A Document-driven Approach to Database Report Generation

Daniel K.C. Chan *

Project VERSO, INRIA - Rocquencourt
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex, France
Email: Daniel.Chan@inria.fr, Phone: +33 (0)1 39 63 52 66, Fax: +33 (0)1 39 63 56 74

Abstract

It can be argued that report generation is the most frequently performed task in database applications. Therefore the efficacy and efficiency of a database report generation mechanism has a significant impact on productivity. This paper introduces a document-driven approach as opposed to the traditional schema-driven approach to report generation resulting in more flexibility. In this approach, generating a report involves specifying it in terms of a user defined SGML-based report model. Contents of a report can be specified using a transformation language together with queries that retrieve data from different databases. A report is output as a SGML document which can be further edited as well as translated to other formats, for instance to HTML to be viewed using an Internet browser. This paper presents the approach using an example and discusses the features and usage of the transformation language which is a small but expressive language. Despite of the fact that we have only investigated the transformation against relational databases, we believe that this approach can unify report generation for different database models.

1 Introduction

It can be argued that report generation is the most frequently performed task in database applications. Therefore the efficacy and efficiency of a database report generation mechanism has a significant impact on the productivity of database application development and operation.

To facilitate report generation, some report generator is usually provided in most relational database systems. Such a report generator usually allows the user to specify both the layout and the contents of a report. In some cases, the report layout can be defined via graphical interfaces. Whereas the majority of the report contents are usually specified using database queries. In the case of advanced database systems such as those using an object-oriented data model, support for report generation is rather rudimentary.

The existing approach to report generation can be described as *schema-driven* as the logical structure of the contents of a report is determined primarily by the database schema and by the restructuring power of the query language used. However, it has been observed that database applications often require reports that are more complicated - in terms of structures and computation - than what many query languages can conveniently express. Also, the lack of a generic report generation mechanism has the consequence that reports specified using one database system are very unlikely to work on another database system even if the underlying database schemata are very similar or the same. In a nutshell, genericity and expressiveness are the pressing problems.

This paper introduces a *document-driven* approach as opposed to the conventional schema-driven approach to report generation. With this approach, generating a report involves specifying it in terms of a *report model* which is independent of any database systems or models. The report contents can be specified using a *transformation language* and queries to possibly different databases.

With the document-driven approach, restructuring that cannot be performed conveniently by query languages can be performed at the report model level using the transformation language. This paper presents a system architecture that can be easily adapted to different systems and therefore can potentially benefit many developers of data intensive applications.

The rest of the paper is structured as follows. Section 2 presents an overview of the architecture for realising the document-driven approach to report generation. Section 3 gives an example of the usage of the system. Section 4 describes the transformation language which is the core of the system. Section 5 concludes.

*Supported by a Marie Curie Fellowship from the European Commission (ERBFMBICT960653).

2 System Overview

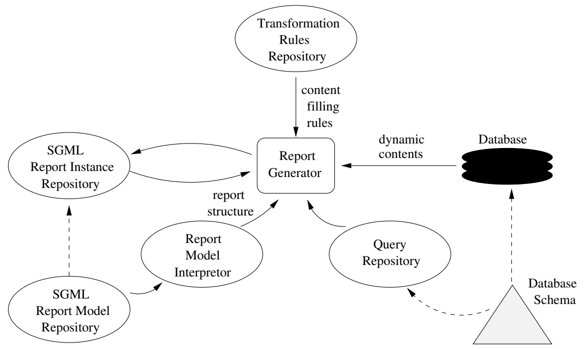

Figure 1. A Conceptual Architecture

The conceptual components of the proposed architecture are depicted in Figure 1. The right hand side depicts the database (together with its schema) from which data are drawn to fill the report. Since the contents of the database gets changed from time to time the corresponding contents of the generated report will therefore change accordingly. This is why the arrow between the database and the report generator is labelled *dynamic contents*. The left hand side depicts the possibility of having a number of *report models* and each model having a number of either partially and completely filled *report instances*. The report model interpretor helps the report generator decide on the possible structures of the report in question. A report can be partially filled, stored in the report instance repository, and then retrieved from there to be filled further. The connection between the data from the database and the report structure as defined in the report model is specified using transformation rules. Such rules are stored in a repository depicted at the top of Figure 1.

In this paper, it is assumed that different databases are queried using their host query languages rather than a universal query language working for all systems and models. The later has been showed to be possible [1] and hence is not discussed in this paper. The query repository therefore contains named queries imposed on specific database schema and databases, for instance a query named `researchers` containing an OQL query statement for an O_2 database called `Rocquencourt` defined using a schema called `INRIA`.

Despite the primary objective to generate hard copy reports, it is desirable that reports can be presented on different media. For example, it would be useful to be able to present a report via an Internet browser. A two-step output phrase is therefore introduced (not depicted in Figure 1). The system first generates the report in a "neutral format" which captures essentially the logical properties of the report. In the second step, the system translates this neutral format into a particular output format which captures both logical and physical properties of the report. This way, some degree of genericity is achieved with the output format. Again SGML is chosen as the neutral format for output and translation to other formats can be easily done using existing converters, e.g. from SGML to HTML.

3 Example Application

The document-driven approach to database report generation is demonstrated in this section using a simple example about the generation of a sales report from a relational database. More complicated applications will be presented in the next section where the transformation language is discussed in detail.

3.1 Data Retrieval

To begin with one has to identify the database from which data are retrieved to fill the report. The relational database used in this example is assumed to have the following schema:

```
Order(CustomerNo, OrderNo, Date, DeliveryAddress, Total)
```

The database is assumed to contain the following tuples:

```
(101, 620, 970312, ...London, 4450)
(102, 614, 980205, ...Aberdeen, 23940)
(102, 618, 970306, ...Cardiff, 3510.50)
(104, 612, 980120, ...Glasgow, 5860)
(104, 619, 980222, ...Edinburgh, 1260)
(105, 617, 970303, ...Manchester, 46370)
(106, 605, 970730, ...Birmingham, 8324)
(108, 613, 980201, ...Bristol, 6400)
```

Data are retrieved using SQL queries which are stored in files in the query repository. For the sales report, a query to retrieve all transactions with an amount higher than £1000 is defined and given the name `BigSpender`. The file in which the query is stored is named `HighEndSales`.

```
BigSpender   :=   SELECT   CustomerNo, OrderNo, Date, Total
                  FROM     Order
                  WHERE    Total > 1000
```

3.2 Report Modelling

The modelling of a report starts with the definition of a report model which captures the content structure of the report. A good report model will facilitate further processing of reports as well as the generation of reports using the approach described in this paper. A report model is defined as a SGML DTD like the following:

```
<!--                   DTD for Sales Report          -->
<!ELEMENT   Report     - -    (Title, Table)          >
<!ELEMENT   Title      - -    (#PCDATA)               >
<!ELEMENT   Table      - O    (Headings, Body)        >
<!ELEMENT   Headings   - O    (Heading)*              >
<!ELEMENT   Heading    - O    (#PCDATA)               >
<!ELEMENT   Body       - O    (Row)*                  >
<!ELEMENT   Row        - O    (Column)*               >
<!ELEMENT   Column     - O    (#PCDATA)               >
```

A report can then be constructed according to the content structure described in the report model. It should be noted that more than one report can be defined using the same report model, for instance one sales report for each branch of a company. Specifying a report with this approach often is a matter of putting in the title of the report and headings for the various rows and columns with the data contents left to be specified using transformation rules. The sales report, which is assumed to be stored in a file named `ExampleReport`, needs only to be partially constructed as follows:

```
<! DOCTYPE Report PUBLIC "Sales Report">
<Report>
<Title>Sales Report</Title>
<Table>
<Headings>
<Heading>Customer Number
<Heading>Order Number
<Heading>Delivery Date
<Heading>Sales Amount
</Report>
```

The contents of the report under the structural part named `Body` is left unspecified and will be filled in using transformation rules which are introduced next.

3.3 Mapping Data to Reports

To map the data retrieved from a database to a report, transformation rules can be used as follows:

```
report ExampleReport;
query HighEndSales;
<Body> <<- BigSpender;

1 st <Column> of <Row> <- CustomerNo;
2 nd <Column> of <Row> <- OrderNo;
3 rd <Column> of <Row> <- Date;
4 th <Column> of <Row> <- Total;
```

The first two lines specify the files in which the report and query are defined. The transformation begins with mapping each tuple in `BigSpender` to a `<Row>` in `<Body>`. The `<<-` operator is called *zip* which takes two collections and pairs up elements of the collections. In this case, one collection is a relation and another is a list of `<Row>`'s. Based on the DTD defined earlier the system can discover that a `<Row>` can have a number of `<Column>`'s. The remaining transformation rules help the system determine how the attributes of a tuple are to be *mapped* to `<Column>`'s in a `<Row>`. Keywords such as `st` in `1 st <Column> of <Row>` are index operator used to identify elements in a list. The `of` operator is a context operator giving additional information about the enclosing structure of the part specified on its left. In this case, it indicates that `1 st <Column>` is a sub-component of `<Row>`.

After the transformation rules are executed, the partially filled report will become completely filled (as far as this example is concerned). The output will appear as follows:

```
<! DOCTYPE Report PUBLIC "Sales Report">
<Report>
<Title>Sales Report</Title>
<Table>
<Headings>
<Heading>Customer Number
<Heading>Order Number
<Heading>Delivery Date
<Heading>Sales Amount
<Body>
<Row> <Column>101 <Column>620 <Column>970312 <Column>4450
<Row> <Column>102 <Column>614 <Column>980205 <Column>23940
<Row> <Column>102 <Column>618 <Column>970306 <Column>3510.50
<Row> <Column>104 <Column>612 <Column>980120 <Column>5860
<Row> <Column>104 <Column>619 <Column>980222 <Column>1260
<Row> <Column>105 <Column>617 <Column>970303 <Column>46370
<Row> <Column>106 <Column>605 <Column>970730 <Column>8324
<Row> <Column>108 <Column>613 <Column>980201 <Column>6400
</Report>
```

4 Transformation Language

4.1 Data & Report Models

The transformation language assumes a model for the relations from which data are retrieved to form the contents of the report. A relation is perceived as a list of tuples each of which having one or more attributes (Figure 2). The interpretation of a relation being a list instead of a set is more for theoretical convenience and such an interpretation is not harmful as long as the ordering of tuples is not assumed in the generation of the report. So far, no such ordering information is exploited in all the examples that have been tried with this approach. Given that tuples are ordered, it is possible to identify a tuple using the position within the relation list. Index operators are used for this purpose and their interpretation is given in Figure 2 as labelled edges between `BigSpender` and `Tuple`'s (a predefined identifier for tuples). Consequently, a relation is a 3-level tree in which all nodes are labelled whereas only some edges are labelled with index operators.

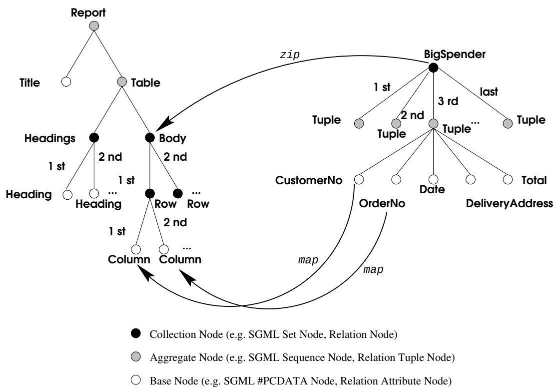

Figure 2. Models for Reports and Relations

Report models are those that can be defined using SGML DTD's. It is therefore possible to construct report models with cycles between components; nevertheless, no such cycles are required with all the examples that have been tested using this approach. It is therefore reasonable to interpret a report simply as a tree. Like the model for relations, index operators can be used to label elements in a collection, e.g. the multiple occurrences of `<Row>` within `<Body>`. Unlike the model for relations, nodes in a report model may not be labelled. Generally speaking, report generation would be more convenient if all the nodes are labelled. In other words, one should avoid using definitions with *anonymous nodes*. For instance, one should replace `(MANY_SECOND)*` in `<!ELEMENT WRONG -- (FIRST,(MANY_SECOND)*) >` with the name of an element which is defined as `(MANY_SECOND)*`. It should be noted that a node can be defined as a *union* of a number of kinds of nodes; however, it does not require special treatment from the transformation language.

4.2 Transformation Rules

As shown in the last section, a report is generated by combining a report specification with data from the database according to some transformation rules. In its most basic form, a rule establishes that data from a node in the model for relations are to be used to build contents of a node in the report model. Two such connections can be specified: one between collections using `zip` and another between non-collections using `map` as shown in Figure 2.

Some connections are not explicitly specified since they can be easily inferred from the given transformation rules. One such example is the connection between `Tuple` and `<Row>`. Typically, explicit connections are required when *anonymous nodes* and *union* are used together.

In its next simplest form, a rule connects a node from the model of relations to a node at the end of a *path* in the report model. A path serves to provide a context to a node which is necessary to distinguish nodes with the same name. A path is constructed using node names separated by the context operator `of` and a node in the path can be prefixed with an index operator such as `1 st` if the edge between the nodes is labelled as such.

4.3 Computation

The transformation language can perform simple computation which is not demonstrated in the example given in the previous section. Assuming that the sales report is extended to include the delivery address as well as the taxed total for each order like the following:

```
                         Sales Report

-------------------------------------------------------------
Customer   Order    Delivery    Delivery      Sales      Taxed
Number     Number   Date        Address       Amount     Total
-------------------------------------------------------------
101        620      97-Mar-12   ...London   4450.00    4895.00
                                ...
```

The transformation rules will have to be changed accordingly to reflect the new requirements. The taxed total including 10% of sales tax is obtained using a simple expression on the right hand side of *map* as shown below. (It should be noted that this calculation can actually be performed by the query language during data retrieval.)

```
4 th <Column> of <Row> <- DeliveryAddress;
5 th <Column> of <Row> <- Total;
6 th <Column> of <Row> <- Total * 1.1;
```

Transformation rules can also have conditions associated with them. For example, one may want to output a string of asterisks when a *null* value is found in the `DeliveryAddress` attribute.

```
4 th <Column> of <Row> <- DeliveryAddress if DeliveryAdress is not null;
4 th <Column> of <Row> <- "*****"         if DeliveryAdress is null;
```

Another tricky scenario is when a query returns no tuple at all. This state of affairs can be detected using the keyword `empty`. One possible action when `BigSpender` returns no tuple is given below. The specification of a path on the left hand side of *map* causes a `<Row>` to be constructed with one `<Column>` containing the string "END".

```
1 st <Column> of <Row> <- "END"           if BigSpender is empty;
```

It is often the case that a query is supposed to return only one tuple. A convenient way to extract the tuple is by using index operator, e.g. `1 st Tuple of BigSpender`. More complex computation can be expressed using statement block – a number of statements enclosed between {}.

4.4 Restructuring

This subsection focuses on how data restructuring can be performed using the transformation language and rule *attributes*. The use of rule attributes is motivated by the use of attributes in attribute grammar. Basically, the nodes of a report model can be decorated with attributes whose values are determined by data from the database or values of other attributes. The use of rule attributes is demonstrated using an example whose output is given below. This sales report requires orders of the same customer to be grouped together where the customer number appears only once, a running total is kept for each order, and there is a grand total for each customer. The complete solution will be given in stages: (1) a solution that supports grouping, (2) a solution that supports grand totals, and (3) a solution that supports running totals.

```
                     Sales Report

------------------------------------------------------------
Customer      Delivery       Sales     Taxed       Running
Number        Date          Amount     Total         Total
------------------------------------------------------------
101           97-Mar-12    4450.00   4895.00       4895.00
                                     -------
                                     4895.00

102           98-Feb-05   23940.00  26334.00      26334.00
              97-Mar-06    3510.50   3861.55      30195.55
                                    --------
                                    30195.55
                     ...
```

The first solution demonstrates the grouping of orders by customer but produces no grand totals and running totals. It assumes a slightly different report model with the following changes:

```
                              ...
<!ELEMENT    Body          - O    (Customer)*          >
<!ELEMENT    Customer      - O    (Number, Orders)     >
<!ELEMENT    Orders        - O    (Row)*               >
                              ...
```

The corresponding transformation rules are shown below. The first two rules are actually a new kind of rules. Unlike the previously shown transformation rules which are for mapping data to the report, these *decoration rules* are used to establish relationships between nodes in the report model. It is achieved by introducing attributes to nodes and establishing relationships between attributes and hence the corresponding nodes. The normal transformation rules can therefore be called *horizontal rules* while the decoration rules can be called *vertical rules*. In this solution, the two rules establish that the <Number>, <Orders>, and all <Row> nodes have an attribute called number. In addition, the values of these attributes under the same <Customer> should be the same. The number attribute of a <Row> is updated using an attribute from the database (the 5th rule) and hence can be used as a source to instantiate the number attributes of <Number> and <Orders> since they are equal by definition.

```
<Customer> -> <Number> <Orders> if number of <Number> = number of <Orders>;
<Orders> -> (<Row>)*            if number of <Orders> = number of <Row>;
<Orders> <<- BigSpender;
<Number> <- number of <Number>;
number of <Row> <- CustomerNo;
1 st <Column> of <Row> <- Date;
2 nd <Column> of <Row> <- Total;
3 rd <Column> of <Row> <- Total * 1.1;
```

The attributes can be seen as sub-nodes depicted as square boxes in Figure 3. Equality of attributes is indicated by dotted lines with arrows at both ends. Actually, more equality relationships than those shown can be inferred from the given rules. The fact that number of <Row> carries a value from the database but not the other two number attributes dictates the evaluation of the sub-tree under <Customer> to begin from <Row>. It should be noted that this solution causes a bottom-up evaluation since the construction of <Customer> is not explicitly specified in the mapping from the database. In other words, <Row>'s are constructed using tuples from BigSpender before they are grouped to construct <Orders> and afterwards <Customer>.

The second solution provides a grand total for each customer and assumes a report model with the following difference:

```
                       ...
<!ELEMENT    Customer  - O    (Number, Orders, Sum)    >
                       ...
```

The key feature of this solution is that computation is inserted to dictate the order of construction of the report components. The first decoration rule specifies that the sum attribute of <Sum> obtains its value from the sum attribute of <Orders> before <Sum> is constructed – the positioning of the statement block decides whether the block is executed before or after the construction of a node. In the second decoration rule the sum attribute of <Orders> is set to be the summation of the orderTotal attributes of the <Row> elements in <Orders>. The computation is performed using a high-order collection operator called

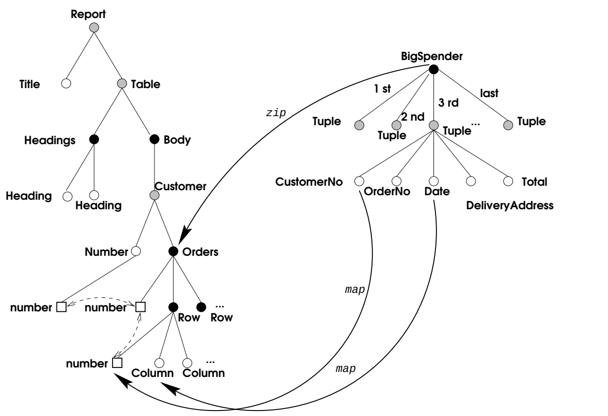

Figure 3. Grouping via Equality over Attributes

reduce which returns the first parameter if the operand collection is empty otherwise it will apply the second parameter (a unary operator) to each element and combine the results using the third parameter (a binary operator) if there is more than one element in the operand collection. Since collection is a native construct in SGML, it is therefore necessary to have appropriate operators for collections. The provision of reduce allows many common collection operators, e.g. summation and average, to be expressed.

```
<Customer> -> <Number> <Orders>
              { sum of <Sum> := sum of <Orders>; }
              <Sum>
              if number of <Number> = number of <Orders>;
<Orders> -> (<Row>)*
            { sum of <Orders> := reduce(0,orderTotal,+,<Orders>); }
            if number of <Orders> = number of <Row>;

<Orders> <<- BigSpender;
<Number> <- number of <Number>;
1 st <Column> of <Row> <- Date;
2 nd <Column> of <Row> <- Total;
3 rd <Column> of <Row> <- Total * 1.1;
number of <Row> <- CustomerNo;
orderTotal of <Row> <- Total * 1.1;
<Sum> <- sum of <Sum>;
```

Actually this report can be generated using simpler decoration rules with the help of an addition query that returns the grand total of each customer (SELECT CustomerNo, SUM(Total) FROM BigSpender GROUP BY CustomerNo). This new relation can be used to construct <Number> and <Sum> while the original relation keeps its role as the source for <Orders>. Consequently, different evaluation strategies can de adopted due to existence of two instantiation sources.

The third and complete solution for the extended sales report extends the second solution with one more attribute called running. It is associated with <Orders> and is reset to zero before the construction of <Orders> (1st rule). Its value is incremented by the orderTotal attribute of <Row> after the construction of each <Row> (2nd rule). The running attribute is then used to fill in the 4th column of the report (8th rule).

```
<Customer> -> <Number>
              { running of <Orders> := 0; }
              <Orders>
              { sum of <Sum> := sum of <Orders>; }
              <Sum>
              if number of <Number> = number of <Orders>;
<Orders> -> (<Row>
             { running of <Orders> :=
                running of <Orders> + orderTotal of <Row>; }
            )*
            { sum of <Orders> := reduce(0,orderTotal,+,<Orders>); }
            if number of <Orders> = number of <Row>;

<Orders> <- BigSpender;
<Number> <- number of <Number>;
1 st <Column> of <Row> <- Date;
2 nd <Column> of <Row> <- Total;
3 rd <Column> of <Row> <- Total * 1.1;
4 th <Column> of <Row> <- running of <Orders> + Total * 1.1;
number of <Row> <- CustomerNo;
orderTotal of <Row> <- Total * 1.1;
<Sum> <- sum of <Sum>;
```

4.5 Expressive Power

The expressive power of the transformation language has been given only an empirical assessment using examples obtained from various sources. The transformation language can express all the 42 examples of Oracle Reports given in [4] including time series calculations (e.g. 4-month moving average). Perhaps, it is only fair to point out that features related to paging (e.g. having data values in page headings and complex page numbering) and printing (e.g. cheque printing and invoice printing that require precise positioning) are not considered within the scope of this approach. In other words, physical properties of a report are supposed to be governed by either style sheets or format translators. As a whole the language has been found to be sufficiently powerful for many reports; moreover, change of requirements usually can be easily dealt with.

4.6 Evaluation

The generation of reports can be achieved using different evaluation strategies depending on the dependency given in the transformation rules. It is possible that there is no dependency between different branches of a node – when there is no vertical rule – which suggests different branches can be constructed independently with the potential of being performed in parallel. Having said that there is always some dependency between nodes and the database. Therefore branches instantiated using the same tuple are to be constructed together. Dependency introduced by decoration rules dictates the order of evaluation between branches of a node. With equality dependency, once one of the equated nodes is instantiated the remaining nodes are automatically instantiated with the same value. With assignment dependency, the order of of evaluation has to be strictly followed. Naturally, all evaluation steps need to respect the conditions given in the rules.

5 Conclusions

This paper presented a document-driven approach to database report generation which allows more flexibility than the traditional schema-driven approach [3, 2]. With this approach data are retrieved from databases using their host query facilities while reports are specified independently as SGML documents. The bridge between the two is defined using a transformation language which is also database system independent. The express power of the transformation language is sufficient for all the reports that we have come across including all kinds of reports that can be generated using ORACLE Reports. Nevertheless, the transformation language is very concise and adaptable to changes or requirements.

In a nutshell, the contribution of this work is the provision of an architecture as well as a database model and database system independent language that can bridge two well established technologies (SGML and databases) for the purpose of generating reports containing data from the databases. The benefits of the approach is genericity, simplicity, expressiveness, and independence. It is a more generic approach because it has broken free the long traditional of having report generation facilities tied closely to database systems. Also, the approach can be uniformly applied to different database systems. The approach is simple because the transformation language is very concise and does not rely on any intermediate model for data restructuring and computation. This is not usually the case with other conversion languages. The approach is expressive since the transformation allows us to express complicated reports. This approach promotes the independence between logical and physical properties of reports due to the use of SGML. As a result, report contents are properly captured leading no information loss. Moreover, the use of SGML as output format turns database reports into first-class documents that can be further processed, for example, for analysis purposes.

Acknowledgements

I would like to thank Jérôme Siméon, Sophie Cluet, and Anne-Marie Vercoustre of INRIA, France as well as Vassilis Christophides of FORTH, Greece for their comments at various stages of the work reported in this paper.

References

[1] D. Chan and P. Trinder. A Processing Framework for Object Comprehensions. *Information and Software Technology*, 39(9):641–651, 1997.

[2] O_2 Technology, 7 rue du Parc de Clagny, 78035 Versailles Cedex, France. *O_2Report User Manual*, 4.5 edition, Mar. 1995.

[3] O_2 Technology, 7 rue du Parc de Clagny, 78035 Versailles Cedex, France. *O_2Web User Manual*, 4.6 edition, Jan. 1996.

[4] Oracle Corporation, U.S.A. *Building Reports Manual*, 2.5 edition, Apr. 1995.