Publish By Example[*]

Sonia Guéhis
Univ. Paris-Dauphine
sonia.guehis@dauphine.fr

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

Philippe Rigaux
Univ. Paris-Dauphine
& INRIA-Orsay
philippe.rigaux@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.

Our model relies on several components. First we propose a simple formalization of database publishing languages, called DocQL. Second we base our method on the concepts of *canonical documents* and *canonical instances*. A *canonical document* is an example from which one can derive a unique DocQL publishing program. A *canonical instance* of a relational schema is an instance that supports the construction of *all* the possible canonical documents over this schema. We finally describe and comment a visual editor that shows how a user can rely on these concepts for intuitively producing publishing programs.

1. INTRODUCTION

This paper considers 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 *relational 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. This is arguably one of the most widespread type of database application nowadays. We use it for illustration purposes in this paper.

[*]Work supported by the Wisdom project (*http://wisdom.lip6.fr*).

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 [5]. Specialized languages, such as Servlets/JSP, PHP or ColdFusion [6], bring partially satisfying solutions. However, in all cases, writing a database publishing program requires heterogeneous technical skills, 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. In most cases – if not in all – the program structure reduces to a mere imbrication of SQL cursors, each instantiating an HTML fragment. The combination of fragments constitutes the final document. Our conviction is that these programs can be conveniently produced without using the heavy machinery of programming and querying languages whose power and complexity goes far beyond what is necessary for this simple task.

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.

Our approach consists of several components, each covered by a dedicated section in the following. After Section 2 which gives a bird eye's view of our approach, we first propose (Section 3) a simple formalization of relational database publishing as a "document query language", called DocQL, already proposed in preliminary form in [11]. A DocQL query can be seen as a syntax-neutral (declarative) specification of a publishing program written in Java/JSP or in any other programming framework. Producing a DocQL query constitutes the target of the publish-by-example process. Moreover, our goal, given a database schema S, is to be able to create by example *all* the possible DocQL queries

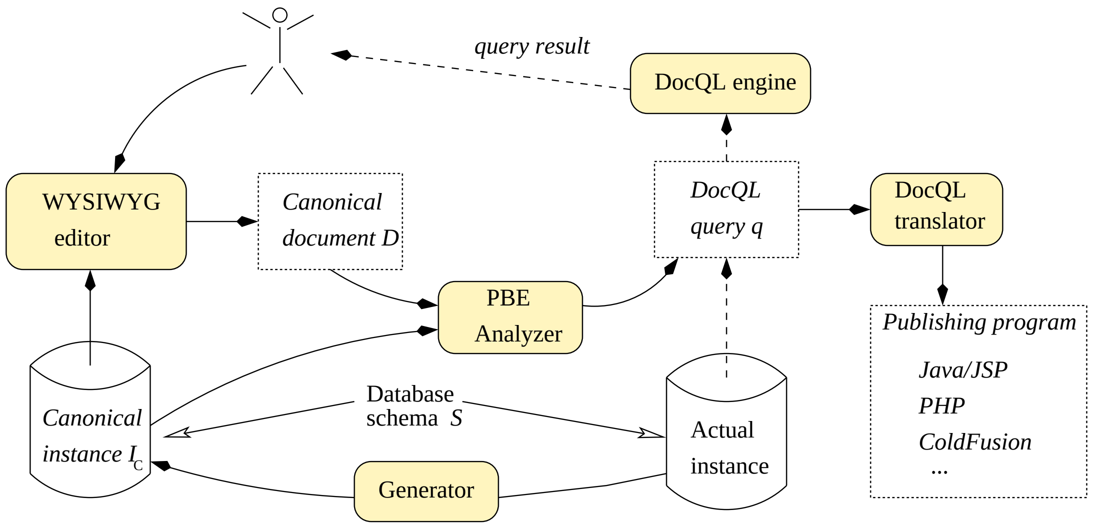


Figure 1: Overview of the publish by example process

(called *publishing queries* in the following) over S.

We then describe (Section 4) our publication model. It 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. Intuitively, a document d is "canonical" for a query q and an instance I of S if $d = q(I)$, and there does not exist a query q' "simpler" than q with $d = q'(I)$. Next we introduce the complementary concept of a canonical instance I_C of S as 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.

Finally our last contribution is an online document editor which demonstrates how, in practice, our publish-by-example mechanism can be implemented (Section 5). We show and comment a short interactive session, discuss the role of the main concepts that support our model, as well as some specific design choices, and point out possible alternatives. Section 6 positions our proposal with respect to the state of the art, and Section 7 concludes the paper.

2. OVERWIEW

Figure 1 presents the main components of our system. The user interacts with a WYSIWYG graphical editor which lets him construct a canonical document D from a canonical instance I_C. The canonical instance is a predetermined instance of the schema S, generated by the system administrator either from synthetic data, or from an actual database instance of S, using an *instance generator*. The canonical instance enjoys some specific properties which allow a non-ambiguous interpretation of the canonical document.

Essentially, the editor enables the navigation in the canonical instance, seen as a graph of tuples and values. The user can position himself on a node in the graph and merge the values associated to the node with character data in order to form so-called *blocks*. At the end of the navigation process, the set of blocks thereby created is organized as a hierarchy that constitutes the canonical document.

Next, the DocQL query is generated by the *analyzer* which takes as input the canonical document and the canonical instance. The query q is inferred without ambiguity by determining the (unique) node from I_C associated to each block of q, and the values that form the dynamic part of the block. The user can then

1. either run the query over the actual instance, through the DocQL engine which directly evaluates q;

2. or translate q to a traditional publishing program, written in any convenient language.

Note that a publishing program can reversely be interpreted as a query q, and then manipulated through the WYSIWYG tool which actually shows the query/program as $q(I_C)$, i.e., as the result on the evaluation of q over the canonical instance. With respect to software engineering purposes, our approach can be viewed as a programming-by-example framework, dedicated to the specific area of dynamic document production.

In the rest of this paper we illustrate our approach over a sample movie database with the following schema:

- Movie (**title**, year, *id_ director*, genre)
- Artist (**id**, last_name, first_name)
- Cast (***title***, ***id_ actor***, character)

The schema represents movies with their (unique) director and their (many) actors. Primary keys are in bold, and foreign keys in italic. Figure 2 shows a database instance.

3. THE PUBLISHING LANGUAGE DocQL

We give the main features of the publishing query language DocQL. Due to space limitations, we limit the presentation to the definitions which are useful to the publication model presented in the next section. Details on optimization and evaluation techniques can be found in [11].

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

3.1 Data model

DocQL aims at a concise specification of publishing programs. Here *concise* means that the language captures with a uniform and simple syntax the queries and programming instructions used to build dynamic documents. More specifically any DocQL query q is equivalent to a publishing program built with the following operators:

- *fragment construction* of the form `block(t);`
- *loops* of the form `while (t := fetch C) block(t);`
- *conditional statements* of the form `if (cond(t)) block1(t); else block2(t);`

Here, `C` is a cursor over the result of a conjunctive SQL query, `t` is a tuple variable and $\texttt{block}_i\texttt{(t)}$ constructs a textual fragment, from static text, values from `t` or other fragments produced by embedded loops or conditional statements. The intuition is that the loops and tests in such programs depend only on the database instance, and that the basic operation is the creation of fragments from database values. The structure of the program dictates the assembling of the fragments that form the final document. Our experience and practice is that this simple, "data-driven", model covers the majority of publishing requirements.

DocQL 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.

Formally, let $\mathcal{T}, \mathcal{R}, \mathcal{A}$ be sets of symbols pairwise disjoint, $\mathcal{T}$ finite, and $\mathcal{R}, \mathcal{A}$ countably infinite. Elements of $\mathcal{T}$ are called *atomic types*, those in $\mathcal{R}$ *relation names*, and those in $\mathcal{A}$ *attribute names.*

Definition 1 (Schema). *A* (graph database) schema *is a directed labeled graph (V, E, λ, μ) with the following structure:*

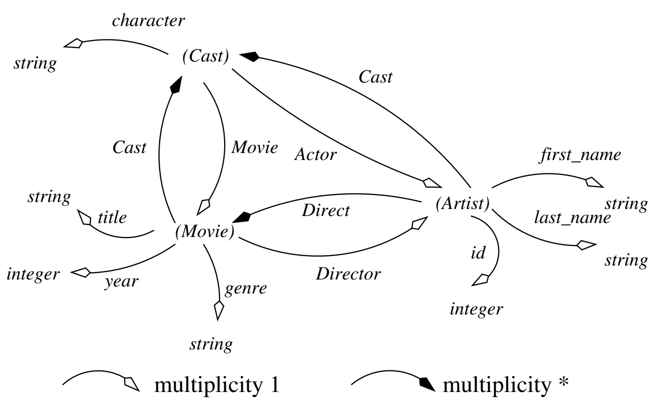


Figure 3: The graph schema

1. *$V \subseteq \mathcal{T} \cup \mathcal{R}$ is a set of vertex, and $E \subseteq (V \cap \mathcal{R}) \times V$ is a set of edges;*
2. *λ is a labeling function from E to $\mathcal{R} \cup \mathcal{A}$ such that, if e and e' are two edges with same initial vertex r, then $\lambda(e) \neq \lambda(e')$;*
3. *μ is multiplicity function from E to $\{1, *\}$;*
4. *if $e \in E$ is of the form $r \overset{\lambda(e)}{\rightarrow} s$, with $r, s \in \mathcal{R}$, there exists an edge $e' \in E$ of the form $s \overset{\lambda(e')}{\rightarrow} r$, called the* reverse edge *of e.*

Figure 3 shows the graph schema of our *Movies* database. The mapping that transforms a relational schema to an equivalent graph schema is based, by default, on the naming of tables and attributes and on the (primary key, foreign key) correspondence. This default choice can be refined or extended in the mapping file that declares the conversion from I to $\mathcal{G}_I$. For instance the link from *Movie* to *Artist* which represents the association between *id_ director* (foreign key in *Movie*) and **id** (primary key in *Artist*) is labeled *Director* in Figure 3 for clarity purposes.

Now let $\mathcal{I}$ be a countably infinite set of *tuple identifiers*, and for each atomic type $\tau \in \mathcal{T}$ let be given the set of values of this type, denoted $[\tau]$.

Definition 2 (Instance). *Let $S = (V, E, \lambda, \mu)$ be a schema. An* instance *$\mathcal{G}_I = (V_I, E_I)$ of S is a mapping from S to rooted labeled graphs defined as follows:*

1. *for each $v \in V$* $\begin{cases} V_I(v) \subset \mathcal{I} \textit{ if } v \in \mathcal{R} \textit{ (tuple to tuple)} \\ V_I(v) \subset [v] \textit{ if } v \in \mathcal{T} \textit{ (tuple to value)} \end{cases}$
2. *if $e \in E$, then each instance of e is of the form $x \xrightarrow{a} y$, with $x \in V_I(initial(e))$, $y \in V_I(terminal(e))$, and $a = \lambda(e)$; moreover, if $\mu(e) = 1$, there does not exist two instances of e with the same initial vertex.*

If N_1 and N_2 are two nodes in the data graph, we note $N_1 \xrightarrow{p} N_2$ if there exists a path p from N_1 to N_2 such that, for each edge of p instance of e, $\mu(e) = 1$. We say that p is an instance of a *unique path*, or that N_2 functionally depends on N_1. If there is at least one edge of p instance of e with $\mu(e) = *$, then p is an instance of a *multiple path*, and we note $N_1 \overset{p}{\twoheadrightarrow} N_2$.

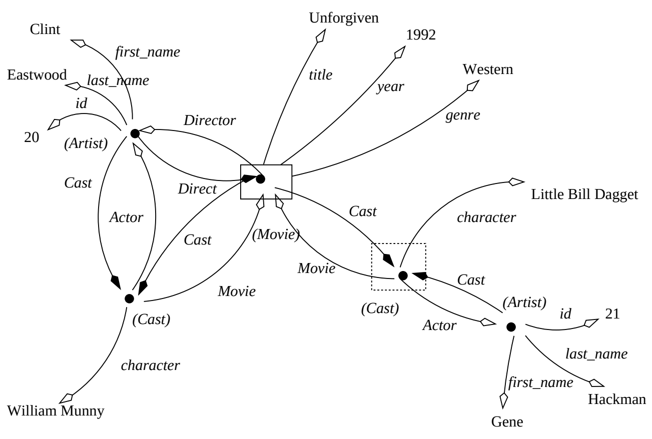


Figure 4: The data graph of our sample instance

Vocabulary. In the following, 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) $N \overset{p}{\twoheadrightarrow} N'$.

Figure 4 shows the graph of the instance of Figure 2. Consider the node (of type *Movie*) in the rectangle. 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*.

3.2 Query language

DocQL combines *navigation* in the data graph with *instantiation* 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{fragment}`, where `path` is a path expression 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[1]. 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. Formally:

Definition 3 (Trace of a query). *Let q be a* DocQL *query, represented as a tree of rules, and $\mathcal{G}_I$ be a data graph. The* trace *$\mathcal{T}_q(\mathcal{G}_I)$ of q with respect to $\mathcal{G}_I$ is a tree of pairs (N, r), where N is a node of $\mathcal{G}_I$ and r is a rule from q, defined inductively as follows:*

1. *if q is the empty query* `@db{}`*, then $\mathcal{T}_q(\mathcal{G}_I) =$* `(root, db)`*, where* `root` *is the pseudo-root of $\mathcal{G}_I$;*
2. *if q is of the form $q' \oplus (r', r)$, where $\oplus$ denotes the extension of the tree q' with a rule r child of r' in q', then $\mathcal{T}_q(\mathcal{G}_I)$ is obtained from $\mathcal{T}_{q'}(\mathcal{G}_I)$ by adding as children of each node (N', r') in $\mathcal{T}_q(\mathcal{G}_I)$ the nodes $N \in \mathcal{G}_I$ such that $N' \xrightarrow{r} N$.*

The result of a query is obtained by "decorating" the nodes of its trace with the (static) character data of their associated rules. Looking at the previous example, we first search for the node *Movie* with title *Unforgiven.* Taking this node as an initial one, the value of each (unique) path `title`, `year`, etc., is evaluated. The multiple path `Cast` leads to all the nodes that represent one of the characters of *Unforgiven.* Applied to the data graph of Figure 4, one obtains the following document:

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

4. THE PUBLISH BY EXAMPLE MODEL

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

4.1 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 $\mathbf{dom} \subset \Sigma^*$ denotes the active domain of $\mathcal{G}_I$. For the sake of simplicity, we suppose that $\mathcal{F} \cap \mathbf{dom} = \emptyset$, in order to distinguish elements from theses two sets. In practice, the distinction may rely on syntactical convention (for instance, a tag: see Section 5). We also assume a set $\mathcal{B}$, distinct from the previous ones, of *block identifiers.*

Definition 4 (Block). *A* block *B is a pair (i, b), where $i \in \mathcal{B}$ is the* block identifier *and $b \in (\mathcal{F}|\mathbf{dom}|\mathcal{B})^*$ 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.

[1]A complete description would also include tests which allow to express negation [11]. We are omitting them for the sake of simplicity.

Definition 5 (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 $v \in$ **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 of 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 4).

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 6 (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 4) 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.

4.2 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, and show that

1. if $\mathcal{G}_I$ is minimal and non-ambiguous, there exists a unique interpretation of a valid block B as a publishing query;
2. if, in addition, $\mathcal{G}_I$ is complete, then *all* the publishing queries over a given schema can be characterized by a block valid on $\mathcal{G}_I$.

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 7 (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 4 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 4, 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 4 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). Note the cycle that corresponds to a cyclic relationship in the graph schema.

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 (see Def. 3). 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:

$$q \subseteq q' \text{ if and only if } \mathcal{T}_q(\mathcal{G}_I) \subseteq \mathcal{T}_{q'}(\mathcal{G}_I), \forall \mathcal{G}_I$$

We associate this minimal element to B:

Definition 8 (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 relation $\subseteq$.*

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. A general 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 *à la* Banks [3] or DBDiscover [12]. 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 described in Section 5. 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 9 (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. The prototype described in Section 5 is a proposal in this direction. 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.

4.3 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. There exists two possibilities:

1. either the user provides, along with the construction of the document, the representative nodes and values which are (temporarily or not) inserted into the instance and later used to determine the corresponding publishing query;
2. or the publication system offers the user a set of predefined nodes and values for the construction of the canonical document.

The first choice reduces to a user interface problem, discussed in the next section. The second 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 10 (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 iff 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 4, 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 Pialat
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 4 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 Eastwood
With :
  - Clint Eastwood, born 1930, as William Munny
     also director of ``Unforgiven''
```

This document is possible thanks to a cycle into the data graph, instance of the cycle *Movie* → *Director* → *Actor* → *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 $k \geq 1$. Figure 5.a shows a minimal cycle in our sample instance, and Figure 5.b its generalization to a cycle of length $k \times n$.

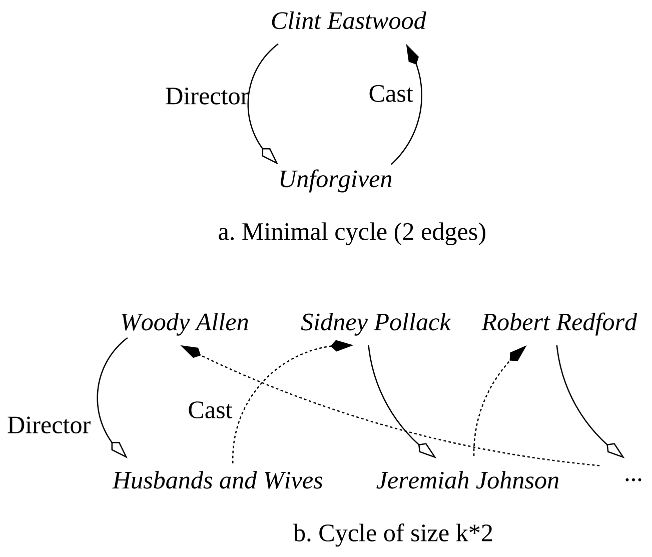


Figure 5: Cycle in a canonical instance

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 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.

5. EDITING PUBLISHING PROGRAMS

We implemented a web-based editor and query system[2] 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.

In this section we first comment our objectives and design choices. We then illustrate the practical aspect of our work by showing and commenting an interactive session.

5.1 Objectives and design choices

Our main objective is to investigate the ergonomic issues: how does the user interact with the system, how does he navigate in the structure of the canonical document, what is the amount of structural information which has to be shown, etc. The design of our interface is an answer to these questions, based on attempts to produce significant publishing queries. We comment our choices in the following, and point out occasionnally possible alternatives. Another objective of this implementation is to enrich the model with several practical complements (for instance, taking account of environment

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

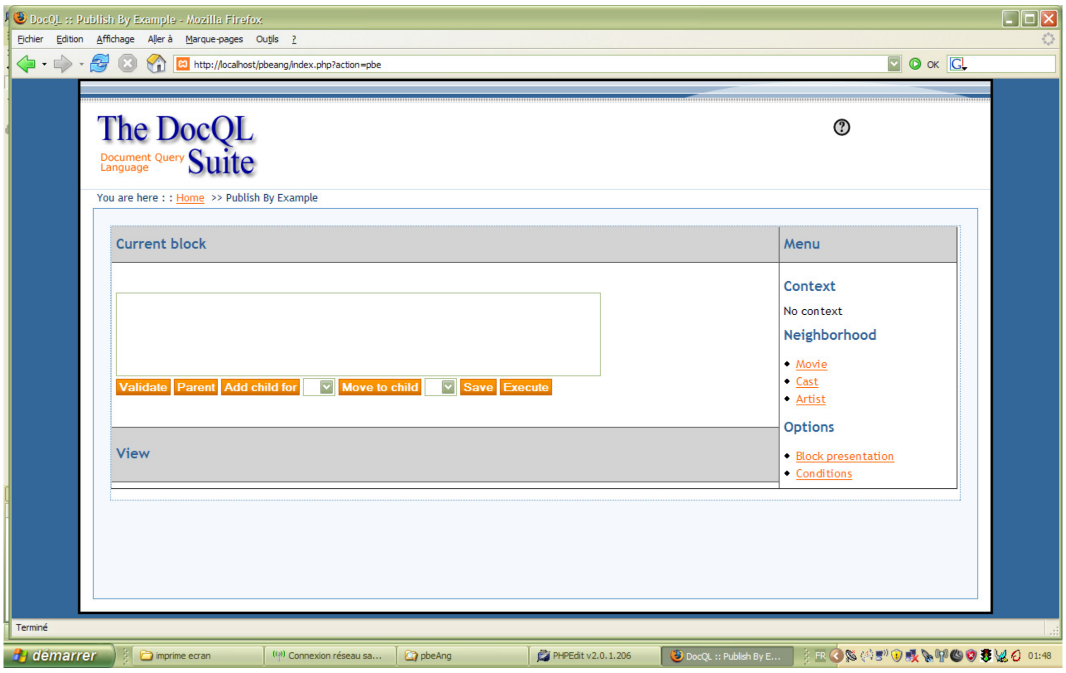

DocQL :: Publish By Example - Mozilla Firefox
Fichier Edition Affichage Aller à Marque-pages Outils ?
http://localhost/pbeang/index.php?action=pbe
The DocQL Suite
Document Query Language
You are here : : Home >> Publish By Example
Current block
Validate Parent Add child for Move to child Save Execute
View
Menu
Context
No context
Neighborhood
Movie
Cast
Artist
Options
Block presentation
Conditions
Terminé
demarrer

Figure 6: Initial state of the editor

variables) which make it usable as a real data-centric Website production tool. Due to space limitations we do not develop this aspect.

As mentioned before, an initial design decision is to choose among two possibles scenarios. The first one relies on a canonical instance. The user benefits from pre-existing paths, tuples and values, and his only remaining concern is to organize this information in a document. In the second scenario, the user creates an *ad hoc* instance by instantiating representative tuples each time a new block is created. Our system adopts the first choice which, in our opinion, leads to a much more intuitive and easy-to-use tool.

A second important design choice is to acquire and maintain, during the course of a session, some structural information about the document under production. This information is used later on to produce the query without having to analyse the document's content in order to identify its block-based organization and distinguish static parts from dynamic ones. A downside of this choice is that it somehow burdens the user with navigation constraints (i.e., the block structure is explicit, and the user edits only one block at a time). Note that this remains a design choice for this specific implementation, and not a constraint of the model.

The session presented in what follows aims at producing the query of Example 1, which outputs a document showing a movie with its director and the list of its actors.

5.2 Overview of the graphical interface

Figure 6 shows the initial state of the interface, before any user input devoted to the DocQL language. It consists of three sub-parts of the window entitled *Publish By Example*.

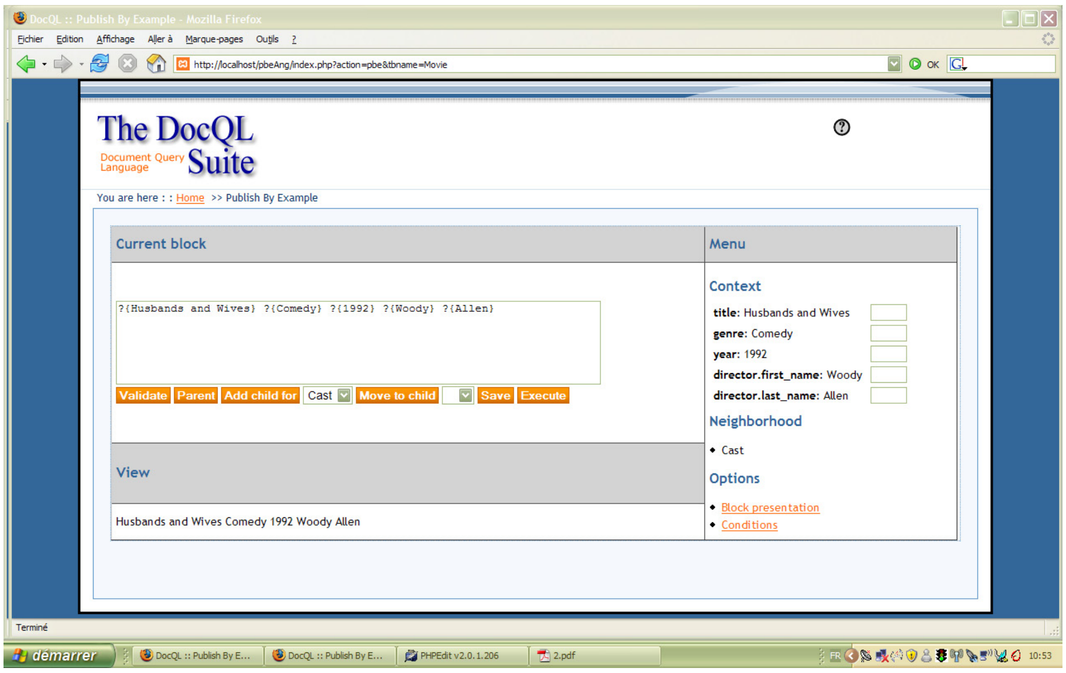
- The right part (*Menu*) presents the *context*, the *neighborhood* and some advanced options for the production of the queries, briefly presented at the end of this section;
- the left part (*Current Block*) is a window that serves to edit a block of a canonical document;
- finally the left-bottom part, called *View*, shows the canonical document whose creation is in progress.

Initially, both the editing window and the view are empty, as well as the *Context* part of the *Menu*. The neighborhood proposed to the user consists of all the access paths to the data graph, each path being referred to by its label. In our session, three paths are available: *Artist*, *Cast* and *Movie*. Note that the default label is simply the table name. This can easily be adapted at the interface level.

5.3 Creating a root block

Initially, choosing a path in the neighborhood is tantamount to defining the type of the node associated to the root block of the canonical document. The system then picks up a representative node for this block in the canonical instance, and proposes the context values (i.e., those that functionally depend on the node), both in the *Context* part, and in the editing window. Figure 7 shows the editor once the initial path *Movie* is chosen.

- **In the *Menu* part.** Each value v of the *Context* is associated first to a label which is, by default, the (unique) path in the data graph that leads from the representative node to v, and second to an input field which allows to express selection criteria. The *Neighborhood* part shows all the paths that lead from a representative node to a node in the neighborhood. In this case the only possible path is *Cast*. The *Option* is context-independent (see the discussion at the end of the section).
- **In the *Current block* part.** The system puts in the editing window, whenever a block is created, the set



You are here : : Home >> Publish By Example

Current block

```
?{Husbands and Wives} ?{Comedy} ?{1992} ?{Woody} ?{Allen}
```

View

Husbands and Wives Comedy 1992 Woody Allen

Menu

Context

title: Husbands and Wives
genre: Comedy
year: 1992
director.first_name: Woody
director.last_name: Allen

Neighborhood

- Cast

Options

- Block presentation
- Conditions

Figure 7: After choosing the initial path *Movie*

of values of the context. In order to make the DocQL query generation easier, we chose to mark the context values with a specific syntax which distinguish them from the free text provided by the user. This is a debatable choice which is discussed below.

- **In the *View* part.** The system shows the current state of the canonical document which is reduced, at this point, to the values of the root block's context.

Let us now focus on the markers of the text fragments that represent "dynamic" values. Two types of markers are currently used:

1. the marker `?{value}`, characterized by the question mark, denotes an *example value* which is actually instantiated to the value retrieved from the database when the DocQL query is evaluated.

2. the marker `!{value}`, characterized by the exclamation mark, denotes a *fixed value*: the DocQL query only retrieves the nodes having this value for the corresponding attribute (in other words this denotes a selection, and a mean to express conditional statements). The fixed value is given by the user in the field associated to the attribute/path in the *Context* part.

Several other markers can be envisaged. For instance `${value}` which denotes a selection with respect to a variable value, `<{value}` and `>{value}` with $<$ and $>$ predicates, etc.

Note also that if we did not choose to rely on a canonical instance, the user would have to provide the context values each time a new block is created in the canonical document. We consider this as an unnecessary invitation to complex manipulations and decisions.

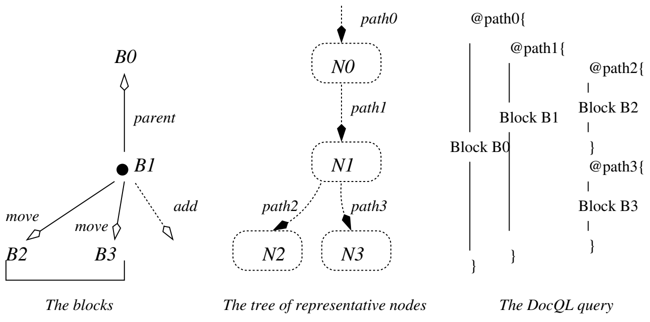
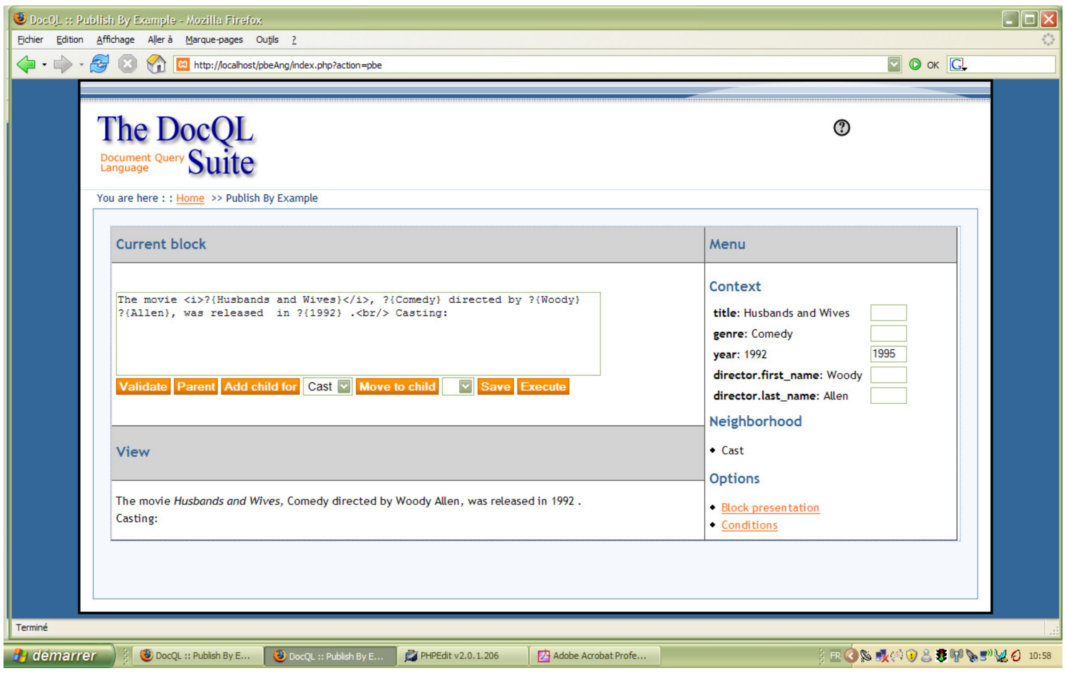


Figure 9: Parallel navigation in blocks and nodes, and the associated DocQL query.

The user can access the editing window and modify the block content, adding free (static) text, XHTML tags or LaTeX commands, all mixed with context values. Figure 8 shows the result of organizing the root block content. Figure 8 also shows a selection: the value 1995 has been associated to the `year` path of the context. The marker becomes accordingly an exclamation mark that indicates a fixed value in the block.

5.4 Adding child blocks

The user can extend the blocks hierarchy of the canonical document, and can navigate in this hierarchy. This can be done with the three buttons located between the editing window and the view, which propose respectively (i) a move from the currently edited block to its parent, (ii) the creation of a child block of the current block, following a selected path to the neighborhood, (button *Add child*, and its associated select menu), (iii) a move toward one of the existing child block (button *Move to*, and select menu of the child blocks).



```
The movie <i>?{Husbands and Wives}</i>, ?{Comedy} directed by ?{Woody}
?{Allen}, was released  in ?{1992} .<br/> Casting:
```

title: Husbands and Wives		
genre: Comedy		
year: 1992	1995	
director.first_name: Woody		
director.last_name: Allen		

The movie *Husbands and Wives*, Comedy directed by Woody Allen, was released in 1992 .
Casting:

Figure 8: Block editing: free text intermixed with context values.

Generally, each move from one block to another in the canonical document corresponds, in parallel, to a positioning on a representative node in the data graph of the canonical instance. Figure 9 shows the parallel interpretation of the three operations *Parent*, *Add child*, *Move to child* with respect to the document structure on the one hand, and to the sub-graph of the representative nodes used as context of each block on the other hand.

Triggering operation *Add child* from the current block B_1 adds for example to the document structure of Figure 9 a block B_4 child of B_1, and associates to B_4 a representative node N_4 taken from the canonical instance.

Back to our session, Figure 10 shows the editor state after creation of a child block of the root block, following the only available path *Cast*. The system proposes a representative node, in that case the actor's name (*Sidney Pollack*) and character (*Jack*) from the casting of *Husbands and Wives*. The view then shows the canonical document obtained by combining the two blocks.

The user can further enrich the hierarchy of his document, adding for instance a child block following the *Direct* path. The system chooses in that case as a representative node a movie directed by *Sidney Pollack*. The navigation operators make also possible to move upward the root block in order to add new child. Once the document is complete (or, actually, at any step during its construction), the query can be generated (*Save* button) and/or executed over a real instance (*Execute* button). In the first case the document designer can build progressively a collection of dynamic fragments whose combination constitutes the dynamic site. The second case corrresponds to a simpler interactive use of the tool, in the spririt of QBE, where the result consists of a hierarchical document. Here is the query produced from the canonical document obtained at the end of our simple session.

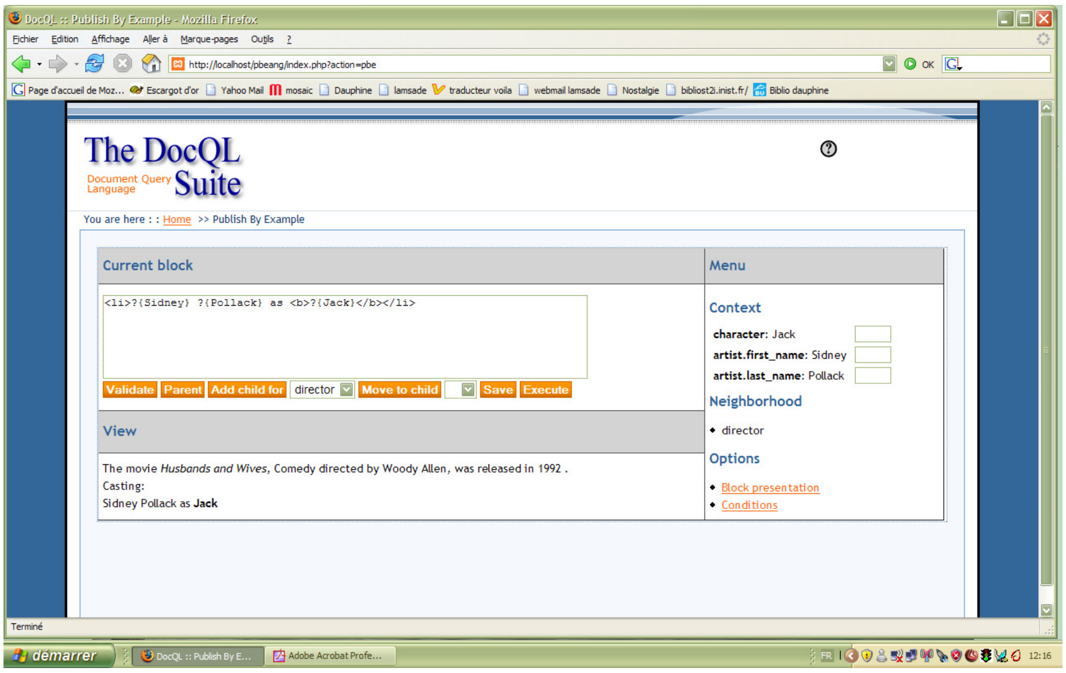
```
@db.Movie[year=1995]{
  The movie <i>@title{}</i>, @genre{},  directed by
    @director.first_name{} @director.last_name{}<br/>
    was released in @year{}.<br/>Casting:
   <ul>
    @Cast{
     <li> @artist.first_name{} @artist.last_name{}
             as <b>@character{}</b></li>
    }
   </ul>
}
```

Note finally that our system produces a DocQL query which can be directly executed over the database, and takes place in our publication framework. Depending on the programming environment, it would just be as easy to generate a program based on more traditional technologies (e.g., JSP/Java).

5.5 Discussion

The short session presented above shows how one can obtain in practice an implementation of our publication model that lets the user produce a publication program with minimal technical knowledge. We are aware that this remains a prototype that can be improved in many ways. We now discuss the following aspects: ergonomy, expressiveness and integration to the other modules of a publication framework.

The ergonomy of our editor remains (relatively) limited, although it reaches its goal of hiding most of the technical concepts to the user. An improvement would be to make transparent the navigation in the blocks of the document. This could be achieved by showing the canonical document as a single editing unit, switching smoothly from one context to a neighbor one as the user executes editing operations on the text. Note however that there may be some ambiguity on the exact border between the occurrences of two blocks,



```
<li>?{Sidney} ?{Pollack} as <b>?{Jack}</b></li>
```

character: Jack
artist.first_name: Sidney
artist.last_name: Pollack

The movie *Husbands and Wives*, Comedy directed by Woody Allen, was released in 1992 .
Casting:
Sidney Pollack as **Jack**

Figure 10: Editing a child block

in which case some explicit information of the border should be asked to the user. Another feature of our prototype is to mark visually the values that come from the database. These syntactic markers should be made invisible in a more sophisticated system. Another turnaround would be to give up the idea of marking database values in the document. This would necessitate however a non-trivial work to extract these values at query-production time. We recall that one of the design choice of our implementation is to enforce the production of *valid* blocks and document, and to keep track, during the production of these blocks and document, of the necessary structural information that allows to produce the query with minimal analysis effort.

As any model, ours needs to be completed with extensions that strengthen its practical scope. We introduced in our prototype several options which correspond to extended functionalities of the DocQL language. A simple example is the declaration and use of environment variables, such as the HTTP parameters transmitted by a user request. We do not elaborate further since none of the extension considered so far conflicts with the core principles of our model.

This last comment leads to the issue of integrating a publish-by-example module to a general-purpose software production platform. A first target of our work is the family of WYSIWYG web-pages editors (e.g., BlueFish, *http://bluefish.openoffice.nl* or its many commercial alternatives). These softwares are pretty good at producing complex but static pages. They also support integration of programming parts when dynamic content is required. We believe that the proposed mechanism, which associates the block structure of a document to navigation paths in a data repository, constitutes a relatively simple extension. It is likely to enable the production of dynamic document by non-database designers with limited additional expertise acquisition.

More generally we advocate the role of a publish-by-example mechanism in applications that rely on a Model-View-Controler architecture. In such contexts it is now widely accepted that the *view* component consists of a combination (decided at run-time by the controler) of static and dynamic fragments. A publish-by-example module allows to produce safe code (no mistake in SQL queries), quickly and easily, with all the potential of a declarative approach (i.e., optimization, verification, security).

6. RELATED WORK

Using graphical interfaces for expressing queries is an old concerns. The early language *Query By Example* (QBE) [16] addresses the main principle of such visual tools: the query expression is based on an image of the result. QBE gave rise to several commercial languages such as Paradox or Microsoft Access [7]. 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: BBQ [13], QSByE [10], QURSED[14], Xing [8], and XQBE [4]. QURSED is a web form and report generator, dedicated to the querying of semi-structured data. It focuses on the specification of form elements, their association with conditional statement with respect to an XML schema, and the generation and evaluation of XQuery queries from the specifications and conditions. XQBE proposes an interface to automatically generate XQuery queries. The workspace is divided in two zones that correspond respectively to the source document(s), over which conditional statements are expressed, and to the result space which describes how the result is to be constructed from the source. All these tools help users to construct complex queries over directed labelled trees. Queries are displayed with a graph-based representation, following a trend initiated by the G-Log language [15]. In 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 [3, 1, 12]. All these proposals do not require a knowledge of the database schema and model the instance as a directed graph. 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*, i.e., exporting existing relational data in an XML view (see [9] for a recent survey and comparison). 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, [2]). 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 paticular the graph-based representation is much more intuitive to the non-expert user than the scattering of information in relational tables.

7. CONCLUSION

We propose in this paper a simple and intuitive method for producing publishing programs. Our proposal relies on two description levels: a formal model which states the main concepts, and an implementation which follows some pragmatic guidelines, such as the choice of building all the documents over a canonical instance which provide, in all circumstances, ready-to-use examples to the document designer. We also choose an approach that imposes the construction of "valid" documents that can be interpreted directly as publishing queries.

Our current work focuses on three complementary issues. First we want to experiment less constrained interaction where the user can freely edit any part of the document without having to navigate from one block to another. This gives rise to some specific problems regarding the identification of blocks, and the distinction between static and dynamic values. Second we aim to enrich the considered data sources, including XML documents, and transient data dynamically produced by an application. Since our data representation is close to semi-structured data model, we believe that this gives us the flexibility to incorporate and integrate in a consistent repository all the data which needs to be accessed by a publishing application. Finally 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.

Acknowledgment. We are grateful to Emmanuel Waller for his useful comments and advices.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, 2002.

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

[3] 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)*, 2002.

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

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

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

[7] M. corp. *Microsoft Office Access*. http://office.microsoft.com/fr-fr/access/default.aspx.

[8] M. Erwig. Xing: A Visual XML Query Language. *Journal of Visual Languages and Computing*, 14(1), 2003.

[9] W. Fan, F. Geerts, and F. Neven. Expressiveness and Complexity of XML Publishing Transducers. In *Proc. ACM Symp. on Principles of Database Systems*, pages 83–92, 2007.

[10] I. M. R. E. Filha, A. H. F. Laender, and A. S. da Silva. Querying Semistructured Data By Example: The QSByE Interface. In *Workshop on Information Integration on the Web*, pages 156–163, 2001.

[11] 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.

[12] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2002.

[13] K. D. Munroe and Y. Papakonstantinou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In *Proc. Intl. Conf. on Visual Database Systems*, 2000.

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

[15] J. Paredaens, P. Peelman, and L. Tanca. G-Log: A Graph-Based Query Language. *IEEE Trans. Knowl. Data Eng.*, 7(3), 1995.

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