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 [3]. Specialized languages, such as Servlets/JSP, PHP or ColdFusion [4], 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 it 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.

**Overview of the approach**

Fig. 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 [8]. 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 over a canonical instance IC of a schema S. The user can then

![Fig. 1. Overview of the publish by example process](image)
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 with the following schema:

- Movie \((\text{title}, \text{year}, \text{id\_director}, \text{genre})\)
- Artist \((\text{id}, \text{last\_name}, \text{first\_name})\)
- Cast \((\text{title}, \text{id\_actor}, \text{character})\)

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 simple database instance.

<table>
<thead>
<tr>
<th>\text{title}</th>
<th>\text{year}</th>
<th>\text{id_director}</th>
<th>\text{genre}</th>
<th>\text{id}</th>
<th>\text{last_name}</th>
<th>\text{first_name}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unforgiven</td>
<td>1992</td>
<td>20</td>
<td>Western</td>
<td>20</td>
<td>Eastwood</td>
<td>Clint</td>
</tr>
<tr>
<td>Van Gogh</td>
<td>1990</td>
<td>29</td>
<td>Drama</td>
<td>21</td>
<td>Hackman</td>
<td>Gene</td>
</tr>
<tr>
<td>Kagemusha</td>
<td>1980</td>
<td>68</td>
<td>Drama</td>
<td>29</td>
<td>Pialat</td>
<td>Maurice</td>
</tr>
<tr>
<td>Absolute Power</td>
<td>1997</td>
<td>20</td>
<td>Crime</td>
<td>30</td>
<td>Dutronc</td>
<td>Jacques</td>
</tr>
<tr>
<td>Absolute Power</td>
<td>1997</td>
<td>21</td>
<td>President</td>
<td>30</td>
<td>Kurosawa</td>
<td>Akira</td>
</tr>
</tbody>
</table>

**Fig. 2.** An instance of the Movies database

In the rest of this paper, Section 2 briefly introduces the DocQL language. We then describe the publication model in Section 3. An online document editor which demonstrates how, in practice, our publish-by-example mechanism can be implemented, is presented in Section 4. Section 5 positions our proposal with respect to the state of the art, and Section 6 concludes the paper.

### 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 [8].

**Data model**

DocQL aims at a concise specification of publishing programs. The language captures with a uniform and simple syntax the queries and programming
Fig. 3. The data graph of our sample instance

instructions used to build dynamic documents. It relies on a navigation mechanism in an instance $I$ modeled as a labeled directed graph $G_I$. 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 Fig 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 $N_1 \xrightarrow{\text{director.last\_name}} N_2$, and $\text{director.last\_name}$ is a unique path. Else we note $N_1 \xrightarrow{p} N_2$ 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) $N \xrightarrow{p} N'$. Consider again Fig 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 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 \( \text{db} \) 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 \text{path} is a path expression and \text{fragment} is the fragment instantiated for each instance of \text{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.

```xml
@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 \( G_I \) 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 Fig 3, one obtains the following document as result of the previous example:

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

The expressive power of the language is that of conjunctive SQL with outer joins. Aggregation and negation cannot be directly expressed in DocQL, but aggregated valued can be obtained via the mapping that transforms the relational instance to the virtual data graph (an even simpler solution is to define SQL views with \text{group by} clauses, which can then be exported in the data graph). This expressive power matches that of standard publishing frameworks, such as Microsoft XML [11]. See also [7] for an in-depth analysis of database publishing expressiveness and complexity.
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 \( G_I \).

Let \( \Sigma \) be an alphabet. \( \mathcal{F} \subset \Sigma^* \) denotes the set of static fragments, and \( \text{dom} \subset \Sigma^* \) denotes the active domain of \( G_I \). For the sake of simplicity, we suppose that \( \mathcal{F} \cap \text{dom} = \emptyset \), in order to distinguish elements from these two sets. In practice, the distinction may rely on syntactical convention (for instance, a tag; see Section 4). We also assume a set \( B \), distinct from the previous ones, of block identifiers.

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

We are interested in blocks that can be unambiguously interpreted with respect to \( G_I \). We first define the notion of representative node of a block.

Definition 2 (Representative node of a block) A node \( N \in G_I \) is representative of a block \((i, b)\) if and only if each value \( v \in \text{dom} \) 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 \( \text{dom} \) 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 Fig 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 \( N_i \).

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 Fig 3) because we can find a node \( N_2 \) (framed with dotted lines), representative of \( B_2 \) in the
neighborhood of $N_1$, with $N_1 \xrightarrow{\text{Cast}} N_2$. 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 $G_I$, our goal is to define a mapping that uniquely determines a query $q$ from $B$ and $G_I$. A complementary question is to know, given a query $q$, whether there exists a block $B$ on $G_I$ that determines $q$. We introduce three constraints on $G_I$: completeness, minimality and non-ambiguity. An instance is said *complete* if, for each node $N$ of type $r \in R$, 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* if there is at most one such edge. The *non-ambiguity* condition is defined as follows:

**Definition 4 (Non-ambiguous instance)** An instance $G_I$ 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. $N' \xrightarrow{p} N'$);
- if $N_1$ and $N_2$ are two nodes of the neighborhood, then $\text{context}(N_1) \cap \text{context}(N_2) = \emptyset$.

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 Fig 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 Fig 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). 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 $G_I$ is minimal, this tree can be viewed as the trace of a query. Given a valid block $B$ and a data graph $G_I$, we call *generating queries* the queries $q$ such that $B = q(G_I)$. In general, two non-equivalent queries $q$ and $q'$ may yield the same result on a specific instance $G_I$. However, when $G_I$ 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. We associate this minimal element to $B$:
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 relation \( \subseteq \).

A syntactic expression of the minimal generating query can be built as from the tree \( T \) of the representative nodes of \( B \) in \( \mathcal{G}_I \). 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. A simpler and sounder approach consists in gathering 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 4.

Note that 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 \( \theta \) 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 \( B \).

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. 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, 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.

Definition 7 (Canonical instance) An instance \( G_I \) of a schema \( S \) is a canonical instance if, for any query \( q \) over \( S \), there exists a canonical document of \( q \) on \( G_I \).

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.

As an example, consider the relational instance of Fig 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 unknown 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. 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, the instance of Fig 3 in which the only represented movie is *Unforgiven* 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* $\rightarrow$ *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. 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 4.a shows a minimal cycle in our sample instance, and Fig 4.b its generalization to a cycle of length $k \times n$.

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 give in the appendix a more sophisticated algorithm that takes into account an expansion factor $k$ for cycle size.

4 Editing publishing programs

We implemented a web-based editor and query system\(^3\) 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.

Our main objective is to investigate the ergonomic issues: interaction with the system, navigation in the structure of the canonical document, the amount of structural information which has to be shown, etc. The session presented in what follows aims at producing a query which outputs a document showing a movie with its director and the list of its actors.

Overview of the graphical interface

Figure 5 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

\(^3\) Publicly accessible on the site http://www.lamsade.dauphine.fr/rigaux/docql
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.

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.

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 6 shows the editor once the initial path Movie is chosen.

![Screenshot of the editor with Movie selected as the initial path](image)

**Fig. 6.** After choosing the initial path *Movie*

- **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 \textit{Cast}. The \textit{Option} is context-independent (see the discussion at the end of the section).

- In the \textbf{Current block part}. The system puts in the editing window, whenever a block is created, the set 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 \textbf{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 \texttt{?\{value\}}, denotes an \textit{example value} which is actually instantiated to the value retrieved from the database when the DocQL query is evaluated.

2. the marker \texttt{!\{value\}}, denotes a \textit{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).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{block_editing.png}
\caption{Block editing: free text intermixed with context values.}
\end{figure}

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 7 shows the result of organizing the root block content. Figure 7 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.

**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).

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 corresponds to a simpler interactive use of the tool, in the spirit of QBE, where the result consists of a hierarchical document. If one adds a child block of the root block, following the only available path **Cast**, the system would propose a representative node with actor's name (**Sidney Pollock**) and character (**Jack**) from the casting of **Husbands and Wives**. Here is the query produced from the canonical document obtained at the end of our simple session.

```plaintext
@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:<br/>
  <ul>
  @Cast{
    <li> @artist.first_name{} @artist.last_name{}
      as <b>@character{}</b></li>
  }
  </ul>
}
```

**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 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.
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.

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 extensions 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.

5 Related work

Using graphical interfaces for expressing queries is an old concern. The early language Query By Example (QBE) [12] and its variants such as Paradox or Microsoft Access [5] address the main principle of such visual tools: the query expression is based on an image of the result. They 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 [9], QURSED[10], Xing [6], and XQBE [2]. All these tools help users to construct complex queries over directed labelled trees. Queries are displayed with a graph-based representation. 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.

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.
6 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.

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. We also plan 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.

References

A Construction of a canonical instance

Algorithm CONSTRUCT builds a canonical instance over a schema $S$. It takes account of an expansion factor $k$ which determines the minimal size of a cycle in the instance. 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 $nodes_r[1], nodes_r[2], \ldots$. 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.

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.

\begin{verbatim}
CONSTRUCT (N, e, path)
Input: N ∈ V, 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 ∉ 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) ≥ 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) then // Stop here: no recursive call needed
    G_i+ = N ⇝ nodes_r[i_r] ; G_i+ = nodes_r[i_r] ←1 N
  else
    // Instantiate a new node nodes_r[i_r], and create the
    // corresponding edge
    nodes_r[i_r] := new(r);
    G_i+ = N ⇝ nodes_r[i_r] ; G_i+ = nodes_r[i_r] ←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) ≠ N
    CONSTRUCT(nodes_r[i_r], e, path)
end for
end if
end
\end{verbatim}

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.