Publish by Example

Sonia Guéhis
University of Paris Dauphine
sonia.guehis@lamsade.dauphine.fr

David Gross-Amblard
University of Bourgogne
david.gross-amblard@u-bourgogne.fr

Philippe Rigaux
University of Paris Dauphine and INRIA-Orsay
philippe.rigaux@lamasde.dauphine.fr

Abstract

We propose an approach for producing database publishing programs by example. *The main idea is to interactively build an example document, representative of the program output. The system infers from this document, without ambiguity, the publishing program. The end-user does not need to know a programming language, a query language or the database schema.*

1 Introduction

We consider the problem of producing "dynamic" documents that contain data retrieved from a relational database. We impose no restriction on our concept of document: it can be non-structured character data (e.g., an email), an XML document (for data exchange purposes), an HTML document (web site publishing), a LaTeX file or an Excel spreadsheet, etc. Their common characteristic is to consist both of *static* parts and *dynamic* parts, the latter being values extracted from the database when the document is produced. We call *database publishing* the process of creating dynamic documents from a relational instance. The most typical example is the production of (X)HTML pages in dynamic web sites. We use it for illustration purposes in this paper.

Relational database publishing is technically simple, but requires in practice the association of programming tools and database concepts which often make the production tedious and error-prone. It constitutes in particular an intricate practical aspect of web site engineering [10]. Specialized languages, such as Servlets/JSP, PHP or ColdFusion [3], bring partially satisfying solutions. However, in all cases, writing a database publishing program requires heterogeneous technical knowledge, including: (i) the basics of a programming language (say, Java/JSP); (ii) a query language (say, SQL); (iii) the database schema.

In the present paper we propose a simple mechanism to produce database publishing programs. The main idea is to interactively construct a sample dynamic document which can then be used to infer without ambiguity the publishing program. What makes such an approach effective is the inherent simplicity of relational publishing which does not require the full power of general-purpose programming and query languages.

The benefits are twofold. First the proposed mechanism does not require any technical expertise. As such if offers to non-expert users an opportunity to create rich documents with minimal efforts. Second it constitutes a generic approach which holds independently from a specific environment, does not require any preliminary decision regarding programming practices and conventions, and avoids the tedious and repetitive programming tasks. One obtains a high-level specification of publishing programs, with potential support for software engineering tasks (e.g., verification) as well as database optimization.

Overview of the approach.

Figure 1 presents the main components of our approach, and their respective roles in a publishing system. First, we formalize relational database publishing as a "document query language", called DocQL, already proposed in preliminary form in [6]. A DocQL query can be seen as a syntax-neutral (declarative) specification of a publishing program written in Java/JSP or in any equivalent programming framework. Producing a DocQL query constitutes the target of the publish-by-example process.

The publishing model relies on the concepts of *canonical documents* and *canonical instances*. A canonical document characterizes uniquely a DocQL query q, and therefore the publishing program which can be derived from q. The user interacts with a WYSIWYG graphical editor which lets him construct a canonical document D from a *canonical instance* I_C.

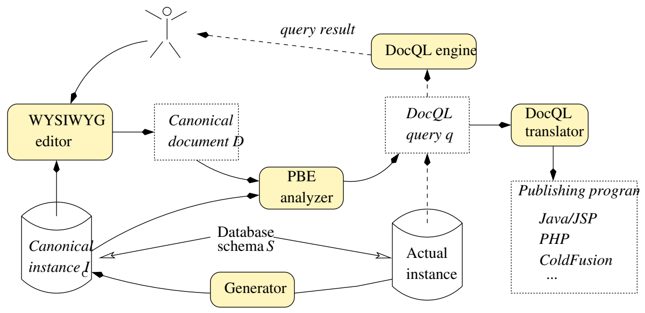

Figure 1. Overview of the publish by example process

A canonical instance I_C of a schema S is an instance such that, for any DocQL query q, there exists a canonical document over I_C that characterizes q. Proposing a canonical instance to a user is tantamount to the ability of producing, *by example*, *all* the possible DocQL queries that can be expressed over S. The canonical instance is a predetermined instance of the schema S, generated by the system administrator using an *instance generator*.

Given a canonical instance I_C and a canonical document D built over I_C, the *Publish By Example* analyser infers a unique DocQL query q. The user can then either run q over the actual instance, through the DocQL engine, or translate q to a traditional publishing program, written in any convenient language.

Running example.

Throughout the paper we illustrate our approach over a sample movie database representing movies with their (unique) director and their (many) actors. Figure 2 shows a simple database instance. Primary keys are in bold, and foreign keys in italic.

In the rest of this paper, Section 2 briefly introduces the DocQL language. We then describe the publication model in Section 3.Section 4 positions our proposal with respect to the state of the art, and Section 5 concludes the paper. A long version of the paper, along with an on-line demonstrator, is available at *http://www.lamsade.dauphine.fr/rigaux/docql*.

2 The publishing language DocQL

We give the main features of the publishing query language DocQL. Since this does not constitute a contribution of the present paper, we limit the presentation to a few illustrative examples. Formal definitions can be found in [6].

Data model.

DocQL aims at a concise specification of publishing

title	year	*id_director*	genre
Unforgiven	1992	20	Western
Van Gogh	1990	29	Drama
Kagemusha	1980	68	Drama
Absolute Power	1997	20	Crime

Movie

id	last_name	first_name
20	Eastwood	Clint
21	Hackman	Gene
29	Pialat	Maurice
30	Dutronc	Jacques
68	Kurosawa	Akira

Artist

title	***id_actor***	character
Unforgiven	20	William Munny
Unforgiven	21	Little Bill Dagget
Van Gogh	30	Van Gogh
Absolute Power	21	President Allen Richmond

Cast

Figure 2. An instance of the *Movies* database

programs. The language captures with a uniform and simple syntax the queries and programming instructions used to build dynamic documents. It relies on a navigation mechanism in an instance I modeled as a labeled directed graph $\mathcal{G}_I$. Tuples are seen as internal nodes, values as leaf nodes, and edges represent either tuple-to-tuple relationships or tuple-to-attribute dependencies.

Figure 3 shows the data graph of the instance of Figure 2. We distinguish functional dependencies between nodes (e.g., between a movie node and its director node) and multivalued dependencies (e.g., between a movie node and its actor nodes). The former are shown with white-headed arrows, the latter with black ones.

If N_1 and N_2 are two nodes in the data graph, we note $N_1 \xrightarrow{p} N_2$ if N_2 functionally depends on N_1, and we say that p is a *unique path*. For instance if N_1 is a movie and N_2 the last name of its director, then $N_1 \overset{director.last\_name}{\rightarrow} N_2$, and `director.last_name` is a unique path. Else we note $N_1 \overset{p}{\twoheadrightarrow} N_2$ and say that p is an instance of a *multiple path*.

The *context* of a node N is the set of leaf nodes that functionally depend on N. The *neighborhood* of N is the set of nodes N' such that there exists an elementary multiple path (i.e., with only one edge) $N \overset{p}{\twoheadrightarrow} N'$. Consider again Figure 3 and the node (of type *Movie*) in the box. Its *context* consists of the values *Unforgiven* (title of the movie), *1992* (year), *Western* (genre), *20*, *Clint*, and *Eastwood* (resp. the id, first name and last name of the director who is uniquely determined by the movie). The neighborhood consists of the two nodes *Cast*.

Query language.

DocQL combines *navigation* in the data graph with *in-*

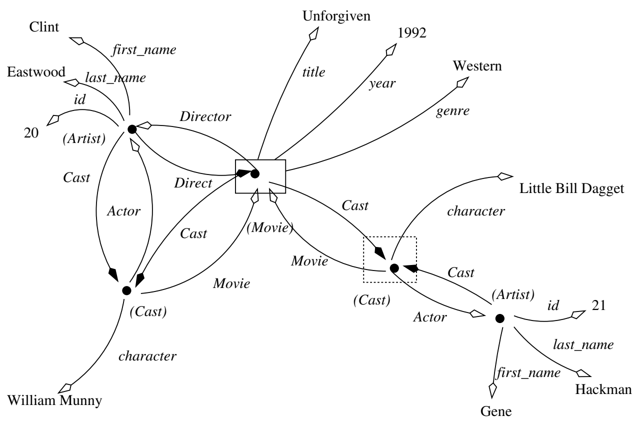

Figure 3. A subset of the data graph of our sample instance

stantiation of the textual fragments that contribute to the final document. A DocQL query is essentially a tree of path expressions which denote the part of the graph that must be visited in order to retrieve the data to include in the final document. Path expressions use an XPath-like syntax. An expression p is interpreted with respect to an *initial node* N_i (unless it begins with `db` which plays the role of / in XPath), and delivers a set of nodes, called the *terminal nodes* of p (with respect to N_i). Each path is associated to a fragment which is instantiated for each terminal node. Path and fragments are syntactically organized in *rules* of the form `@path[condition]{fragment}`, where `path` is a path expression, `condition` is a node condition and `fragment` is the fragment instantiated for each instance of `path`.

The following example shows a DocQL query over our *Movies* database. It produces a (rough) document showing the movie *Unforgiven* along with its director and actors.

Example 1 `@db.Movie[title='Unforgiven']{`

```
 @title{}, @year{}, directed by
  @director.first_name{} @director.last_name{}
 Featuring:
  @Cast{
    - @artist.first_name{}
        @artist.last_name{}as @character{}
  }
}
```

The semantics of the language corresponds to nested loops that explore the data graph, one loop per rule. This navigation produces the *trace* of a query q, which is a finite unfolding of the graph $\mathcal{G}_I$ representing the nodes visited during the evaluation of q. The result of a query is obtained by "decorating" the nodes of its trace with the (static) character data of their associated rules. Applied to the data graph of Figure 3, one obtains the following document as result of the previous example:

```
Unforgiven, 1992, directed by Clint E-
astwood, Featuring:
  - Clint Eastwood as William Munny
  - Gene Hackman as Little Bill Dagget
```

3 The Publish by Example Model

We now develop our model by defining our two key concepts: *canonical documents* and *canonical instances*.

Structure of canonical documents.

A canonical document has a hierarchical structure. Each node of the document's structure is called a *block*. A block is a character string with (optional) references to other blocks. The textual part of a block consists of fixed text and values from the active domain (i.e., leaves) of the graph $\mathcal{G}_I$.

Let Σ be an alphabet. $\mathcal{F} \subset \Sigma^*$ denotes the set of static fragments, and **dom** $\subset \Sigma^*$ denotes the active domain of $\mathcal{G}_I$. For the sake of simplicity, we suppose that $\mathcal{F} \cap \textbf{dom} = \emptyset$, in order to distinguish elements from theses two sets. In practice, the distinction may rely on a syntactical convention We also assume a set $\mathcal{B}$, distinct from the previous ones, of *block identifiers*.

Definition 1 (Block) *A* block B *is a pair* (i, b)*, where* $i \in \mathcal{B}$ *is the* block identifier *and* $b \in (\mathcal{F}|\textbf{dom}|\mathcal{B})^*$ *is the* block body. *We denote by* components(B) *the set of blocks recursively referenced by the body of* B.

We are interested in blocks that can be unambiguously interpreted with respect to $\mathcal{G}_I$. We first define the notion of *representative node* of a block.

Definition 2 (Representative node of a block) *A node* $N \in \mathcal{G}_I$ *is* representative *of a block* (i, b) *if and only if each value* $v \in \textbf{dom}$ *in* b *belongs to the context of* N.

Recall that the *context* of a node N is the set of values v that functionally depend on N. Consider for example the block B with body "**Unforgiven**, published in **1992** and directed by **Clint Eastwood**", where values from **dom** appear in bold. The node N corresponding to the movie *Unforgiven* is representative of B, because each value v belongs to the context of N (see Figure 3).

Let B be a block and N be a representative node of B. We say that B is *valid* with respect to N if there exists a representative node for each component of B, such that the structure of the subgraph induced by these nodes is homomorphic to the structure of B. Formally:

Definition 3 (Block validity) *A block* B *is* valid *with respect to a node* N *if and only if* N *is a representative node,* *and for each child block B_i of B there exists a node N_i in the neighborhood of N such that B_i is valid with respect to N_i.*

A block B is said to be valid on $\mathcal{G}_I$ *if there exists a node N in $\mathcal{G}_I$, such that B is valid with respect to N.*

Consider block B_1 with body "**Unforgiven**, **1992**, featuring: #ref(2)", referencing block B_2 with body "**Little Bill Dagget** played by **Gene Hackman**". B_1 is valid with respect to the node N_1 (framed with solid lines in Figure 3) because we can find a node N_2 (framed with dotted lines), representative of B_2 in the neighborhood of N_1, with $N_1 \overset{Cast}{\twoheadrightarrow} N_2$. Note that **Little Bill Dagget**, **Gene** and **Hackman**, all belong to the context of N_2.

Interpretation of valid blocks.

Given a block B valid on $\mathcal{G}_I$, our goal is to define a mapping that uniquely determines a query q from B and $\mathcal{G}_I$. A complementary question is to know, given a query q, whether there exists a block B valid on $\mathcal{G}_I$ that determines q. We introduce three constraints on $\mathcal{G}_I$: completeness, minimality and non-ambiguity. An instance is said *complete* if, for each node N of type $r \in \mathcal{R}$, and each edge type e of the form $r \xrightarrow{a} r'$, there exists at least one edge $N \xrightarrow{a} N'$. The instance is *minimal* is there is at most one such edge. The *non-ambiguity* condition is defined as follows:

Definition 4 (Non-ambiguous instance) *An instance $\mathcal{G}_I$ is* non-ambiguous *if and only if, for all node N, the following conditions hold:*

- *if N' is a node in the context (resp. in the neighborhood) of N, there exists only one path p such that $N \xrightarrow{p} N'$ (resp. $N \overset{p}{\twoheadrightarrow} N'$);*
- *if N_1 and N_2 are two nodes of the neighborhood, then $context(N_1) \cap context(N_2) = \emptyset$.*

Checking this property for a given instance is easily done by visiting each node and verifying its context and neighborhood.

The first condition requires that if N' is a node in the context or in the neighborhood of N, then the path leading from N to N' can be uniquely determined. The instance on Figure 3 would be ambiguous if, for example, the movie title *and* the director's name were both 'Eastwood' (condition on the context).

The second condition ensures that a node in the neighborhood of N can be uniquely determined by any value of its context. Still looking at Figure 3, assume that we add a (multiple) path *producer* between movies and artists. The instance becomes ambiguous if the producer's name is *William Munny*, since we can no longer determine whether this value is the character of the neighborhood's node *Cast* or the name of the neighborhood's node *Producer*.

The instance of Figure 3 is non-ambiguous, but not minimal nor complete. If we remove the node squared with dashed lines (and the corresponding *Artist* subgraph), the instance becomes also minimal (and complete).

If the instance is minimal and non-ambiguous, a unique tree of representative nodes can be associated to a valid block B, with one node for each descendant of B and B itself. Since $\mathcal{G}_I$ is minimal, this tree can be viewed as the trace of a query (i.e., the tree of nodes visited during query evaluation). Given a valid block B and a data graph $\mathcal{G}_I$, we call *generating queries* the queries q such that $B = q(\mathcal{G}_I)$. In general, two non-equivalent queries q and q' may yield the same result on a specific instance $\mathcal{G}_I$. However, when $\mathcal{G}_I$ is a non-ambiguous instance, there exists a unique minimal element (up to equivalence) in the generating set of a block B. Minimality is defined with respect to query (and trace) containment.

Definition 5 (Minimal generating query) *Let B be a valid block on an instance $\mathcal{G}_I$. The* minimal generating query *q of B is the smallest element (up to query equivalence) of the set of generating queries of B according to query containment.*

A syntactic expression of the minimal generating query can be built as follows. First, the tree T of the representative nodes of B in $\mathcal{G}_I$ is computed. One method to achieve this is to consider values from each block as keywords and to perform a search of representative nodes according to these keywords. A simpler approach is to gather information on the representative nodes visited by the user during the interactive construction of the block. The latter solution is applied in our prototype. Second, once the tree of representative nodes T is obtained, the rules of the DocQL query are recursively built according to the following procedure:

```
CreateRule ($B$, $N_p$, $T$)
Input: $B$, a block valid on $\mathcal{G}_I$,
    $N_p$ the representative node of the parent of $B$,
    $T$, the tree of representative nodes.
begin
  Take into tree $T$ the node $N$, representative of $B$.
  $rule.path :=$ the (unique) path from $N_p$ to $N$
  $rule.body :=$ " ";
  for each syntactic element $e$ of $B$ do
    if $e \in \mathcal{F}$ then // $e$ is a static text
      append $e$ to $rule.body$
    if $e \in$ dom // $e$ is a value $v$ from the graph
      append a rule @p to $rule.body$, where $p$ is
      the (unique) path from $N$ to $v$
    if $e \in \mathcal{B}$ // $e$ is a block $B'$, child of $B$
      append the result of the recursive call to
      CreateRule ($B'$, $N$, $T$) to $rule.body$
  end for
  return $rule$
end
```

This procedure is initially called with the root block and the virtual node corresponding to the graph entry point. It is noteworthy that the soundness of this procedure is guaranteed only on a non-ambiguous instance.

This algorithm builds a DocQL query without predicates. The structure of a valid block yields only the specification paths in the database, without the ability to express conditions on the encountered values. In order to complete this specification, the user (assisted by the system) may provide a function f binding to each block B a condition (or a conjunction of conditions). A condition on a block B is defined by $a\theta b$, where θ is a relational comparison operator, and a and b are unique paths or simple values. We can finally define canonical documents:

Definition 6 (Canonical document) *A* canonical document *of a query q is a pair (B, f), where B is a valid block such that q is (equivalent to) the minimal generating query of B, and f is a function that binds a conjunction of conditions to each component of B.*

A Publish-By-Example interface must assist the interactive construction of a canonical document representing the awaited query q, in the most intuitive and simple way. Note that, in order to produce a canonical document characterizing q, all the representative tuples required for block interpretation must be available in the manipulated instance. The following section addresses this issue.

Canonical instances.

The construction of a canonical document D assumes that the instance proposed to the user allows both the construction and the interpretation of D. This gives rise to the question of constructing a specific instance, called *canonical instance*, that allows to build a canonical document for *all* the possible queries over the graph schema.

Definition 7 (Canonical instance) *An instance $\mathcal{G}_I$ of a schema S is a* canonical instance *if, for any query q over S, there exists a canonical document of q on $\mathcal{G}_I$.*

An instance is canonical if it is complete, minimal and non-ambiguous. Completeness is required for allowing all the possible navigations in the graph with respect to the schema, whereas the minimality and non-ambiguity serve to a proper interpretation of a canonical document as a query. Recall that an instance is complete if, during the navigation in the graph, we can find at any moment a choice for each possible path type.

As an example, consider the relational instance of Figure 3, and assume that *Movie* contains only the tuple *Kagemusha*. Suppose that a user wants to produce a publishing query showing a movie with the list of its actors. It is not possible to build a canonical document for this query on this instance, since the casting is unkown for *Kagemusha*. This instance is not canonical.

If, instead of *Kagemusha*, *Movie* contains the tuple *Van Gogh*, we can produce the following canonical document that shows a film, its director and its actors:

```
Van Gogh, 1990, directed by Maurice P-
ialat with :
  - Jacques Dutronc, born in 1935
```

By contrast, the instance containing only film *Van Gogh* is not sufficient to build an example for a publishing query showing a film, its actors, and for each actor, the list of films possibly directed by this actor. Indeed, in this instance *Jacques Dutronc* is not a director. Nevertheless the relationship between an artist and a movie as a director exists, and a user may want to exploit this relationship. Therefore this instance is still not canonical.

Finally, as a last example, consider the instance of Figure 3 in which the only represented movie is *Unforgiven*. This instance allows the construction of the canonical document giving a film, its actors, and the films directed by these actors:

```
Unforgiven, 1992, directed by Clint E-
astwood with :
  - Clint Eastwood, born 1930, as Wil-
  liam Munny also director of ``Unfor-
  given''
```

This document is possible thanks to a cycle into the data graph, instance of the cycle *Movie* $\rightarrow$ *Director* $\rightarrow$ *Actor* $\rightarrow$ *Movie* in the graph schema. The cycle size in the instance is proportional to the cycle size in the schema. With the two nodes *Eastwood* and *Unforgiven*, the instance cycle has a minimal size (two edges). Although satisfying with respect to the completeness of the canonical instance as a support for canonical documents, a shortcoming of a small cycle is to show repeatedly the same node at different places in a document, with a possible confusion on the role of each occurrence. In the previous example, *Eastwood* and *Unforgiven* both appear twice, each time in a different context. This may be misleading to the user, and results in an apparent lack of generality.

The instance can be extended to longer cycles of size $k \times n$, where n is the cycle size in the graph schema and $k \geq 1$ is a parameter of the system. Figure 4.a shows a minimal cycle in our sample instance, and Figure 4.b its generalization to a cycle of length $k \times n$.

Observe that the occurrence of a cycle in the graph schema implies the occurrence of a cycle in the canonical instance, otherwise the instance would not be complete. In case of a path without cycle, the two extreme nodes would be left without "corresponding node", and the ability

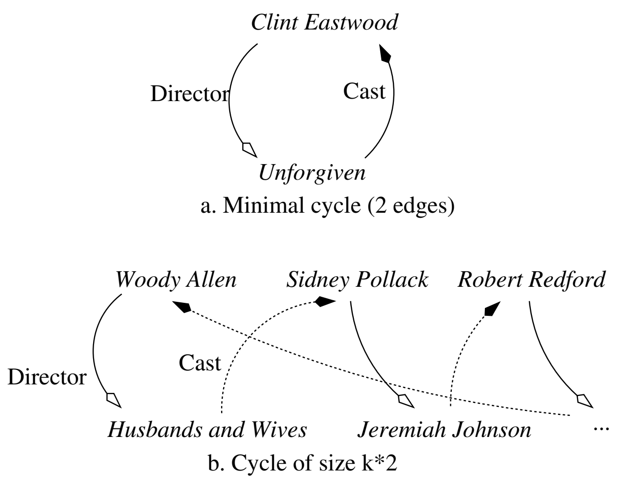

Figure 4. Cycle in a canonical instance

to build a canonical document from these nodes would be compromised.

The production of a canonical instance must ensure that the required properties are verified. If only cycles of minimal size are to be constructed, then the construction algorithm is straightforward: a node is instantiated for each node type of the schema, and an edge between these nodes is instantiated for each edge type in E. We describe in the following a more sophisticated algorithm that takes into account an expansion factor k for cycle size.

The algorithm maintains a global array $nodes_r$ for each node type r of the schema. $nodes_r$ contains the sequence of instances built by the algorithm, denoted $nodes_r[1]$, $nodes_r[2]$, etc. The algorithm returns a path $r_1.e_1.r_2.e_2.\cdots.r_n$, $r_i \in V$ and $e_i \in E$, extended at each recursive call, and representing nodes and edges created during function calls. We use two auxiliary functions on paths:

1. $dist(path, r)$ returns the number of steps in $path$ since the first occurrence of a node of type r;
2. $nb(path, r)$ returns the number of occurrences of a node of type r in $path$;

The algorithm takes as input a node N, the type e of the edge to create, and the path created since the initial call. The global variable K denotes the minimal size required for a cycle.

```
CONSTRUCT ($N$, $e$, $path$)
Input: $N \in V_I$, a node, $e$ an edge type such that
    $N$ is an instance of $initial(e)$, $path$ the path.
begin
  // We extract the type of the terminal node of $e$
  $r := terminal(e)$
  // If it is the first time we reach $r$ in the path:
  // we take the first node of $r$
  if ($r \notin path$) then $i_r := 1$
  // If the first occurrence of $r$ in the path is at distance
  // greater than $K$ : the size of the cycle is satisfying, and
  // again we take the first node of $r$
  else if ($dist(path, r) \geq K$) then $i_r := 1$
  // Otherwise, we use a new instance of $r$, that does not occur
  // in the path
  else $i_r := nb(r, path) + 1$

  // Now $i_r$ denotes the current instance of $nodes_r$
  if ($nodes_r[i_r]$ exists)
    $\mathcal{G}_I += N \xrightarrow{e} nodes_r[i_r]$ ; $\mathcal{G}_I += nodes_r[i_r] \xrightarrow{e^{-1}} N$
    // Stop here: no recursive call needed
  else
    // Instantiate a new node $nodes_r[i_r]$, and create the
    // corresponding edge
    $nodes_r[i_r] := new(r)$;
    $\mathcal{G}_I += N \xrightarrow{e} nodes_r[i_r]$ ; $\mathcal{G}_I += nodes_r[i_r] \xrightarrow{e^{-1}} N$
    // Now, recursive calls are needed, one for each possible
    // path from $nodes_r[i_r]$
    $path := path + e.r$
    for each $e$ in $E$ with $initial(e) = r$ and $terminal(e) \neq N$
      CONSTRUCT($nodes_r[i_r], e, path$)
    end for
  end if
end
```

Algorithm CONSTRUCT must be called for each connected component of the graph schema, taking any relation node type in each component as a starting point for the instance creation.

This algorithm builds a synthetic canonical instance, with somehow meaningless node values. In practice, relying on a real instance would yield more user-friendly node values. However there is no guarantee to find a canonical instance into a real instance. In that case it is necessary to complete the instance with synthetic values, or to link artificially existing but unrelated values.

We implemented a web-based editor and query system[1] for our publication model. The system allows to build canonical documents, derives their associated DOCQL queries and may either immediately evaluate the query on a real instance, or save the query as a named dynamic fragment which can later on be composed with others.

4 Related work

Using graphical interfaces for expressing queries is an old concerns. The early language *Query By Example*

[1]Publicly accessible on the site *http://www.lamsade.dauphine.fr/rigaux/docql*

(QBE) [11] addresses the main principle of such visual tools: the query expression is based on an image of the result. QBE and its variants remain oriented toward the expression of relational queries, and deliver relational tables as result.

The "by example" paradigm has been adapted and extended to semi-structured data and XML document by many proposals: QSByE [4], QURSED[8] and XQBE [2]. QURSED is a web form and report generator, dedicated to the querying of semi-structured data. XQBE proposes an interface to automatically generate XQuery queries. All these tools help users to construct complex queries over directed labeled trees. Queries are displayed with a graph-based representationIn contrast, in our approach, the user does not manipulate a query but a query result. This limits the technical knowledge required from the user, and favors the integration of our tool with document editors.

An implementation of our model could take advantage of keyword-search techniques in relational database [5, 9, 7]. Applied to a canonical instance, they could probably deliver a non-ambiguous graph of representative nodes/tuples. This supports our belief that an interface based on alternative design principles is possible.

The publishing language which constitutes the target of our publishing process can be related to *XML publishing*. The specification of the exported data is usually expressed as a tree of co-related SQL queries and can be viewed as an abstraction of nested cursors over result sets. This is quite similar to the publishing mechanism adopted in the present paper. One can therefore envisage to adapt our example-based approach to XML publishing languages.

Finally we note that our data model is closely related to the field of functional dependencies. In particular the concept of canonical instance shares with Armstrong relations its motivation of building a representative instance to assist the end-user in his designing tasks (see, in particular, [1]). Although we could have used this standard framework in a more direct way, we believe that the tailored approach chosen in the current paper fits more intuitively to our goals. In particular the graph-based representation is much more intuitive to the non-expert user than the scattering of information in relational tables.

5 Conclusion

We propose in this paper a simple and intuitive method for producing publishing programs. We described the formel model which states the main concepts and specially those of canonical documents and canonical instances.

Our publish-by-example mechanism is implemented and we are currently validating our tool with respect to an actual data-intensive web application (namely the MYREVIEW system, *http://myreview.lri.fr*) to check its ability to produce and maintain the set of dynamic fragments that constitute the *view* (presentation) part.

References

[1] C. Beeri, M. Dowd, R. Fagin, and R. Statman. On the Structure of Armstrong Relations for Functional Dependencies. *J. ACM*, 31:30–46, 1984.

[2] D. Braga, A. Campi, and S. Ceri. XQBE : A Visual Interface to the Standard XML Language. *ACM Trans. on Database Systems*, 30:398–443, 2005.

[3] Macromedia ColdFusion MX 7, 2007. http://www.adobe.com/fr/products/coldfusion/.

[4] Irna M. R. Evangelista Filha, Alberto H. F. Laender, and Altigran Soares da Silva. Querying Semi-structured Data By Example: The QSByE Interface. In *Workshop on Information Integration on the Web*, pages 156–163, 2001.

[5] G.Bhalotia, C.Nakhe, A.Hulgeri, S.Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in databases using BANKS. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 431–440, Washington, USA, 2002. IEEE Computer Society.

[6] S. Guéhis, P. Rigaux, and E. Waller. Data-driven Publication of Relational Databases. In *Proc. IEEE Intl. Database Engineering & Applications Symposium (IDEAS'06)*, 2006. Also in BDA'06.

[7] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 670–681, Hong Kong, China, 2002.

[8] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. QURSED: Querying and Reporting Semistructured Data. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2002.

[9] S.Agrawal, S.Chaudhuri, and G.Das. Dbxplorer: A system for keyword-based search over relational databases. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 005–016, Los Alamitos, CA, USA, 2002.

[10] S.Ceri, P.Fraternali, A.Bongio, M.Brambilla, S.Comai, and M.Matera. *Designing Data-Intensive Web Applications*. Morgan-Kaufmann, 2002.

[11] Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.