A Model and a Language for Large Textual Databases

Marek Maurizio and Renzo Orsini

Department of Computer Science, Ca' Foscari Venice University, via Torino 155, Venezia Mestre, Italy

Abstract. The markup approach to represent and store large corpora of annotated textual documents is criticized for several reasons: it poses problems in expressing non-hierarchical structures, it limits the annotations in type and complexity, it makes difficult the writing of complex textual analysis programs since it requires the use of generic query languages like XQuery which are not well suited to the special need of the domain. We present a model and a language, called Manuzio, developed to be at the base of a new generation of textual document management systems which overcome the previous shortcomings. The model is an object based one, specialized for the specific domain, and has abstraction mechanisms which present some similarities with those of the object oriented database models. The language has query facilities and allows the development of sophisticated textual analysis applications. A prototype for a system has been designed and applied to several test cases.

1 Introduction

Research in the field of humanities is often concerned with texts in the form of documents, literary works, transcriptions, dictionaries, and so on. Interpreting, analyzing, sharing insights and results on these documents is the core of scholarship and research in literature, history, philosophy, and other, similar, fields. Technology advancements are affecting profoundly the study and use of texts in the humanities area, and a growing number of projects and experiments have been presented in literature to explore the possibilities given by electronic representations of those texts.

In this paper we propose a formal model to represent textual information with a focus on the representation of literary texts. The model is based on the idea that texts can be seen both as a sequence of characters and as multiple hierarchies of objects called *textual objects*, similar to classical object-oriented programming languages objects, but more focused on the specific domain. In our approach collections of documents with the same structure are organized in persistent textual repositories, databases of textual objects. To interact with textual repositories we have developed a novel programming language, Manuzio, with persistent capabilities and specific constructs to query, annotate, and manipulate textual objects and their annotations. From the language point of view the persistency of textual objects is transparent, so that they are treated just like any other of the language's values, minimizing the paradigm mismatch usually experienced by programmers in need to access external data.

The rest of the paper is organized as follows: in Section 2 a survey of related works is given; the main drawbacks of the existing proposals are put in evidence along with solutions provided by our approach. In Section 3 the foundations of the data model are presented. In Section 4 we present the major features of the Manuzio language along with some examples as a proof of concept. In Section 5 an overview of the whole Manuzio system architecture is given. Finally, in section 6 conclusions and hints on future work are given.

2 Motivations and Related works

Descriptive markup, realized mostly through `XML`, has become the de facto standard way of encoding literary texts in digital form. `XML` has, however, several important shortcomings: first, on its capabilities to express complex textual structures and annotations on them, second, in making difficult the expression of queries and complex operations, and, last but not least, on providing a solid background for building large libraries of multi-level annotated texts.

In particular, our proposal aims to solve some of the problems in the following areas:

- **Structure of text**: XML allows the representation of exactly one hierarchy, while in general multiple overlapping hierarchies are common in texts. Consider, for instance, a collection of classical lyrics, with two parallel hierarchies lyric > stanzas > verses > words, and lyric > sentences > words. This problem, known as the *overlapping problem*, has been largely discussed in literature. Different ways of marking concurrent hierarchies in pure XML have been proposed (see for a discussion [5]), but all these approaches are in fact workarounds. They complicate the document compromising human readability and often require special tools to be handled: standard query languages for `XML` are not natively compatible with such approaches. In our approach, instead, an object graph-based data model is used to allow a straight representation of such hierarchies.
- **Annotations**: the main limit of the markup approach when dealing with annotations on literary texts is the impractical representation of complex, structured annotations. In `XML` all attributes can be represented only as strings, which can be interpreted as different types. Moreover, only single elements can have attributes, so that it is difficult to annotate non-contiguous portions of text. Finally, annotations cannot overlap. In Manuzio, it is possible to declare annotations of any type, including structured values, or references to other portions of the text. The capability of referencing multiple textual objects as a single entity, or of having overlapping testual objects allows the user to express annotations also on arbitrary, even non-contiguous, parts of the text.
- **Query/Programming Language**: `XML` has a rich set of query languages, like *XSLT* or *XQuery*, to retrieve specific elements from a document and

manipulate them. These languages, however, perform poorly in terms of usability when applied to `XML` documents that represent multiple hierarchies. Solutions have been proposed [8, 6, 7], but the resulting languages, while efficient and usable, are not standard and, moreover, are still not focused on literary analysis applications. Our model has an associated programming language with native support for textual objects and capabilities to access persistent textual repositories, which makes easy to express complex queries as well as large textual analysis programs.
– **Integrated digital libraries**: XML is not well suited to represent collections of documents with well defined structures and annotations, since it has different kinds of type definitions and schema languages with complex syntax and semantics. A number of schema languages has been proposed [9], for instance the the `TEI` specification for literary texts, but schema adherence is not inherently mandatory for XML documents. Moreover, it is difficult to make multi-level/multi-user annotations to a set of documents, for the reasons above discussed. In Manuzio, on the other hand, documents must follow a schema (even if flexible for the schema defining mechanisms allowed, as we will see in the rest of the paper), and the nature of the textual objects (which can encompass any portion of text) and of their annotations (which are in fact attributes with every kind of value) are the base for building complex integrated digital libraries with robust multi-user access and modification.

Manuzio aims to draw a bridge between humanities researchers and programmers; when analyzing a text the humanities researchers often experience a cognitive distance between their work and the data they work with. When working with `XML`-encoded texts with multiple hierarchies, for instance, the difficulty in expressing queries of arbitrary complexity can hinder the research process. When using ad-hoc, graphical systems, instead, the researcher is limited to the answers the system is meant to give, and to expand such limits is, when possible, a difficult task. In our approach users can express common queries in a simple way and, at the same time, programmers can write both queries and programs with arbitrary complexity with a high-level specific language.

3 Model

In this section the main concepts behind the Manuzio data model will be presented. The main idea of the model is to consider the text in a dual way: as a finite sequence of characters as well as one or more hierarchies of textual objects. In the rest of the paper we refer to the text being modeled as the *full text*.

A *textual object* is an abstract representation of a portion of the full text together with structural and behavioral aspects. Textual objects have been inspired by the content objects expressed in [4], from concepts of concept-oriented data modeling [11], and, most of all, from objects of object-oriented programming languages. Usually a textual object has a logical meaning, like a paragraph, a chapter, a word, and so on. A *component relation* exists between textual objects: in Manuzio most textual objects are composed by other, "smaller" objects, which text is contained in their parent's text.

Definition 1. *A* textual object *is a software entity with a state and a behavior. The state defines the precise portion of the full text represented by the object, called the* underlying text*, and a set of* properties*, which are either* component *textual objects or* attributes *that can assume values of arbitrary complexity. The behavior is constituted by a collection of local procedures called* methods*, which define computed properties or perform operations on the object.*

Ordered sequences of textual objects of the same type are an entity of central importance in our model and are called *repeated textual objects*. Differently from a simple sequence, the elements of a repeated textual object cannot contain duplicates. The text represented by a repeated textual object can have gaps: it is not required for its elements to be contiguous. For instance we can consider the first three words of a sonnet as a repeated textual object, all the lines of a poem as another one, all the first lines of Shakespeare's roman plays as yet another, and so on.

Definition 2. *A* repeated textual object *is a sequence of textual objects of the same type, called its* elements*. Its underlying text is a composition of the underlying text of its elements. The order of its elements is induced by their natural order in the text.*

Each textual object is an instance of its textual object type. Repeated textual objects also have types, which can include other useful information about the collection of textual objects they represent. It is important to note that components, introduced in Definition 1, can reference both a textual object or a repeated textual object. A poem, for instance, can have a single textual object component to represent its title, as well as a repeated textual object to represent its words.

Definition 3. *A* textual object type *specifies the interface of a textual object. In particular, it specifies the name and type of its components and attributes. A* repeated textual object type *is a type which instances are repeated textual objects. Each repeated textual object type is an aggregation of textual objects of a same type, called its* elements*, and specifies the name and type of its attributes.*

Note that both textual object types and repeated textual object types can have attributes. This distinction is important because it offers a flexible way to annotate texts. A repeated textual object composed by the three words, for instance, can be annotated with a grammatical structure. Such structure will be tied to the three words as a whole, but not to any of them individually.

In Manuzio schemas, i.e. sets of type definitions, are designed so that, for any textual object type like, for instance, `Verse` there is one and only one *associated* repeated textual object type with elements of type `Verse` and its name is, by convention, `Verses`. This restriction does not hinder the expressiveness of the model and has significant impact on the elegance of the language's type system.

As in other traditional object-oriented models, in Manuzio a subtyping relation can be defined among textual object types through which we can model textual objects at different levels of detail. This feature adds to the model the ability to make incremental changes to textual objects.

The component relation is always a function from a single textual object to single or repeated textual objects. A `Poem`, for instance, could have a single textual object component of type `Title`, as well as a repeated textual object component of type `Parts`. The component relation can be considered as a graph where nodes are textual object types and arcs are relations.

Definition 4. *A* well-formed textual schema *is a set of textual object types which forms a bounded partial order set with respect to the component relation and for which: a) there exists a minimal, undecomposable type, (the* `Unit` *type); b) there exists a maximal type, called by convention* `Collection`*, which has a single instance,* `collection`*, the textual object whose underlying text is the full text and the components are the top-level components of the model.*

The specific definition of the *Unit* type is decided in the modeling phase, so that, for certain schemas, the unit type can be the word, for others the syllable, the character, and so on. For this reason the *granularity* of the model can vary from schema to schema. The *Collection* type is, instead, a maximal type which always has a single instance, called *collection* that is an ancestor of all the other textual object types and which underlying text is the whole full text.

To represent models we developed a simple graphical notation, shown in Figure 1. Single textual object types are represented as boxes split in two parts: the upper part contains the name of the type, while the lower one, if present, contains the name and the types of its attributes and methods. Components are represented by labeled arrows. A single arrow represent a component relation between two single textual objects, while double arrows represent a relation between a single textual object and the repeated textual object associated with the pointed type. The example Figure 1 shows the structure of a collection of italian poems. The top part of the schema is a single hierarchy of books > sections > poems. Each object of type `Poem` is composed by a `Title` and some `Parts`. `Strophes` are composed by both `Sentences` and `Lines`, two unrelated types that forms a parallel, overlapping hierarchy. Finally both sentences and lines are composed by a sequence of `Words`, which is the *Unit* type in this example. Each word has an associated method used to compute that word's stem.

4 Language

4.1 Overview

In this section we present the Manuzio programming language, whose main goals is to define textual models and write queries and programs over textual repositories. While the development of a new language is often considered an academic exercise, we felt that our choice to implement a completely novel language

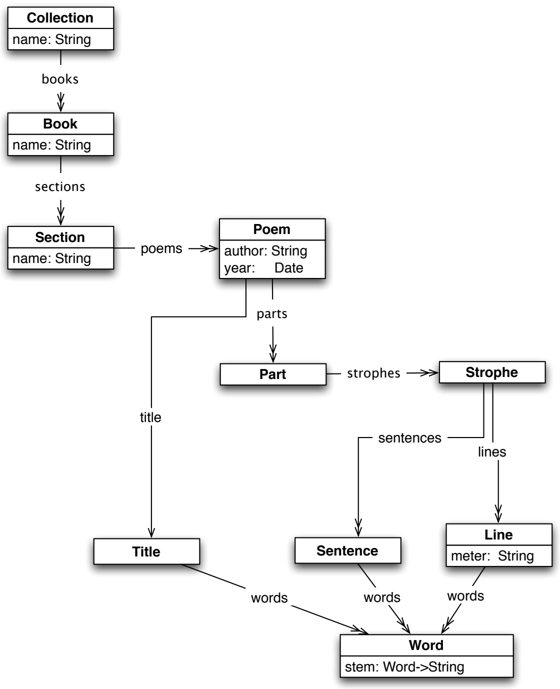

Fig. 1. A Manuzio schema for a collection of italian poems.

granted an unbiased, free environment to experiment new features. We are aware that the resulting language is still neither adequately polished nor particularly efficient. However, the results obtained by experimenting with this language can be very useful to design and develop future implementations.

The Manuzio language is a functional, type-safe language with object-oriented elements and persistent capabilities to interact with textual repositories. In the rest of the section we will show only the main features of the language, while a full description of its syntax and semantics can be found in [10]. The examples in the rest of the section show some typical use case of Manuzio and are based on the schema presented in Figure 1. Comments are started by the # symbol and are used to display the results of computations.

4.2 Textual Objects

In Manuzio both textual objects and repeated textual objects are equipped with a rich set of predefined operators. The most important ones are called the *access operators*, and are used to access their components and attributes. Such operators are called *get..of* and *getall..of*[1].

[1] The syntax of Manuzio makes large use of keywords instead of symbols.

Source Code 1 Component and attribute retrieval from textual objects and repeated textual objects.

```
let collection : Collection = usedatabase "montale";
let allbooks : Books = get books of collection;
get name of collection; #=> "Poesie di Montale":String
get name of allbooks;   #=> ['Ossi di Seppia',
                        #    'La Bufera e Altro', ...]:[String]
```

The `get` operator is used to access a component or an attribute. By specifying the name of a component, the textual object (or repeated textual object) associated to that component is returned. The operator can be applied in a seamless way both on single and repeated textual objects. When applied to repeated textual objects it retrieves the components of every the repetition's element and performs a flattening such results: the final result is a repeated textual object. This behavior has been chosen because it represents the most common way of accessing data in a literary text context.

In Source Code 1 the command *usedatabase* is used to connect to a textual repository. This command returns a value of type `Collection` from which all other textual objects can be reached. In this example all the books of the collection are retrieved by the `get` operator and the result, of type `Books`, is assigned to the identifier *allbooks*. The `get` operator retrieves also attributes in addition to components. In the example the name of the collection, a string, is retrieved. Also, the name of all books is also retrieved and the result is a sequence of strings.

The `getall` operator is used instead to retrieve a textual object's components recursively. By specifying a textual type T the `getall` operator traverses the textual hierarchy starting from the caller object t and fetch all the textual objects of type T that are, directly or indirectly, components of t. In Source Code 2, for instance, the words of a set of books are retrieved. The effect of this call is to descend the hierarchy of all components of the books recursively until the bottom of the scheme is reached, and collect all the different objects of type `Word` that are found. The resulting value is a repeated textual object of type `Words`. Note that, since there can be multiple paths connecting a type T with another type S, one component can be found multiple times. This is not an issue because all the elements of a repetitions are unique, so a set union of the results is performed.

Another important operator is `text..of`, that performs the mapping between textual objects and their underlying text. Such operator returns a string, and can be applied both to single textual objects and to repetitions. When applied to repetitions the operator returns a sequence of strings, each containing the text of one of the repetition's elements. An example of component access and text operators usage can be found in Source Code 2, where the words of the first three books are assigned to the *words* identifier. The underlying text of these words are then retrieved, uniqued and sorted.

Source Code 2 Retrieval of all words in the first three books and retrieval of the underlying text of each word.

```
let words = getall Word of books[1..3];
sort unique (text of words); #=> ["A", "ALBUM", "ALTA",
                             #    "ANGUILLA", ...]:[String]
```

Other important operators on textual objects take into account their natural ordering, and are used to compute distances or to construct repeated textual objects based on positional properties, like, for instance, all the words within a certain distance from a given word. There exist also a comparison operator (<=>), that can be used to examine the relations between object's positions [3]. They allow, for instance, to know if an object is fully contained in another one, if one partially precedes another, and so on. Table 1 summarize the syntax and semantics of the operators discussed so far.

4.3 Query Operators

Another family of Manuzio operators allow the writing of complex queries on textual repositories. Some of them are inspired by the Galileo language [2] and their syntax is similar to the one typical of object databases query languages. This family of operators allows users to interact with a textual repository directly from the Manuzio programming language without the need of embedding a query language with different syntax, types, and paradigm. Moreover, it is important to note that the same set of operators is able to work on sequences, repeated textual objects, and other kind of collections with a single simplified syntax.

In Source Code 3, for instance, the first query returns a repeated textual object of type `Poems` that contains all the poems which text is shorter then 300 characters. The title of such poems is then retrieved.

In Source Code 4, instead, we show how to construct a sequence of records each containing the title of a poem and the number of verses in that poem that contains the word stem "amor", only for poems where this word occur at least once. The first query selects all the verses of the collection where at least one word satisfies our requisite. The result is a repeated textual object of type `Verses`. Each verse is then grouped by the title of the poem it is contained

Operator	Semantics
get l of e	returns the component or attribute of e which label is l
getall T of e	recursively returns all the components of e of type T
text of e	returns the underlying text of e
e distance from e'	returns the distance between e and e'
e <=> e'	compare the relative position of e and e'

Table 1. The main operators on textual objects.

Source Code 3 Query operators examples.

```
let short_poems : Poems =
  select p from p in poems where size of text of p < 300;
text of (get title of short_poems)
  #=> ["LONGOMARE", "LASCIANDO UN DOVE", ...]:[String]
```

Source Code 4 Compute a data structure to find the most love-related poems.

```
let loveVerses : Verses =
select v
    from v in (getall Verse of collection)
    where some w in (getall Word of v)
    with (get stem of w) = "amor";

select {title = poemTitle, numberOfLoveVerses = (size of partition)}
from v in loveVerses
groupby {poemTitle = get title of (parent Poem of v)};
  #=> [{poemTitle = "L'ANGUILLA", numberOfLoveVerses = 1},
       {poemTitle = "INCANTESIMO", numberOfLoveVerses = 2},
       ...] : [{poemTitle:String, numberOfLoveVerses:Int}]
```

in, and such title is returned along with the number of verses in the relative partition.

5 Implementation

In the development of our first prototype the focus has been on the implementation of the language processor and persistent store. The schema definitions, the corpus parsing process, and the user interface aspects have been developed, instead, in a simpler way in order to have a working prototype. The results, while not ready for a production environment, let us evaluate critical language mechanisms early in its development and will influence future implementations.

5.1 Interpreter

The Manuzio interpreter is a highly modularized, easily extensible interpreter to be used in the process of language development and evaluation rather then as a fast and optimized software. The current version of the interpreter implements all the major features of a functional programming language with a particular focus on strings, textual objects, and their operators. The interpretation of textual schema declarations, however, is not, at the time of writing, fully implemented.

5.2 Persistency Model

The current implementation of the text store has been realized through a relational database in order to achieve a good tradeoff between performances and quick development of the prototype. The language persistent layer encapsulates the implementation so that the user is not aware of the underlying storage system.

Textual objects are first-class values of the language and also stored in the textual repository. Each textual object is univocally identified by the pair (T, s), where T is the object type and s it's position in the fulltext. For this reason, two textual objects with the same text but in different position are considered different both at language and database level. The database contains also information about the textual objects types. This has two important consequences: first, the database schema is generic and does not need to be changed to represent different textual models, and second, the database contains all the information needed to reconstruct its whole textual model. In this way the Manuzio language users do not need to declare textual object types before using them. Such types are automatically parsed from the database and loaded into the environment by the *usedatabase* command.

5.3 Data loading

The problem of loading initial data into textual repositories arise from the fact that textual collections already in digital form often lack a standard encoding. Such lack of a standard both motivate the proposal of alternatives like Manuzio and makes hard to produce general import algorithms. Currently in Manuzio, data loading has been carried out through ad-hoc parsers from both `XML` and plain-text input documents. A complete parser for the Unified Scripture Format `XML` (`USFX`)[1] format is being developed.

5.4 Graphical User Interfaces

We have currently two different, web-based, graphical interfaces. The first one allows the production of simple queries patterned after the structure of the textual schema. The user does not need to know the language, and at the same time can produce queries most sophisticated that the traditional “keywords style” interfaces. The tools synthesize a Manuzio query, then call the interpreter to evaluate it. The result is then shown to the user as text.

A second interface has been defined to browse directly the textual repository managed by the relational DBMS. The user can navigate a schema hierarchy and perform simple filter operations.

5.5 Overall System Architecture

A sketch of the whole system and of its intended use is shown in Figure 2. A domain expert and a programmer cooperate to instantiate textual repositories: the domain expert analyzes the input text, and defines the textual object types to be used in the new textual repository. The programmer then writes a set of recognizer functions that, given the input text and the textual object types defined in the textual schema, identifies the instances of these types in the text and fills the repository.

In the user section of the schema, instead, different kind of users interact with the textual repository through either a graphical user interface or directly by writing Manuzio programs. Textual analysis programs written in Manuzio use a special command to connect to the textual repository and retrieve the contained textual object types directly from it. The data in the repository can then be queried by multiple users, and their results can be annotated and shared.

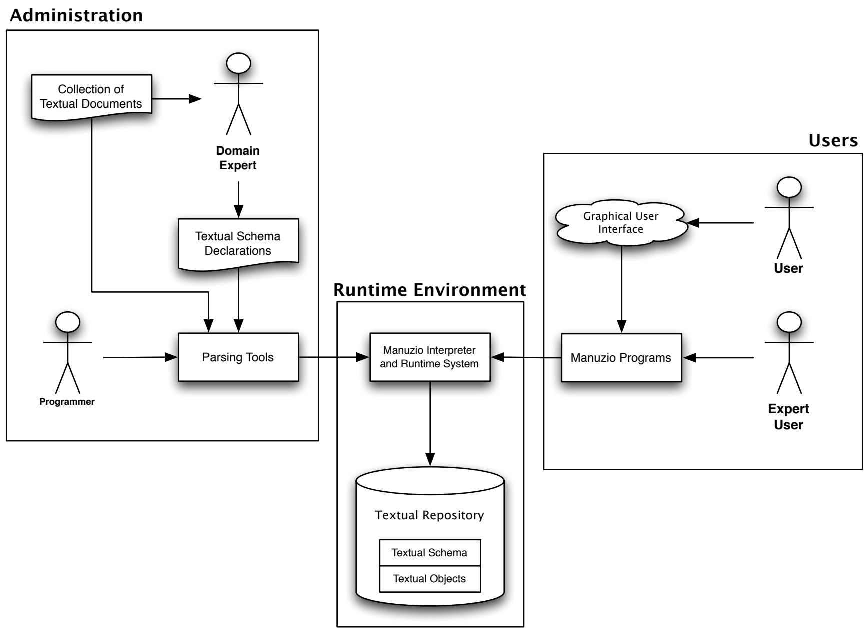

Fig. 2. The system functional architecture.

6 Conclusions and future work

In this paper we presented the Manuzio model and its associated language. Our model has some similarities with the classical object-oriented models in the field of programming languages. In fact, if we consider textual objects as composed only by attributes and methods, we can view them just as another kind of objects. The novelty of our approach is given by the domain-specific composition mechanism which organize our textual objects in a directed acyclic graph. The nature of the domain allows the construction of a bounded partial order set of textual object types so that interesting, powerful, and easy to use operators can be introduced in the associated language. An important aspect of the Manuzio model is the uniform way of dealing with both single as well as repeated textual objects through a set of powerful, homogeneous operators.

The implemented prototype has been used to perform different kinds of applied analysis on three textual repositories written in different languages: a collection of selected latin epic poems, the entire assortment of Shakespeare's plays, and all the italian poems of Eugenio Montale. Common queries on our repositories can be executed in acceptable times. We are aware that a great deal of work on optimization must yet be done to provide satisfying performances for larger collections of texts. However, we think that work on modeling and linguistics aspects of retrieval of texts and computations over them is very important, and prerequisite to enrich the solutions offered by research areas such as information retrieval and digital libraries. In particular, we believe that our language allows the user to take into account structural and semantic information in queries and programs, and this could easily improve the quality of the work in such areas.

An alternative implementation of the language as a library for a well-known object-oriented language is also under development to test the feasibility of such approach. Other side projects, in various stages of development, include an authoring system for "Manuzio texts" and a web-based visualization tool for textual repositories. Moreover, other system modules are under implementation. One will allow the direct authoring of textual objects, with a novel, touch-based, graphical interface, while another one will allow the users to annotate the result of the queries cooperatively.

Finally, we note that the implementation is easily interoperable with the majority of existing, `XML`-based, textual analysis tools. We have implemented, for instance, a set of simple algorithms to import `XML`-encoded data into Manuzio text repositories and to export views of a them, or query results, in `XML`.

References

1. Unified scripture format xml (usfx) url: http://ebible.org/usfx/.
2. A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.
3. J.F. Allen. Time and time again: The many ways to represent time. *International Journal of Intelligent Systems*, 6(4):341–355, 1991.
4. K.B. Bruce. *Foundations of object-oriented languages: types and semantics*. The MIT Press, 2002.
5. Steven J. DeRose. Markup overlap: A review and a horse. In *Extreme Markup Languages*, 2004.
6. I.E. Iacob and A. Dekhtyar. Processing xml documents with overlapping hierarchies. In *Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, page 409. ACM, 2005.
7. I.E. Iacob and A. Dekhtyar. Towards a query language for multihierarchical xml: Revisiting xpath. In *Proceedings of the 8th International Workshop on the Web and Databases (WebDB 2005), Baltimore, Maryland, USA*. Citeseer, 2005.
8. I.E. Iacob, A. Dekhtyar, and W. Zhao. Xpath extension for querying concurrent xml markup. Technical report, Citeseer, 2004.

9. D. Lee and W.W. Chu. Comparative analysis of six xml schema languages. *ACM Sigmod Record*, 29(3):76–87, 2000.
10. Marek Maurizio. *Manuzio: an Object Language for Annotated Text Collections*. PhD thesis, Dipartimento di Informatica, Università Ca' Foscari di Venezia, 2009.
11. A. Savinov. Concept-oriented model. *Encyclopedia of Database Technologies and Applications*, 2005.