$, uses two variables $\$V_{ix}$, $\$V_{iy}$ 2
such that the lowest common ancestor of $node(\$V_{ix})$ and $node(\$V_{iy})$
in the *EST* is a CHOICE node
  Output an error indicating the unsatisfiable conjunctions 3
Step 2: Manipulation of f
// Introduces OR nodes to f based on CHOICE nodes in the *EST*
For any two variables $\$V_{ix}$, $\$V_{jy}$ used in conjunctions c_i and c_j of b, 4
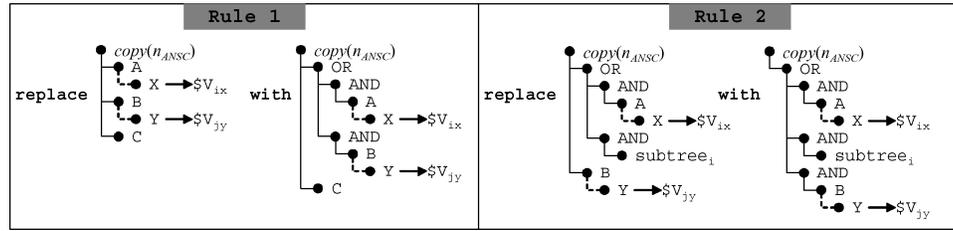respectively, where $1 \leq i, j \leq n$ and $i \neq j$

Fig. 14.   OR node introduction rules.

$\bar{\text{I}}$f both the paths from $node(\$V_{ix})$ and $node(\$V_{jy})$ 5
to their lowest element node common ancestor n_{ANSC} in the EST
contain either a CHOICE node or an optional element, excluding n_{ANSC}
    Apply the Rules 1 and 2 of Figure 14   6
// Label AND nodes with Boolean expressions
For each conjunction c_i of b, $1 \le i \le n$ 7
    $\bar{\text{I}}$n f, identify the lowest AND node a_i that is the common ancestor   8
of all the element nodes labeled with the variables used in c_i and
label it with Boolean expression c_i
If the AND node is labeled with more than one conjunctions   9
    Combine them with the $\vee$ Boolean connective   10
Step 3: Addition of f to the CTG
Set the children of the root AND node of f as children   11
of the root AND node of the CTG
Take the union of the sets of Boolean expressions labeling   12
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

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 ConstructCTG algorithm on the condition fragments defined on the EST of Figure 15(a). 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 ConstructCTG 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 15(d) 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
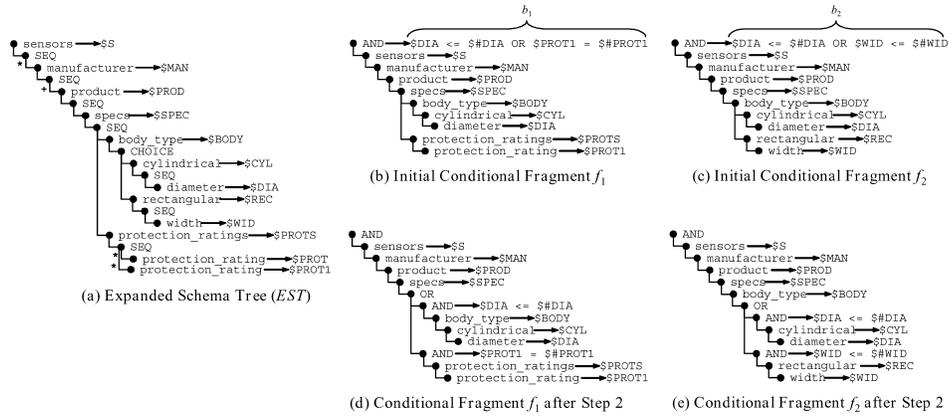
(a) Expanded Schema Tree (*EST*)

(b) Initial Conditional Fragment f_1

(c) Initial Conditional Fragment f_2

(d) Conditional Fragment f_1 after Step 2

(e) Conditional Fragment f_2 after Step 2
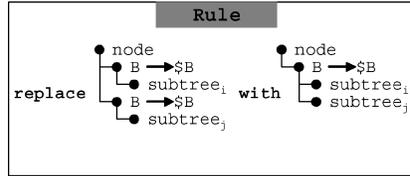
Fig. 15.　Example of the Construct*CTG* algorithm.

Fig. 16.　Node elimination rule.

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 17(a), where the two fragments are indicated in two different tones of gray.

7.1.2 *Eliminating Redundancies.*　The Editor eliminates redundancies on the merged *CTG* in order to improve the performance of the generated TQL queries. As shown in Amer-Yahia et al. [2001], 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.

The application of the rule takes time linear to 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.

The rule is eliminating redundancies particularly those introduced during the construction of the *CTG* as presented in the previous section. For example, the Construct*CTG* algorithm constructs the *CTG* of Figure 17(a) 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.

(a) Initial *CTG*

(b) *CTG after* node elimination rules
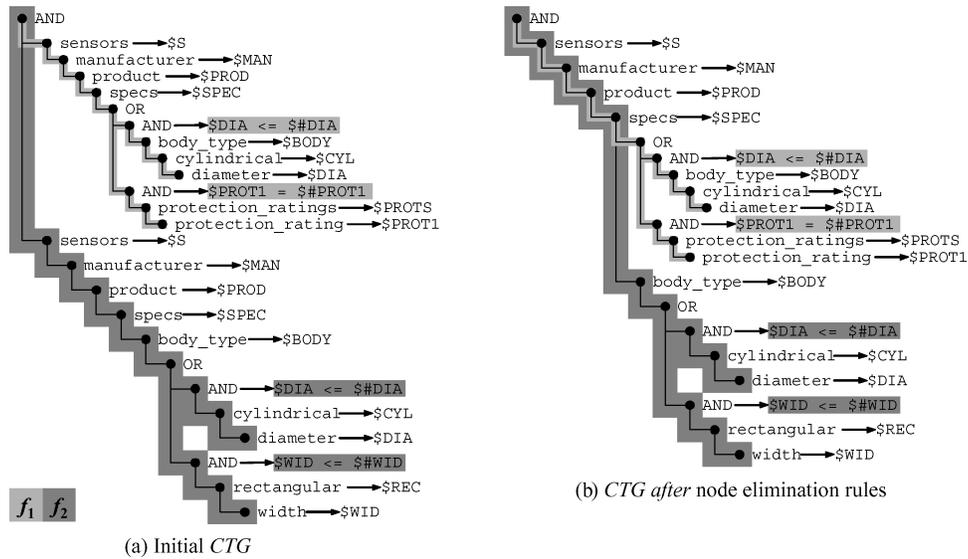
Fig. 17. Eliminating redundant nodes on the *CTG*.
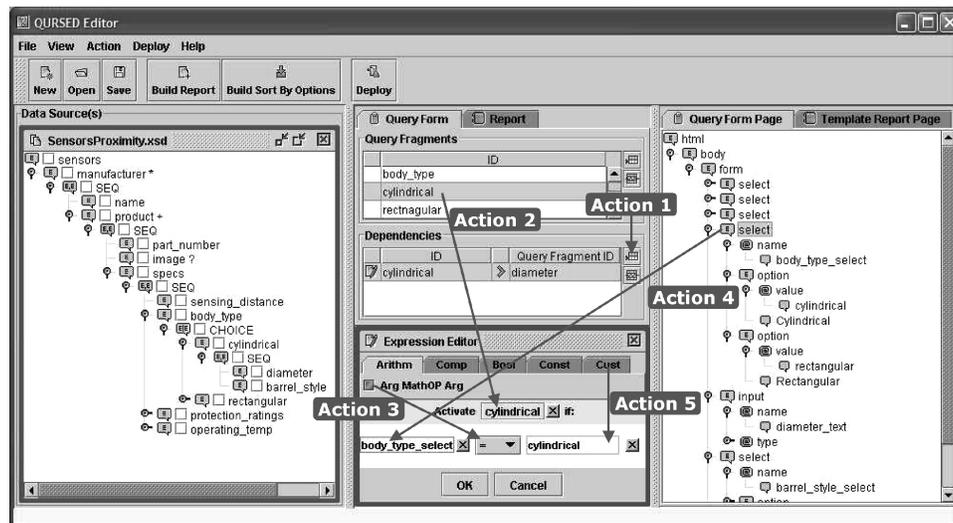
Fig. 18. Building dependencies.

The resulting *CTG* is shown in Figure 17(b). 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, that is, 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 $d_1 : \langle f_2, \$\#BODY=$"cylindrical", $\{f_1\}\rangle$ of Section 6.1

Fig. 19.   Schema-driven constructed report page.

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 10(a)) active if the parameter $#BODY is set to "cylindrical".

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 designates 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.

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.

   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
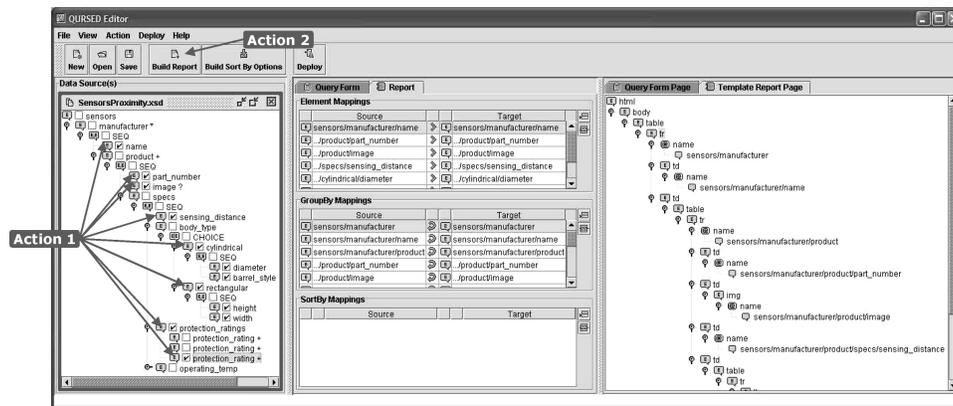
Fig. 20. Selecting elements nodes and constructing the template report page.

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 21(a). 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 21(c) shows how a WYSIWYG XHTML editor renders the template report page. The Editor translates the above actions into a *QSS* as follows.

In Action 2, the Editor automatically generates the result tree generator of Figure 21(b) 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 21(b) 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 21(c) 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 21(b), the element variable `$NAME` appears as the child of the `td` element node of the "manufacturer" table.

We discuss next how each type of semistructureness of the *EST* is 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 or is not present. Figure 21(a) demonstrates the effect of the visual action to select the optional element `image` to appear on the report page.
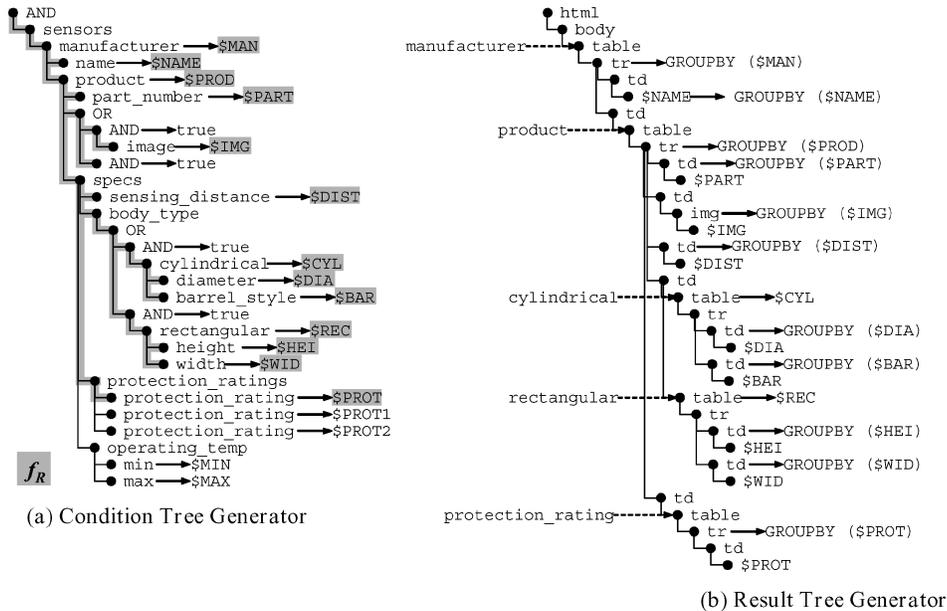
(a) Condition Tree Generator

(b) Result Tree Generator
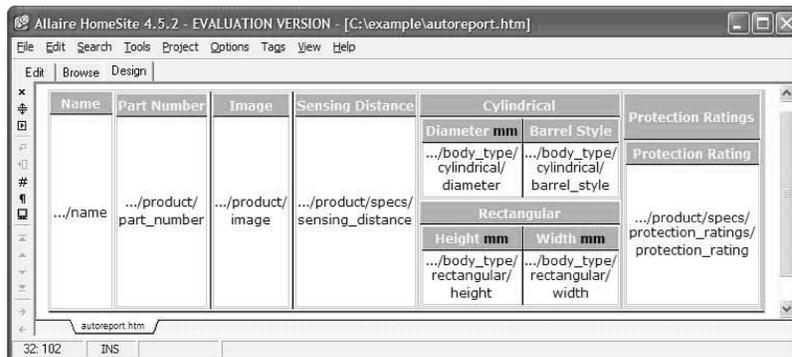
(c) Template report page

Fig. 21.   Automatically generated result fragment, *RTG*, and template 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 21(b), 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.

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, that is, 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 there exists one, or *null*, otherwise.

Algorithm AutoReport
Inputs: 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*
Traversing f_R top-down, for an element node n_i 1
  If n_i is labeled with a variable $\$V_i$ and $parent(node(\$V_i))$ is a   2
  CHOICE node and $parent(n_i)$ is not an OR node
    If there is a sibling n_j of n_i labeled with a variable $\$V_j$ such that   3
    $node(\$V_j)$ is a sibling of $node(\$V_i)$
      For all sibling element nodes n_j of n_i labeled with a variable $\$V_j$ 4
      such that $node(\$V_j)$ is a sibling of $node(\$V_i)$
        Apply Rule 1 of Figure 22   5
    Else   6
      Apply the Rule 2 of Figure 22 // Treat n_i as optional element   7
  If n_i is labeled with a variable $\$V_i$ and $node(\$V_i)$ is optional,   8
  or n_i is named with a variable $\$V_i$ and
  at least one child of $node(\$V_i)$ is optional
    Apply the Rules 2 and 3 of Figure 22 correspondingly   9
Step 2: Construction of the result tree generator *RTG*
Create a node n_r named "html", a node n_b named "body",   10
a node n_t named "table", and a node n_{tr} named "tr"
Set n_r as the root of the *RTG*, n_b as a child of n_r,   11
n_t as a child of n_b, and n_{tr} as a child of n_t
Traversing the *EST* top-down and left to right, ignoring SEQ,   12
CHOICE and ALL nodes, for an element node n_i
  BuildTable(n_i, n_{tr})   13

BuildTable(n_i, n_{tr})
If n_i is either repeatable or $parent(n_i)$ is a CHOICE node   14
  Create a node n_{td} named "td" and a node n_t named "table"   15
  Set n_{td} as a child of n_{tr} and n_t as a child of n_{td} 16
  Create a node named "tr" and set it as the current n_{tr} 17
  If $parent(n_i)$ is a CHOICE node   18
    Attach the Boolean expression $var(n_i)$ to n_t 19
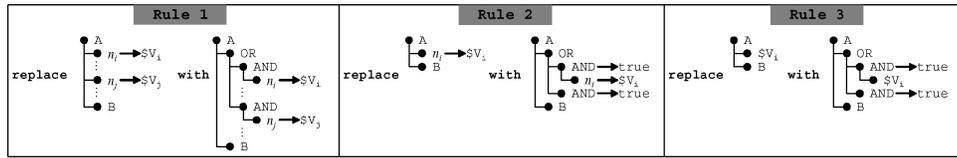
Fig. 22.   OR node introduction rules for result fragment f_R.
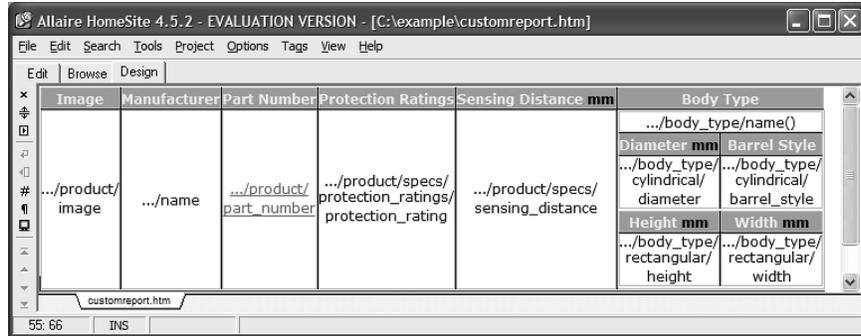
Fig. 23.   Editing the template report page.

If n_i is repeatable	20
Add $var(n_i)$ to the group-by list of n_{tr}	21
If n_i is a selected element node	22
Create a node n_{th} named "th" and add it as a child of n_{tr}	23
Create a node named $name(n_i)$ and add it as a child of n_{th}	24
If n_i is a leaf element node	25
Create a node named "td", add it as a child of n_{tr},	26
and set it as the current n_{td}	
Create a node named $var(n_i)$ and add it as a child of n_{td}	27
If $var(n_i)$ is not in any group-by list of an ancestor node	28
Add $var(n_i)$ to the group-by list of n_{td}	29
For every child element node n_c of n_i	30
BuildTable(n_c, n_{tr})	31

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 (see Figure 22).

7.3.2   *Template-Driven Construction of Result Tree Generator.*   The developer can create more sophisticated report pages and result tree generators by providing the Editor with 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 this, she constructs the template report page shown in Figure 23 and provides it to the Editor.

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
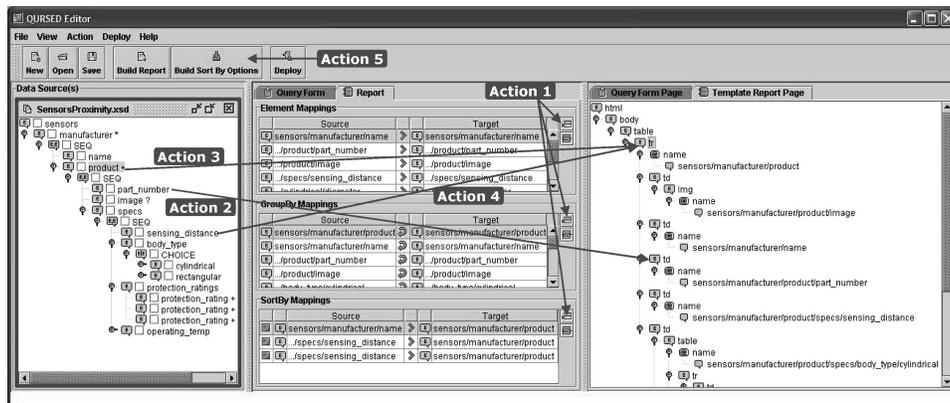
Fig. 24.   Performing element and group-by mappings on the template report page.

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 in 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.

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 8(b).

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 except 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 8(b), is generated. The Editor automatically defines a group-by mapping for each sort-by mapping if one does not exists. Note 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 Selections" list is initially empty. During run-time, the end-user can select any item from the "Sort by Options", select "ASC" or "DESC" order, and, using the "+" button, add it to the "Sort By Selections" list. When the end-user submits the query form, the corresponding ordering parameters are instantiated with the order the end-user selected as explained in the QSS2TQL algorithm in Section 6.

An engineering benefit from the way the developer builds the result tree generator is that the template report page can easily be opened from any external 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 ConstructCTG 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 the developer with a set of actions that is similar to the actions provided for the specification of dependencies as presented in Section 7.2. The setting of the Editor is the same as 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 7.2 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 during run-time.

—In Actions 4 and 5, the developer can not only specify parameters and constants as operands of the predicates in the Boolean expression, but also any variable by dragging any element node from the EST on the left panel.

The Boolean expressions that the developer defines on the template report page are listed in the "Boolean Expressions" table of the middle panel of Figure 24.

```
● 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
```

Fig. 25.   Result Tree Generator with Boolean expressions for dynamic projection.

Note that the Boolean expressions containing variables are translated to XQuery conditional expressions [Boag et al. 2005], according to TQL2XQuery algorithm in Appendix A. For example, the three Boolean expressions that label the `img` elements in Figure 4(b) 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 are labeled `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 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 is 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 determine the projection of "Part Number" and "Sensing Distance".

8. CONCLUSIONS

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

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 [Boag et al. 2005] but the draft includes an issue regarding an explicit GROUPBY construct. Such a construct is presented in Appendix B. The choice of XQuery augmented with GROUPBY expressions has been made because of the importance of grouping operations for producing nested XML and XHTML output. Explicit GROUPBY expressions enable easier optimization of such grouping operations as shown in Deutsch et al. [2004]. As Appendix B shows, XQuery+GROUPBY expressions can always be translated to XQuery expressions, often of significantly increased complexity. Their use results in cleaner query expressions and more opportunities for optimization but does not affect the generality of the algorithm.

TQL2XQuery inputs a result (sub)tree RT, rooted at n_{RT}, of a TQL query. The algorithm outputs an XQuery expression using nested FWOR (FOR–WHERE–ORDER BY–RETURN) expressions and element constructors where FWOR expressions are always nested in the RETURN clause of their parents. An FWOR expression e defines a scope s_e. It follows that scopes are nested. Every variable $\$V$ in the FOR clause of an FWOR expression e corresponds to a node n in CT as discussed in Section 7.1, and we write $n = node(\$V)$ and $\$V = var(n)$. We also write $scope(\$V) = scope(n) = s_e$ to denote the FWOR expression e that has $\$V$ in its scope. In the algorithm, we represent the current scope by S and the current FWOR expression by E. We define $allvars(S)$ to be the set of all the variables that are in S or in any scope that S is nested in, and we assume that the root N_{CT} of the CT is known to the algorithm.

Initially, the algorithm is called with TQL2XQuery($N_{RT}, \oslash, nil$). N_{RT} is the root of RT for the TQL query under translation. The initial scope is empty as is the initial FWOR expression.

> **Algorithm** TQL2XQuery
> **Inputs:** An RT node n_{RT}, the current scope S, and the current FWOR expression E.
> **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:
> Set V variables in the group-by list G of   1
> n_{RT} ←variables in the attached Boolean expression b of n_{RT}
> If S is empty and V is not empty then // Top-level group-by list   2
>   C̄reate a new FWOR expression e, with FOR clause: 3

"FOR $var(child(N_{CT}))$ IN $document$('source.xml')"

Set $E \leftarrow e, S \leftarrow s_e$ 4

$\bar{\text{F}}$or each child c_i of $child(N_{CT})$ 5

ApplyConditions(c_i, V, E, S)   6

$\bar{\text{A}}$dd conjunctively to WHERE clause of E the   7

Boolean expression labeling the root AND node N_{CT}

If there exists a variable $\$V_i \in V$ such that $\$V_i \notin allvars(S)$ 8

// $node(\$V_i)$ is under an OR node in CT

Create a new FWOR expression e, set $E \leftarrow e, S \leftarrow s_e$ 9

For every distinct variable $\$Vi \in V$ and $\$Vi \notin allvars(S)$ 10

GenerateResultVariable($\$V_i, E, S$)   11

If the group-by list G of n_{RT} is not empty   12

Add to RETURN clause of E the expression "GROUPBY G AS"   13

If n_{RT} has an attached Boolean expression b 14

Add to RETURN clause of E the expression "IF b THEN"   15

If $name(n_{RT})$ is a constant   16

Add to RETURN clause of E the expression <$name(n_{RT})$>   17

$\bar{\text{F}}$or each child c_i of n_{RT} 18

TQL2XQuery(c_i, S, E)   19

Add to RETURN clause of E the expression </$name(n_{RT})$>   20

Else if $name(n_{RT})$ is a variable   21

// then the node is guaranteed to be a leaf node,

// see Definition 4.2 in Section 4

Add to RETURN clause of E the expression "{$name(n_{RT})$}"   22

If the sort-by list S of n_{RT} is not empty   23

Add to ORDER BY clause of E the S list   24

ApplyConditions(n_{CT}, V, E, S)

// n_{CT} is the current node and V is the top-level group-by list

If n_{CT} is an OR node, not denoting optional element,   25

and no $var(n_D)$ is in V, where n_D is a descendant of n_{CT}

$\bar{\text{F}}$or each child AND node n_AND 26

Create a new SOME...SATISFIES expression e' 27

$\bar{\text{F}}$or each child c_i of n_{AND} 28

ApplyConditions($c_i, V, e', s_{e'}$)   29

$\bar{\text{A}}$dd conjunctively to SATISFIES clause of e' 30

the Boolean expression labeling n_{AND}

$\bar{\text{A}}$dd conjunctively to WHERE clause of E the disjunction   31

of all SOME...SATISFIES expressions

Else If n_{CT} is an OR node and there is a $var(n_D)$ is in V, 32

where n_D is a descendant of n_{CT}

$\bar{\text{F}}$or each child AND node n_{AND} having a $var(n_D)$ is in V,   33

where n_D is a descendant of n_{AND}

$\bar{\text{F}}$or each child c_i of n_{AND} 34

ApplyConditions(c_i, V, E, S)   35

$\bar{\text{A}}$dd conjunctively to WHERE clause of E 36

the Boolean expression labeling n_{AND}

Else If n_{CT} is an AND node   37

$\bar{\text{F}}$or each child c_i of n_{AND} 38

ApplyConditions(c_i, V, E, S) 39

$\bar{\text{A}}$dd conjunctively to WHERE clause of E 40

the Boolean expression labeling n_{AND}

Else   41

GenerateConditionVariable(n_{CT}, E, S)   42

$\bar{\text{F}}$or each child c_i of n_{CT} 43

ApplyConditions(c_i, V, E, S)   44

GenerateConditionVariable(n_{CT}, E, S)

If n_{CT} is an element node	45
Construct a path expression $pe = var(parent(n_{CT}))/name(n_{CT})$	46
If n_{CT} has a name variable	47
// refers to the XPath's `name()` function [Malhotra et al. 2005]	
Construct a path expression $pe = var(parent(n_{CT}))/$`name()`	48
Add to `FOR/SOME` clause of E the variable declaration "$var(n_{CT})$ `IN` pe"	49

GenerateResultVariable($\$V, E, S$)

$B \leftarrow \oslash$	50
Find in CT the lowest element node ancestor n_{LA} of $node(\$V)$ such	51
that, in RT, $var(n_{LA})$ in $allvars(S)$	
Construct a relative path expression pe initially consisting of $var(n_{LA})$	52
Walking down the tree path from n_{LA} to $node(\$V)$,	53
for a node n_{CT} of CT on that path:	
Īf n_{CT} is an element node	54
Construct a path expression $pe = var(n_{LA})/name(n_{CT})$	55
Add to `FOR` clause of E the variable declaration "$var(n_{CT})$ `IN` pe"	56
Īf n_{CT} is an AND node with a Boolean expression b	57
Add b to B	58
Īf n_{CT} has a name variable	59
Construct a path expression $pe = var(n_{LA})/$`name()`	60
Add to `FOR` clause of E the variable declaration "$var(n_{CT})$ `IN` pe"	61
Set n_{CT} as n_{LA} and repeat from line 53	62
For every Boolean expression $b_i \in B$	63
F̄or every variable $\$V_i$ used in b_i and not in $allvars(S)$	64
GenerateResultVariable($\$V_i, S, E$)	65
Add to WHERE clause of E the conjunction of the expressions in B	66

Initially, the algorithm produces a `FWOR` expression e for the top-level group-by list of the RT and applies all conditions that appear in CT. This step ensures that the top-level group-by list, and all subsequent ones, is applied on qualified bindings only. All variables in CT that are not under an OR node are declared in the `FOR` clause of e. The Boolean expression labeling the root AND node of CT appears in the `WHERE` clause of e. For each OR node in CT, the algorithm produces a `SOME…SATISFIES` expression e'. The set of e' expressions are connected using Boolean disjunction and placed in the `WHERE` clause of e. Nested OR nodes in CT result in nested `SOME…SATISFIES` expressions. All variables in CT that are under an OR node are declared in the SOME clause of some e'. Boolean expressions labeling AND nodes that are children of an OR node appear in the `SATISFIES` clause of some e'. This step is implemented in lines 2–7 and the subroutines ApplyConditions and GenerateConditionVariable.

As a second step, the algorithm traverses the result tree depth-first and produces a `FWOR` expression, nested in the `RETURN` clause of the enclosing `FWOR` expression when it encounters a group-by list containing a variable labeling a node under an OR node in CT (lines 8–9). The `FOR` clause of the `FWOR` expression declares the variables in the group-by list by traversing the condition tree (lines 10–11 and subroutine GenerateResultVariable). If the nodes of the result tree have an attached Boolean expression, then an `IF…THEN` condition expression is added to the `RETURN` clause of the `FWOR` expression (lines 14–15). Each node of the result tree either constructs an element or generates element content in the `RETURN` clause (lines 16–22). Finally, if a node in the result tree has a

sort-by list, then an ORDER BY clause is added (lines 23–24.) 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, for example, 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 $root IN document(''source.xml''),
        $S IN $root/sensors,
        $MAN IN $S/manufacturer,
        $NAME IN $MAN/name,
        $PROD IN $MAN/product,
        $PART IN $PROD/part_number,
        $SPEC IN $PROD/specs,
        $DIST IN $SPEC/sensing_distance,
        $BODY IN $SPEC/body_type,
        $N_BODY IN $BODY/name(),
        $PROTS IN $SPEC/protection_ratings,
        $PROT IN $PROTS/protection_rating,
        $PROT1 IN $PROTS/protection_rating
    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 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>
```

B. GROUPBY PROPOSAL

The proposal extends the XQuery syntax with the following GroupBy expressions (productions that follow extend those in http://www.w3.org/TR/xquery/#nt-bnf).

```
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
          /********** 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.

To construct an intermediate collection, we only need 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>
```

ACKNOWLEDGMENTS

REFERENCES

ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D.   2000.   *Data on the Web*. Morgan Kaufman, San Francisco, CA.

ABITEBOUL, S., SEGOUFIN, L., AND VIANU, V.   2001.   Representing and querying XML with incomplete information. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

AMER-YAHIA, S., CHO, S., LAKSHMANAN, L. V. S., AND SRIVASTAVA, D.   2001.   Minimization of tree pattern queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

ATZENI, P., MECCA, G., AND MERIALDO, P.   1997.   To weave the web. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. 206–215.

BEA SYSTEMS, INC.   2005.   Liquid Data for WebLogic. http://bea.com.

BERNSTEIN, P. A., BRODIE, M. L., CERI, S., DEWITT, D. J., FRANKLIN, M. J., GARCIA-MOLINA, H., GRAY, J., HELD, G., HELLERSTEIN, J. M., JAGADISH, H. V., LESK, M., MAIER, D., NAUGHTON, J. F., PIRAHESH, H., STONEBRAKER, M., AND ULLMAN, J. D.   1998.   The Asilomar report on database research. *SIGMOD Record 27*, 4, 74–80.

BIRON, P. V. AND MALHOTRA, A.   2004.   XML Schema Part 2: Datatypes Second Edition. W3C Recommendation (Oct.). http://www.w3.org/TR/xmlschema-2/.

BOAG, S., CHAMBERLIN, D., FERNANDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J.   2005.   XQuery 1.0: An XML Query Language. W3C Working Draft 11 (Feb.). http://www.w3.org/TR/xquery/.

CAREY, M. J., HAAS, L. M., MAGANTY, V., AND WILLIAMS, J. H.   1996.   PESTO: An integrated query/browser for object databases. In *Proceedings of the 22nd International Conference on Very Large Data Bases*. 203–214.

CHAWATHE, S. S., BABY, T., AND YEO, J.   2001.   VQBD: Exploring semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

CLUET, S., DELOBEL, C., SIMÉON, J., AND SMAGA, K.   1998.   Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 177–188.

COHEN, S., KANZA, Y., KOGAN, Y. A., NUTT, W., SAGIV, Y., AND SEREBRENIK, A. 1999. EquiX—Easy querying in XML databases. In *Proceedings of the 2nd International Workshop on the Web and Databases*. 43–48.

COMAI, S., DAMIANI, E., AND FRATERNALI, P. 2001. Computing graphical queries over xml data. *ACM Trans. Inf. Syst. 19*, 4, 371–430.

DEUTSCH, A., FERNANDEZ, M. F., FLORESCU, D., LEVY, A. Y., AND SUCIU, D. 1998. XML-QL. W3C Query Languages Workshop.

DEUTSCH, A., PAPAKONSTANTINOU, Y., AND XU, Y. 2004. The NEXT logical framework for XQuery. In *Proceedings of the 30th International Conference on Very Large Data Bases*. 168–179.

DRAPER, D., FANKHAUSER, P., FERNANDEZ, M., MALHOTRA, A., ROSE, K., RYS, M., SIMÉON, J., AND WALDER, P. 2005. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Feb.). http://www.w3.org/TR/xquery-semantics/.

DRAPER, D., HALEVY, A. Y., AND WELD, D. S. 2001. The nimble integration engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

DUBINKO, M., KLOTZ, L. L., MERRICK, R., AND RAMAN, T. V. 2003. XForms 1.0. W3C Recommendation (Oct.). http://www.w3.org/TR/2003/REC-xforms-20031014/.

EYAL, A. AND MILO, T. 2001. Integrating and customizing heterogeneous e-commerce applications. *VLDB J. 10*, 1, 16–38.

FALLSIDE, D. C. AND WALMSLEY, P. 2004. XML Schema Part 0: Primer Second Edition. W3C Recommendation (Oct.). http://www.w3.org/TR/xmlschema-0/.

FERNANDEZ, M., MALHOTRA, A., MARSH, J., NAGY, M., AND WALSH, N. 2005. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft (Feb.). http://www.w3.org/TR/xpath-datamodel/.

FERNANDEZ, M. F., FLORESCU, D., LEVY, A. Y., AND SUCIU, D. 2000. Declarative specification of web sites with strudel. *VLDB J. 9*, 1, 38–55.

FERNANDEZ, M. F., MORISHIMA, A., AND SUCIU, D. 2001. Efficient evaluation of XML middle-ware queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

FRATERNALI, P. 1999. Tools and approaches for developing data-intensive Web applications: A survey. *ACM Comput. Surv. 31*, 3, 227–263.

FRATERNALI, P. AND PAOLINI, P. 2000. Model-driven development of Web applications: The AutoWeb system. *ACM Trans. Inf. Syst. 18*, 4, 323–382.

GENESERETH, M. AND NILSSON, N. J. 1987. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman, San Mateo, CA.

GOLDMAN, R. AND WIDOM, J. 1998. Interactive query and search in semistructured databases. In *Proceedings of the 1st International Workshop on the Web and Databases*. 52–62.

KAY, M. 2005. XSL Transformations (XSLT) Version 2.0. W3C Working Draft (Feb.). http://www.w3.org/TR/xslt20/.

KNUTH, D. E. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

LEVY, A. Y., RAJARAMAN, A., AND ORDILLE, J. J. 1996. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Data Bases*. 251–262.

LEVY, A. Y., RAJARAMAN, A., AND ULLMAN, J. D. 1996. Answering queries using limited external processors. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 227–237.

LUDÄSCHER, B., PAPAKONSTANTINOU, Y., AND VELIKHOV, P. 2000. Navigation-driven evaluation of virtual mediated views. In *Proceedings of the 7th International Conference on Extending Database Technology*. 150–165.

MACROMEDIA, INC. 2003. ColdFusion. http://www.macromedia.com/software/coldfusion/.

MACROMEDIA, INC. 2003. Dreamweaver UltraDev. http://www.macromedia.com/software/ultradev/.

MACROMEDIA, INC. 2003. HomeSite. http://www.macromedia.com/software/homesite/.

MALHOTRA, A., MELTON, J., AND WALSH, N. 2005. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft (Feb.). http://www.w3.org/TR/xpath-functions/.

MICROSOFT CORP. 2005. ASP.NET. http://www.asp.net/.

MICROSOFT CORP. 2004. BizTalk Server. http://www.microsoft.com/biztalk/.

MICROSOFT CORP. 2003. Visual InterDev. http://msdn.microsoft.com/vinterdev/.

MUNROE, K. D. AND PAPAKONSTANTINOU, Y. 2000. BBQ: A visual interface for integrated browsing and querying of XML. In *Proceedings of the 5th Working Conference on Visual Database Systems*. 277–296.

ORACLE. 2004. XSQL. http://www.oracle.com.

PAPAKONSTANTINOU, Y., BORKAR, V. R., ORGIYAN, M., STATHATOS, K., SUTA, L., VASSALOS, V., AND VELIKHOV, P. 2003. XML queries and algebra in the Enosys integration platform. *Data Knowl. Eng. 44*, 3, 299–322.

PAPAKONSTANTINOU, Y., PETROPOULOS, M., AND VASSALOS, V. 2002. QURSED: Querying and reporting semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 192–203.

PETROPOULOS, M., VASSALOS, V., AND PAPAKONSTANTINOU, Y. 2001. XML query forms (XQ-Forms): Declarative specification of XML query interfaces. In *Proceedings of the 10th International World Wide Web Conference*. 642–651.

POPA, L., VELEGRAKIS, Y., MILLER, R. J., HERNÁNDEZ, M. A., AND FAGIN, R. 2002. Translating Web data. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 598–609.

QUASS, D., RAJARAMAN, A., SAGIV, Y., ULLMAN, J. D., AND WIDOM, J. 1995. Querying semistructured heterogeneous information. In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*. 319–344.

RAGGETT, D., HORS, A. L., AND JACOBS, I. 1999. HTML 4.01 Specification. W3C Recommendation (Dec.). http://www.w3.org/TR/html4/.

SCHÖNING, H. AND WÄSCH, J. 2000. Tamino—An Internet database system. In *Proceedings of the 7th International Conference on Extending Database Technology*. 383–387.

SHANMUGASUNDARAM, J., SHEKITA, E. J., BARR, R., CAREY, M. J., LINDSAY, B. G., PIRAHESH, H., AND REINWALD, B. 2000. Efficiently publishing relational data as XML documents. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 65–76.

SILBERSCHATZ, A., STONEBRAKER, M., AND ULLMAN, J. D. 1990. Database systems: Achievements and opportunities. *SIGMOD Record 19*, 4, 6–22.

SUN MICROSYSTEMS, INC. 2005. JavaServer Pages. http://java.sun.com/products/jsp/.

TIBCO SOFTWARE INC. 2005. XML Transform. http://tibco.com.

VASSALOS, V. AND PAPAKONSTANTINOU, Y. 2000. Expressive capabilities description languages and query rewriting algorithms. *J. Logic Program. 43*, 1, 75–122.

ZLOOF, M. M. 1975. Query by example. In *Proceedings of the AFIPS Conference*. 431–438.