A Model and a Language for Representing and Manipulating Annotated Text Collections

Marek Maurizio and Renzo Orsini

Dipartimento di Informatica,
Università Ca' Foscari di Venezia,
Via Torino 155, Venezia Mestre, Italy,
{marek,orsini}@dsi.unive.it

Abstract. Traditionally, collections of texts are digitally represented as a set of documents containing the text along with some kind of markup to define extra information, like metadata, annotations, etc. We propose a different approach that models the textual information in a dual way: as a formatted sequence of characters, as well as a composition of a particular kind of objects, called *textual objects*. With them, it is possible to represent different structures over the same text, together with complex annotations. Manuzio is a statically typechecked language to define a schema of such textual objects, and to write complex queries and applications on them with a set of powerful operators. In this paper we introduce the foundation of our textual model, the main features of the language, as well as a sketch of a system to manage persistent collections of texts and execute Manuzio programs.

Key words: object-oriented language, component, text-analysis, textual object, Manuzio language

1 Introduction

Commonly, text which must be automatically processed is represented through the use of some kind of markup language, which intersperses the base text with other, distinguished, text carrying some information, like metadata, formatting instructions, declaration of textual structures, etc. This approach has been widely diffused also by the availability of standards like XML, which made possible the definition of specific formats for many kinds of text, from literary texts (TEI [**?**]) to web pages (XHTML).

Many are the advantages of this approach: the marked text is easily created or modified with a text editor and at the same time can be efficiently processed; text can be exchanged among different systems and programs and in general used in a robust, interoperable way; many different kinds of information can be added to the text in a string-encoded format.

These advantages are, however, balanced by several, noteworthy, shortcomings, both on the power and expressiveness of the representation and on the way in which computation can be carried over that. A first severe limitation is that

marking can be applied only to contiguous non-overlapping segments of text, and that the text can be structured only in a strictly hierarchical fashion. Solutions exist to overcome in part these limitations, like the ones surveyed in [**?**], but they tend to be cumbersome, to produce complex unreadable texts, and to notably increase the complexity of programs dealing with such texts. We could summarize these critics by saying the traditional markup approach is not *scalable*: it is a simple and elegant solution for simple text annotations, but it is not adequate to deal with very complex situations, where annotations are made on different levels of the texts, belongs to different categories of meaning, are created by different authors, and so on.

Another important disadvantage of the markup approach is, in our opinion, the fact that programs for processing marked text are not easily written, requiring the mastering of complex query languages, not specialized for the particular domain, like those typical of XML (for instance, XPath, XQuery, XSLT). This problem becomes particularly serious when one has the objective of developing complex text analysis applications, like for instance those in the field of text mining, or applications which perform sophisticated syntactic or semantic analysis.

To overcome these disadvantages, we have developed a new data model which can be considered a domain specific object-oriented model, with many similarities to classical object-oriented models, both in the programming languages and in database area. This model has, however, a few key differences from the classical ones, and, in our opinion, some of its characteristics could be useful in domains different from that of texts in which it is now applied.

In this paper we will present the model, together with a concrete example of a programming language in which it is embodied, the Manuzio language. The language, which is still being designed, is intended to be used by a multi-user system to store persistently a digital collection of texts over which queries and programs are evaluated.

The objectives that we are trying to achieve through this approach are the following:

- to represent collections of texts with any kind of structure, including different overlapping structures for the same text;
- to represent any kind of annotations, even with complex information, on any part of the text, taking into account whatever text structure we are interested in;
- to provide a language in which to make queries, even sophisticated ones, on text and annotations, and build text processing applications;
- to build a system which would store in a persistent way one or more collection of texts, over which the programs written in the Manuzio language would be executed efficiently.

1.1 Related works

Solutions for text representation which are not markup-oriented have been already presented in the literature. For instance [**?**, **?**] present a model where text is seen as one or more hierarchies of objects that is the foundation of more complex systems like those presented in [**?**, **?**, **?**]. The approach that we propose presents a few similarities with those described in these papers, but it aims to provide a more complete solution. On one hand the Manuzio model is easily scalable, as the structure of each textual collection can be defined ad-hoc. On the other hand Manuzio provides a full programming and query language along with the model; such a language has been built to be expressive and easy to use in its specific domain of application. Finally, the Manuzio system is aimed to allow data to be stored in a persistent repository, to annotate it in a multi-user way, and to share results effortlessly.

The rest of the paper is organized as follows: in section **??** the foundations of the data model are presented. In section **??** we have a look at the major features of the Manuzio language and, finally, in section **??** an overview of the full system is given.

2 The Manuzio Model

We consider the textual information in a dual way: as a formatted sequence of characters, as well as a composition of logical structures called *textual objects*. These objects will be defined both in terms of the text which they represent (called the *underlying text*), as well as in terms of the other textual objects which are related to them. Textual objects participate to a *composition* relation, which specifies which objects are components of others. Textual objects can have also *attributes* and *methods*, and are classified through a set of types, called *textual object types*, among which a *specialization* relation is defined.

In this section we will present the model in a somewhat abstract way, through the help of a graphical notation, so that one can understand the main concepts without being distracted by syntactic details; in the next section, we will show a concrete syntax for the defined concepts.

2.1 Text

To put a concrete base to our model, we assume the existence of some text which must be represented. This can varies from a single textual document, to a collection of literary works of one or more authors, to a complete digital textual library. To simplify the discussion, and without loss of generality, we can assume that we intend to model a single (possibly very large) text, represented by a sequence of Unicode characters, in a format which is chosen by the user.

Definition 1. *The* full text *is a sequence of Unicode characters that represents all the text described by a specific Manuzio* textual model.

2.2 Textual Objects

A textual object is a computer representation of a text portion of the full text together with its structural and behavioral aspects.

Definition 2. *A* textual object *is a software entity with an identity, a state and a behavior. The identity defines the precise portion of the full text represented by the object, the* underlying text*. The state is constituted by 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, eventually with parameters, called* methods*, which define computed properties or perform operations on the object.*

Fig. **??** shows the structural aspects of a small set of textual objects taken from the start of a Shakespeare's sonnet. Each box represents a textual object and encloses its underlying text. If a box A is contained in another box B, then the textual object corresponding to A is a component of the object corresponding to B. So, the first line contains the first nine words of the sonnet, the first sentence contains two words more, and so on. Attributes of objects are represented through balloon-like shapes: in the example, the first line is associated with an attribute that represents its meter, the word 'marriage' has an attribute which contains a comment about it; the entire sonnet has attributes author and work.

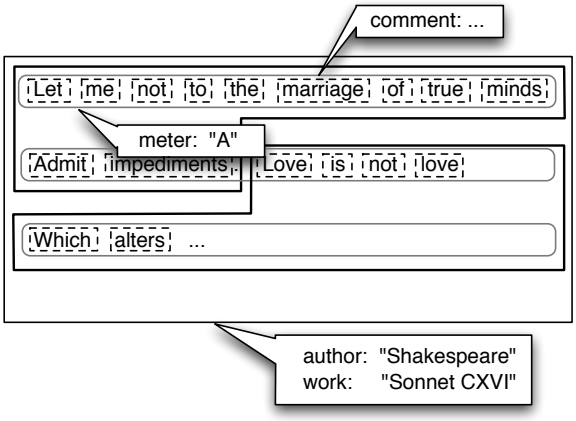

Fig. 1. Example of Textual Objects with Attributes

The containment, or component relation between two textual objects must satisfy the following property:

Definition 3. *A textual object TO_1 a* component *of a textual object TO_2 if and only if the underlying text of TO_1 is a subtext of the underlying text of TO_2.*

The *subtext* concept is not equivalent to that of *substring*, which is contiguous part of a string: a subtext can comprise non-contiguous parts of a text. This is another essential aspect of our model, and has an important consequence on textual objects: a textual object can consist of a repetition of components.

Definition 4. *A* repeated textual object *is either a special object, called the* empty textual object*, or it is a homogeneous sequence of textual objects, and its underlying text is the composition of the underlying text of its components.*

For instance, we can consider the first three words of the previous sonnet as a repeated textual object, all the lines of the sonnet as another, and so on.

In the next section we will show how to cope with the types of the different kind of objects.

2.3 Textual Object Types

To complete the above definition of textual objects, we must say that each textual object is in effect an instance of a *textual object type*, which represents a recognizable part of the text. For instance, in Fig. **??** we show, for the previous example, the types corresponding to the different kind of boxes: `Word`, `Line`, `Sentence`, `Sonnet`.[1]

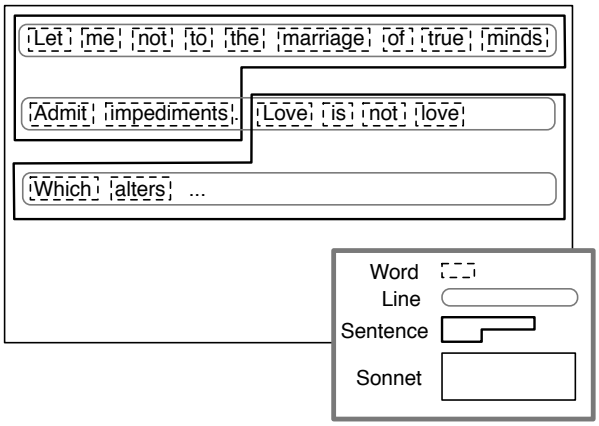

Fig. 2. Example of Textual Object Types

Each type has an associated interface that defines the properties and method signatures of the type's instances.

Definition 5. *A* textual object type interface *specifies the type name, the names and types of the properties, as well as the names and the parameters of the*

[1] By convention, a type name is capitalized.

methods together with their types. The type of a component is a textual object type, while the type of an attribute is a data type, like integer, string, boolean, a record type, etc. The parameters and result types of a method can be either textual object types or any other type.

The type of a repeated textual object is derived from that of the elements of the repetition:

Definition 6. *If a textual object type Ts is the* plural form *of a textual object type T with a given interface, then its instances are repeated textual objects which have as immediate and unique components objects of type T (called also its* elements*). The type T is called the* singular form *of Ts.*

For example, a `Poem` can have a component with name `title` and type `Sentence`, as well as a component with name `lines` and type `Lines`, which is the plural form of the type `Line`. This means that the lines of a poem are a repeated textual object with elements of type `Line`.

While type equality is by name, it is important to note that the type is used in defining the equality between two textual objects: two objects are equal if and only if they have the same type and the same underlying text.

2.4 The Component Relation

The component relation among textual objects is naturally extended to their types:

Definition 7. *A type T_1 is a* direct component *of a type T_2 if there is a component in T_2 which is of type T_1 or of its plural form T_1s. A type T_1 is a* component *of a type T_3 if it is a direct component of T_3 or if it is a component of some of its direct components.*

We introduce a graphical representation for a set of object type interfaces which evidences such relation. Each interface is represented by a rectangle 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. The components, on the other hand, are represented by arcs which connect a type interface with the interfaces of its component types. An arc is labelled with the name of the component, and is a single-pointed arrow or double-pointed arrow, depending on the singular or plural form of the component type. For instance, in Fig. **??**, a very simple model about poems which arises from the previous examples is presented.

The component relation is transitive and antisymmetric. Through this relation, we can now give the formal definition of a *well-formed textual model*:

Definition 8. *A* well-formed textual model *of a certain full text 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*

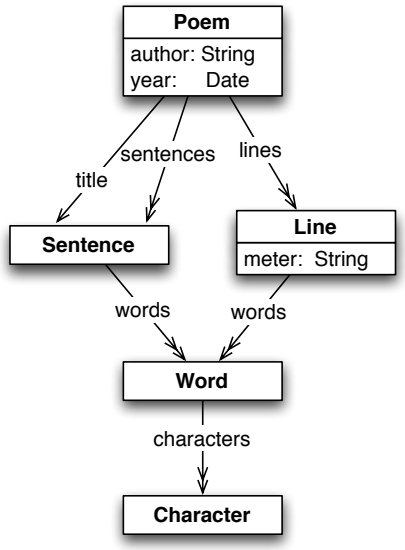

Fig. 3. A simple model about poems

type, (the `Unit` *type); b) there exists a maximal type, called by convention* `Total`*, which has a single instance,* `total`*, the textual object whose underlying text is the full text and the components are the top-level components of the model.*

For instance, in Fig. **??** the `Unit` type is `Character`,[2] while the `Total` type, not shown in the figure, has only a component `poems` of type `Poems`.

2.5 Subtyping and Inheritance

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. For instance, if a type `Work` has components `title` and `sentences`, and attributes `author` and `year`, we could define the type `Poem` as specialization of `Work`. `Poem`, in addition to *inheriting* properties and methods from `Work`, could have a new component, `lines`, and a new attribute, `meter`.

Definition 9. *A type A is* subtype *of a type B if it is defined as such; in this case A inherits all the properties and the behavior of B. A can also have new properties and methods, and can redefine the type of its components with a more specialized object type.*

[2] Different textual models can have different unit types (e.g. lemmas, syllables, written phonemes) depending on the granularity of the model in which the user is interested.

The presence of the subtyping (or specialization) relation between two textual object types A (the *subtype*) and B (the *supertype*) has the effect that every instance of the subtype is also an instance of the supertype. For example, every poem can be treated both as a generic work (for instance by asking for its author), and as an object with a component `lines` (for instance to count them).

We extend our graphical notation for this new concept. A subtype is graphically connected to its supertype through an arrow with a hollow arrowhead, and shows only the new information (with respect to its supertype). For this reason, the lower part of the rectangle contains only the new attributes, while only the arrows representing the new components are drawn. In Fig. **??** an enrichment of the previous example with subtypes is shown.

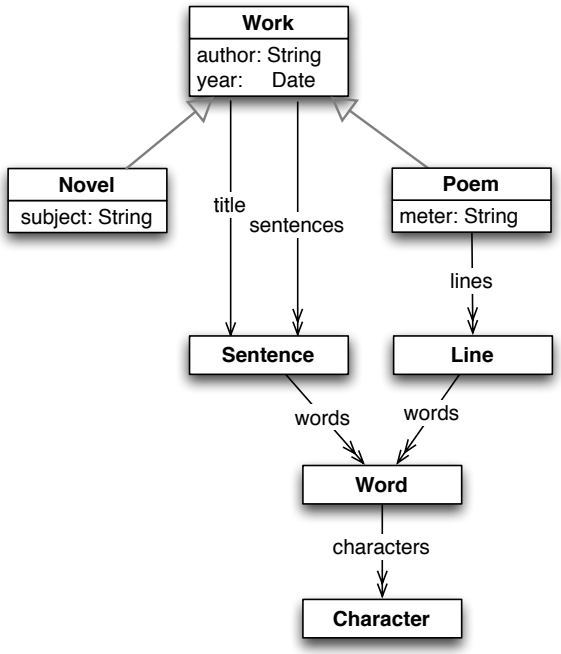

Fig. 4. A model about poems and novels.

In the figure, both `Novel` and `Poem` are subtypes of `Work` so that they inherit the components `title` and `sentences`, as well as the attributes `year` and `author`. Moreover, the `Novel` type has the new attribute `subject`, while the `Poem` type has the attribute `meter` and a component `lines` which allows to model lines of a poem.

The subtype relation among textual objects types is independent from the component relation, a part from the fact that it applies to the same set of types. For completeness, every type which is not defined as subtype of another

is implicitly subtype of the abstract type `TObject`, which has no components and attributes, and has a set of methods to deal with the underlying text, some of which will be presented in the next section.

3 The Manuzio Language

In this section we present the Manuzio programming language that provides a full syntax to define textual models and operate on them. Of course other approaches could be followed to implement our model, like, for instance, by extending an existing object-oriented language with a suitable library. The choice of a full specialized language, instead, is motivated by our interest in exploring the model's characteristics in a setting not constrained by the peculiarities of an existing language.

Only the aspects of the language related to the Manuzio model are presented here. For what concerns its other aspects it is sufficient to say that it is a static type checked language, it has a rich set of predefined types and type constructors (like Integer, String, Boolean, Sequences, Records etc.), and, finally, it has persistence handling capabilities, since it is intended to be embedded in a system which manages persistent collections of texts (see section **??**). In effect, some of the Manuzio features have been influenced by object-oriented database languages, in particular the Galileo language [**?**]. A full description of the language will be available in the forthcoming manual.

3.1 Textual Object Types

A textual object type can be defined with the following syntax:

```
type TypeName (pl. PluralTypeName) is SuperTypeName
  has
    componentName : ComponentType
    ...
  attributes
    attributeName : AttributeType
    ...
  methods
    methodName : MethodType := meth (parameters:ParametersTypes) is
        methodBody
    ...
end
```

In Fig. **??** the type definitions of a textual model concerning simple theatrical plays is shown. Note that, if the plural type name is omitted then it is assumed to be the type name with a juxtaposed 's'. Moreover, if the super type name is omitted, `TObject` is assumed.

```
type Play
  has
    title : Sentence
    scenes: Scenes
  attribute
    author : String
    year   : Integer
end

type Scene
  has
    speeches : Speeches
  attribute
    directions : String
end

type Prologue is Scene end

type Epilogue is Scene
  has
    salutations : Speech
end

type Speech (pl. Speeches)
  has
    sentences: Sentences
    lines : Lines
  attribute
    speaker : String
end

type Line
  has words : Words
end

type Sentence
  has words : Words
end

type Word
  has
    characters : Characters
  methods
    part_of_speech : String := meth () is
        compute_part_of_speech(self, self.Sentence)
end
```

Fig. 5. Model definition example.

In the model described in Fig. ??, a `Play` is composed by a `title` and some `scenes`, and has as attributes the `author` and the publication `year`. The `Speech` type has two components, `lines` and `sentences`, which are independent ways of considering a speech. The types `Prologue` and `Epilogue` inherits their properties from the supertype `Scene`. While `Prologue` does not have any additional property, the `Epilogue` type adds a `salutations` component. The type `Word` has components of the basic predefined type `Character`, which represents a unicode character, as well as a method to compute its part of speech.

An important aspect of the language is that no operators exist to create textual objects instances. This is due to the fact that they are created during the full text analysis and stored persistently by the system. On the other hand, many language operator returns textual objects as the result of their evaluation.

3.2 Object Access

The usual dot notation is used both to select properties (components and attribute) and to call the methods of a textual object. For instance, if `p` is a play then:

```
p.title
```

returns the textual object of type `Sentence` which is the component `title` of the play `p`, while:

```
p.author
```

returns instead the value of the attribute `author`, which is a `String`.

The dot operator can also be applied to repeated textual objects, with a semantics similar to a mapping operation. For instance, if `s` is an object of type `Speeches`, then:

```
s.lines
```

returns an object of type `Lines`, which contains all the lines of all the elements of `s`. Instead, the selection of an attribute from a repetition, like:

```
s.speaker
```

returns a sequence of values of type `String`.

Note that the composition of this operator always performs an implicit flattening of its result. For instance:

```
s.lines.words
```

returns a textual object of type `Words`.

For the inheritance property of the subtyping mechanism, the dot operator for a certain type can be applied also the the instances of its subtypes. For instance, since `Epilogue` is a subtype of `Scene`, we can select the `speeches` of an epilogue `e`:

```
e.speeches
```

Another way to select components of textual objects is by the exploitation of the component relation among their types, through the operators `.|` and `|.`. For instance, since the type `Word` is a component of the type `Sentence`, if `s` is an instance of `Sentence` and `w` is an instance of `Word` then:

```
s .| Word
```

returns the textual object composed by all the words which are components of `s`, while:

```
w |. Sentence
```

returns the (single) object of type `Sentence` which has `w` as component.[3]

These two operators are not limited to direct components. For instance, if `p` is a `Play`:

```
p .| Word
```

returns the textual object composed by all the words which are components of `p`, while:

```
w |. Play
```

returns the object of type `Play` which has `w` as component.

3.3 Repeated Objects Operators

The language offers a set of operators specific to repeated textual objects which allow to work on their elements. These operators are patterned on the classical sequences operators, although their result is always a textual object.

A first group of operators can be used to get a part of a repetition by specifying the elements in which we are interested by using integers and ranges as indexes. In the following examples, `p` is an object of type `Play`.

```
p.lines.index(1)
p.words.slice(1..3)
```

The first expression returns an object of type `Line`, and the second of type `Words`.

It is also possible to count the elements of a repetition, like in:

```
p.lines.size
```

[3] Note that the component relation definition implies that this object is always unique.

The language defines for repeated textual objects other operations typical of sequences, like concatenation, test for a condition holding on some or all the elements, test for inclusion of an element, etc., which are not described here since their are typical for data structures present in other languages (see for instance [**?**]).

We will focus here on the `select` operator which is the most important to query textual objects. As suggested by the name, it has a syntax similar to SQL selection, and can be used to retrieve objects through conditional expressions. Here, it will be described through examples, in which we assume `c` a textual object of type `Plays`.

The first example shows the simple form `select E1 from id in E2`, where `E2` is an expression returning a repeated textual object whose elements are bound, in order, to the identifier `id`, used in the evaluation of the expression `E1`. The result is the collection of such values.

```
select p.title
from p in c
```

This example returns a textual object of type `Sentences` constituted by the titles of all the plays in `c`.

A select expression can have a `where` clause, which can be used to give a condition to filter the elements over which the construct iterates. For instance, the following example returns only the titles of Shakespeare's plays.

```
select p.title
from p in c
where p.author = "Shakespeare"
```

Finally, we show how to use this construct to build complex queries. For instance, to find all the lines of Shakespeare's plays with exactly five words, we could write:

```
select l
from l in ( select p.lines
            from p in c
            where p.author = "Shakespeare" )
where l.words.size = 5
```

The internal `select` returns a textual object of type `Lines` and the external one iterates on its elements.

3.4 Other Operators on Textual Objects

The operators of this section are present in every textual object, since they are defined on `TObject`, which is the supertype of any other type. These operators can be used to retrieve and manipulate the underlying text of an object and to test various objects properties.

Given a textual object, we can obtain its underlying text with `text`. For instance, given a sentence `s`, the following expression:

```
s.words.text
```

returns a sequence of values of type `String` each containing the unicode characters of a word of `s`.[4]

Other operators exists to extract various information from the underlying text of an object. Note that, since the language has a complete set of operators on regular data types, once we get the text of a textual object we can apply to it all the operators available on strings (which includes, among others, pattern matching through regular expressions).

Comparison operators on textual objects can take into account the structure as well as the underlying text. Two textual objects can be tested for identity (i.e. if they are in effect the same object) with the identity operator '`==`' like in:

```
o1 == o2
```

When applied to textual objects, the simple equality operator is an abbreviation for testing the equality of their underlying text:

```
o1 = o2
```

which is equivalent to:

```
o1.text = o2.text
```

Also the other comparison operators, when applied to textual objects, takes into account their underlying text. For instance, to tell if an object starts before another, we can write:

```
o1 < o2
```

which is an abbreviation for:

```
o1.text_position < o2.text_position
```

that returns true when the object o1 precedes o2 in the full text (analogously for `<`, `>=`, `<=`).

In addition, the operator '`><`' returns `true` when an object overlaps another, so that, for instance, we can test if a sentence and a line overlap.

Finally, the language has other operators to test the relations between object's positions, known as the Allen's relations [**?**]. They allow, for instance, to know if an object is fully contained in another one, if one partially precedes another, and so on.

[4] Remember that, as seen in section **??**, an underlying text can comprise non-contiguous parts of the full text.

4 An overview of the Manuzio System

The Manuzio system is based on the presented language and has the capability to store in a persistent way complex annotated text collections and allow their manipulation by different users in a coherent and cooperative way. To reach such a goal the Manuzio language has other features, in addition to those shown in section **??**, to deal with users permissions, dynamic annotations, and management of the model and data persistency.

While the full system architecture is the subject of a forthcoming paper, here we will present an overview of its main functionalities.

1. The system should provide an efficient way of storing and querying very large quantities of textual material, together with annotations. To achieve this objective we are investigating different solutions, including those based on relational database technology.
2. To manage the textual model evolution, the Manuzio language has constructs to extend the model's schema with new types, to extend a type with new attributes and methods, to make persistent textual objects retrieved by a query, and to add and modify annotations on them.
3. The system should allow the access to concurrent users, through an appropriate set of permissions. For instance, different groups of users could work with different sets of annotations on the same textual objects.
4. A graphical, user friendly, interface is planned to perform assisted queries whose results are visualized with a choice of different graphical formats and mediums.
5. To exchange texts and annotations with other systems the XML standard format will be used through a set of tools which facilitates the mapping between it and the Manuzio internal format. In particular, XML is the privileged way of loading the data into the textual database, an operation which is done by a parsing process that can be automatic, semi-automatic or manual, depending on the complexity of the source data.

The system is currently under development, but the Manuzio model capabilities and a subset of the language's features have been tested using a simple prototype which has been used to perform a clause-related analysis of a medium-sized latin text corpus.

5 Conclusions and future work

This paper is an introduction to a novel approach for dealing with annotated collection of texts. We have presented the Manuzio model that, while specialized for this specific domain, has many similarities with other object-oriented models. In fact, if we consider only attributes and methods of the textual objects, we can view them just as another kind of objects. On the other hand, the novelty of our approach is given by the specific composition mechanism of objects which

connects their underlying text with their structure. The nature of the specific domain allows the construction of a bounded partial order set of their types and the introduction of interesting operators that take account of this structure. Another interesting aspect of the language is, in our opinion, the introduction of plural form types, along with their operators, which allow to treat repeated textual objects as the single ones, in a simple and uniform way.

While Manuzio is still a work in progress, our first experiments have shown the feasibility of the approach in dealing with collection of literary texts. We are now implementing an interpreter for the Manuzio language and developing a first prototype of the system. Moreover, since we think that this approach could be applied fruitfully in different domains, like, for instance, music and genetic sequences, we are currently investigating this possibility.

Acknowledgment

This work has been supported in part by grants of the italian Ministero dell'Istruzione, Università e Ricerca Scientifica, under the PRIN Project "Musisque deoque II. Un archivio digitale dinamico di poesia latina, dalle origini al Rinascimento italiano" about a digital archive of latin poetry. We are indebted to Paolo Mastandrea and Luigi Tessarolo, who have stimulated our interest in this subject and provided useful suggestions and challenging examples through discussions about their previous work.