МАТЕМАТИЧКИ ВЕСНИК
47 (1995), 61–84

UDC 519.765
оригинални научни рад
research paper

AN OBJECT MODEL FOR NATURAL LANGUAGE DOCUMENT

Gordana Pavlović-Lažetić, Goran Nenadić

Abstract. An object model of a text is defined, and structural and manipulative aspects of the model are presented. The model proposed supports text in its general sence (as a natural language document), which comprises structural, syntactic, semantic, orthographic, stylistic properties of both the underlying natural language and the text itself. Extended class algebra is defined as a basis for manipulative part of the model. Set of classes is then proved to be closed under the operations of the extended class algebra.

1. Introduction

Object programming represents, nowadays, a dominant approach to resolving problems programmers are faced with, since it often presents the most natural way of dealing with those problems. This is one of the reasons for database field to grow towards object-oriented systems.

Object data models are not standardized as opposed to other data models (relational, hierarchical and network ones). The core object data model [4] consists of objects, object identifiers, classes, attributes, inheritance and class hierarchy, encapsulation of objects and their behavior, message passing and methods implementing it. Different extensions to the core object model include complex objects (class composition hierarchy), version handling, dynamic schema changing etc.

Still, text is one of the most frequent *objects* of computer processing, and specifically of managing in databases, thus making it natural and strongly motivated to consider object methodology for text processing. The very first step is then to define an object model of natural language (Serbian) text.

In this paper, such a model is defined, and structural and manipulative aspects of the model are presented. As opposed to numerous programming systems that treat text simply as a string of characters, the model proposed supports text in its general sense (as a natural language document), which comprises structural, syntactic, semantic, orthographic, stylistic properties of both the underlying natural language, and the text itself. Class hierarchies defined are rich enough to take into account specific aspects of Serbian language.

Formally, structural part of the model relies on three types of constructors defining a structure of a natural language document. Manipulative aspect of the model incorporates tools for specifying, addressing and retrieving specific properties of a document, aiming at document creation, retrieval, deletion or updating. A well known formalism for addressing all the relevant properties of data in the context of the relational model is *relational algebra* [1]. It has been extended in different ways to suite the needs of more structured, or more semantic models such as nested relational models ([7], [6]), or different object models ([3], [2], [8]). An extension to the relational algebra for the object model of natural language document defined in the paper is considered (*extended class algebra*). A set of classes is proved to be closed under the operators of the extended class algebra.

2. Text and natural language document

Computer processing of textual data, from the very beginning, introduced a significant distinction between the notions of a text in general and a text in a computer senses. This distinction persisted till now. A text, in a computer sense, represents a string of characters, whose domain of interpretation may vary, depending on a system or user desires, independently of the text itself. Thus, a text is a computer dependent record of specific data, whose interpretation does not recognize semantics of the information that is to be encoded by such a record. A domain in which a text is to be interpreted (usually visualized) is completely independent of the text itself. For example, a text intended for recording an information in English could be easily presented on a screen in Cyrillic.

Natural language document possesses knowledge that (usually) is not stored with the document itself. First, it comprises specificities of a language the document was created in, document structure, specific orthographic and grammatical norms respected in writing, period of time the document was created in, style, and what is the most important, natural language (NL) document carries by itself a specific message, i.e. semantics that can be (uniquely or not) extracted from the text. Thus, as opposed to a computerized text which is linear (or one-dimensional), a NL document is multilayered i.e. multidimensional and can be viewed on different levels, each of which covers a specific component of such a document. Internal computer representation is just one of the components. So far, the solutions based on the procedural approach to programming, usually relied, in natural language document processing, on a document linearization, while a portion of information the text possessed was frequently lost. There is a strong impression that object methodology offers a possibility to naturally overcome multidimensionality of a NL document. It offers an approach to a text that enables not only an interpretation of the text, but a possibility to extract specific knowledge and semantics from a NL document stored in a computer.

In a computer stored NL document, there are three layers representing both syntactic (notational) and semantic structure of the text. This layered property determines a way of accessing and processing textual documents. It also induces a necessary structure of textual type objects (to be indicated later). The following three layers can be distinguished that need to be stored in a computer:

- **Internal text representation**, devoted to a computer only and representing, basically, a classic computerized text (a character string carrying specific information). This is the layer that almost all the conventional text processing is done upon, and it represents the lowest level of a text representation in a computer.
- **Logical organization**, devoted to a computer as well as to a human user, and reflecting semantic and logical structure of an NL document (chapters, sections, bibliography, unit titles, keywords, etc.)
- **Graphical presentation**, devoted to a human user solely, comprises possibilities of text visualization on a human readable media. Thus, it is not necessary to have this level of a text stored as a separate unit in a computer, but it is possible to keep just those elements of a graphical structure which provide, on a demand, for a graphical presentation of a text on a screen or a printer. A text as a whole, on this level, is a string consisting of *graphical* characters only — allograph codes and control characters of a device a text is to be presented on.

The differences mentioned between a "classic" computerized text and a NL document stored in a computer, demand for different software systems for their processing. In the first case those systems are *text editors* (word processors), while NL document processing is in charge of *NL document processors.*

3. The object model

Development of object-oriented databases was significantly influenced by the object methodology of programming. It also influenced organization and access to textual databases, consisting of integral texts stored in a computer, aiming at long duration, archiving, interchange, retrieval and intelligent processing (obtaining specific information and knowledge). While first three functions mentioned are also present in classic computerized texts, last two ones are tightly coupled with NL document and have to be adapted for a language the document is organized in. For example, retrieval of a textual database containing Serbian language texts has to provide for unifying flective forms of words, and it also has to overcome the two-alphabets problems present. It implies that an implementation of a retrieval method is dependent on a language a text is created in[1], but its interface (its "call") would be unique and independent of a natural language, which clearly suggests application of an object methodology.

Object model of a NL document, regardless of implementation, would have to reflect source language origin and a structure of a NL document. It also has to provide for representing the most part of relevant information the document possesses, in an abstract way and without obligations toward existing text preparation systems. The object model would be aimed at organizing complex structure

[1]Thus, an information about language the document was created in, has to be present (that is not the case with "classic" computerized text)

documents, featuring potentially long life and wide use. Basic demands on such a model are:

— abstraction and extendibility of a class of documents the model is able to describe;

— adaptation of the model to a specific natural language (Serbian language in this case), and reflection (in the model) of all the specificities of its use (for Serbian language — two alphabets, flection, rich morphology, dialects etc.);

— possibility of efficient and sophisticated NL document processing, which implies that the model has to possess and to reflect precise logical structure of a text, including word as the most important constitutional unit of a text. The model also has to provide for text mark up, as well as for logical structure granulation up to the level necessary for processing;

— possibility of reflecting and presenting standard NL document types and their structures, as well as the possibility of organizing and storing texts in accordance with such structures;

— independence of existing systems for graphical text preparation, still providing for conversion of texts formatted by some of those systems, into objects according to the model, and vice versa.

Complexity of an NL document, and the feature of it being layered, account for the presence of both class and class composition hierarchies in the object model of text:

- vertical hierarchy (type *IS-A*) comprises inheritance of attributes and methods between different sorts of documents (e.g., BOOK *IS-A* NL-DOCUMENT).
- horizontal (or composition) hierarchy (type *IS-PART-OF*), based on properties and portions a specific sort of documents standardly possesses (e.g., CHAPTER *IS-PART-OF* BOOK).

Vertical hierarchy represents interrelationship between different classes of documents. Existence of different classes of documents has to cover the most part of standard sorts of documents encountered in textual databases, and to provide for adding new classes. Inheritance of attributes (properties) between classes has to reflect structure, organization and mutual relationship between different classes of documents, while methods need to provide, in a standardized way (through unique interfaces, i.e., signature) for classic procedures of textual data processing, taking into account class of a document (function overloading, encapsulation). Inside different classes some of the functions are redefined or even completely dropped (e.g., generating index for objects of type BOOK is different from the same function for objects of type DICTIONARY, while objects of type ARTICLE do not need such a function at all).

- NL-DOCUMENT
    - BOOK
        - TEXTBOOK
        - ...
    - OFFICE-DOCUMENT
        - REPORT
        - LICENCE
        - STATEMENT
        - AGREEMENT
        - ...
    - ARTICLE
    - DOCUMENT-WITH-ENTRIES
        - DICTIONARY
        - ENCYCLOPEDIA
    - LETTER
        - PRIVATE
        - BUSINESS
    - CALL-FOR-PAPERS
    - ...

- DOCUMENT-PART
    - CHAPTER
    - SECTION
    - PARAGRAPH
        - TEXT-PARAGRAPH
            - ABSTRACT
            - ENTRY
            - ...
        - FIGURE-PARAGRAPH
            - FIGURE
                - BIT-MAP
                - POSTSCRIPT
                - ...
            - TABLE
            - ...
    - PARAGRAPH-PART
        - NAME
        - WORD
            - PERSONAL-NAME
        - DATE
            - ...
        - NUMERAL
            - ARABIC
            - ROMAN
        - OBJECT-REFERENCE
            - FIGURE-REFERENCE
            - WORD-REFERENCE
            - ...
        - HORIZONTAL-SPACE
        - TITLE
        - FOOTNOTE
        - VERTICAL-SPACE
        - TABLE-OF-CONTENTS
        - ...

Horizontal hierarchy has to provide for detailed representation of a document structure. This hierarchy represents another level of composing and again, it has to reflect specificities of different classes of documents, representing parts that specific classes consist of. Granulation of a text logical structure is defined explicitly by a horizontal hierarchy, up to a desired depth.

Document types hierarchy and vertical hierarchy of classes used in class composition hierarchy reflecting structure of a NL-document may be represented, by indicating just class names and hierarchical organization, in the following way (it may include other classes, too), see the table on the previous page.

Class DOCUMENT-PART represents the root of the second class hierarchy. Along with basic *IS-A* relation between classes presented (e.g., POSTSCRIPT *IS-A* FIGURE, FIGURE *IS-A* FIGURE-PARAGRAPH and FIGURE-PARAGRAPH *IS-A* PARAGRAPH, and, at the end, PARAGRAPH *IS-A* DOCUMENT-PART), *IS-PART-OF* relation exists between classes: WORD *IS-PART-OF* TEXT-PARAGRAPH, TEXT-PARAGRAPH *IS-PART-OF* SECTION, SECTION *IS-PART-OF* CHAPTER. Similarly, TEXT-PARAGRAPH consists of WORDs, NUMERALs and FOOTNOTEs, which, in turn, consist of WORDs, NUMERALs etc.

Object structure of specific classes may be very complex, voluminous and demanding. Some of the structures will be presented in the following section.

4. Examples of class structure in the object model of text

Before we present some possible structures of specific classes, we will sketch the notation we will use for constructors of types of specific attributes (formal description of these constructors will be presented, together with the formal description of the model, in section 5). Indicator of a specific type inside curled braces will denote a finite ordered set with a direct access (access to the k-th element) consisting of objects of that type. Specific realization of this constructor may be an array, a list or a similar structure. A pointer to an object of a given type (in a specific implementation it may be a unique identifier of the object in the database) will be denoted by an asterisk followed by a name of the corresponding type. For example, {WORD} will denote a finite ordered set of objects of the type WORD, and {* WORD} – a finite ordered set of pointers to objects of the type WORD.

4.1. Class NL-DOCUMENT. The class NL-DOCUMENT is an abstract class and it does not have instances, but it represents the basis of the overall hierarchy of NL documents. Inside it, declaration of methods and functions needed in text processing may be given, and a basic structure may be specified that each NL document has to possess. Thus, basic attributes defined in this class will include the following:

Author: {NAME}
Title: TITLE
Date: DATE
Abstract: ABSTRACT
Orthography: ORTHOGRAPHY-ID
Versions: {* NL-DOCUMENT}
Contents: {*DOCUMENT-PART}

The attribute *Author* consists of an (ordered) set of objects of type NAME, which can be either personal name (subclass of WORD) or name of an institution. Domain of the class TITLE is set of words, numerals etc., while the attribute *Versions* may point to the set of earlier versions of the same document. The attribute *Orthography* carries information on which orthography norms the NL document is organized. Specifically, this element may be of crucial help for our language, in checking correctness of text input, since different orthographies assume different rules for writing specific words. It can be also used as a source of transcription rules for foreign names. The attribute *Date* contains the date of creation of the document. The parts of class hierarchy which have roots in classes NL-DOCUMENT and DOCUMENT-PART are connected with *Contents* attribute. Each document has contents represented by objects of DOCUMENT-PART subclasses. For different kinds of documents consist of different parts (books consist of chapters, sections consist of paragraphs, etc.), it is necessary to redefine this attribute in NL-DOCUMENT subclasses, namely to specify the appropriate subclass of DOCUMENT-PART NL document consists of (see examples in 4.1.1 and 4.1.2).

In this class, foundations may be given to methods necessary for NL document processing. Among the others, there can be *insert-and-save, visualization, editing-and-changing, morphological-processing, retrieval* (by different attribute values), *sorting* (on different attributes), *concordance-generating, spelling-checking, consistency-checking, synonym-generating, style-checking, statistics*, and so on. Each of the methods can be (and usually is) redefined in subclasses in a way that makes it possible to save and use all the information relevant for the corresponding type of document. Specifically, function *insert-and-save* has to provide for recording not only contents but also the overall structure of a document, while *visualization*, has to produce, based on what has been recorded, the corresponding string of graphical characters on a screen or printer.

A method *conversion* may exist, too, that would convert a document organized in some of the text preparation systems (MS WORD, CHI, TEX, etc.) into the form implied by the model. This is especially important having in mind existence of significant textual holdings in different text and word processing systems formats.

4.1.1. Class Book. Class BOOK is a descendant of the class NL-DOCUMENT, so it has the following properties, in addition to the ones mentioned with the class NL-DOCUMENT:

Contents: {∗CHAPTER }
Preface: TEXT-PARAGRAPH
Epilogue: TEXT-PARAGRAPH
Appendix: TEXT-PARAGRAPH
Table-of-contents: TABLE-OF-CONTENTS
Bibliography: {∗ BIB-REF}
Index: {∗WORD}
Cataloging: {WORD}
Title-page: {PARAGRAPH-PART}

The domain of *Contents* is redefined in this class: a book consists of chapters. Class CHAPTER structure is presented in subsection 4.2.2, while TEXT-

PARAGRAPH objects represent sets including only objects belonging to PARAGRAPH-PART subclasses (word, numeral, date, etc.). Objects of class CONTENTS are sets of *Titles*, which match document part titles (in this case titles of chapters, for books consist of chapters). During the visualization, a page number of media on which document is presenting can be added to this attribute. In addition, functions *index-generating* and *table-of-contents-generating* may be defined, which may be called for only from the visualization method of this class, while the method *insert-and-save* has to be redefined as to enable creation of objects of the type BOOK.

Notice that objects of the type CHAPTER may be saved as standalone objects and thus may be used as constituent parts of many (different) books (shared composite reference [4]). Similar holds for bibliography, where the same objects may be used not only as parts of objects of the type BOOK, but also as parts of all the other objects referring to some documents.

4.1.2. Class ARTICLE. Class ARTICLE is a descendant of the class NL-DOCUMENT, too, but it may include just the following additional attributes:

Contents: {* SECTION}
Acknowledgment: TEXT-PARAGRAPH
Bibliography: {*BIB-REF}
Key-words: {WORD}

The set of functions may be extended by a function for table-of-contents generation, while some of the inherited parent functions would have to be redefined. By specifying values for *Key-words* attribute (by the author or automatically — by performing a method) we have possibility of describing document contents. Thus, we allow to retrieve documents from database (articles, chapters, even paragraphs) which cover matter specified as condition in a user query against textual database.

4.2. Class DOCUMENT-PART. The class DOCUMENT-PART represents an (abstract) root class for the horizontal (composition) hierarchy. Except that it represents the root class of the hierarchical, logical structure of a document, its role is also to represent graphical information that are sufficient for presentation of document parts on an output device. Thus, this class is to pay for visualizing ability. Specific attributes of this class may be:

Font: FONT-ID
Letter-Size: INT
Code-page: CODE-ID
Position: POSITION
Language: LANGUAGE-ID.

Domains of classes FONT-ID, CODE-ID, LANGUAGE-ID have to provide for recognition and usage of different writings and code pages, as well as possibility of using different language's words in a document.

All the functions of this class predominantly support visualization and saving information important for it, so it is necessary to implement methods for intelligent writings conversion (*transliteration*), changing letter size etc., as well as visualization method, different statistics, etc. It is important to point out that all the document parts are stored in a canonical form (which may be even computer dependent). Only in case of visualization, these parts are presented in accordance with graphical features of a text, stored in the above attributes. All the processing is done over canonical form, and specificities of visualization are taken care of on the basis of the above data.

4.2.1. Class WORD. This class has to describe the basic constituent unit of an NL document. Taking into account all the complexity of a word in Serbian language (flective forms, morphographemic aspects), the best solution is to have an object of this class representing de facto a pointer to the corresponding entry in an electronic dictionary [9], which directly provides for all the knowledge stored about the word (i.e., about the word form appeared in the document). Information about basic form and forms of a word that can appear are present in e-dictionary (e.g., for nouns: kind – proper, common etc., properties – gender, number, case, ... ; for verbs: tense, form, mood, number, ...). This way all the problems may be solved concerning spelling checking, determining basic (dictionary) form of a form occurred (e.g., for index generation), flection problems, consistency control etc. If the e-dictionary contains places where "problematic" words can be hyphenated at visualization then the problem of hyphenation would be trivial. Along with this basic data (of pointer type), a dialect of a word may be also stored as an attribute (e.g. *Dialect: (ekav, ijekav, ikav)*). This attribute indicates the dialect in which the word is to be visualized, while keeping it stored in a form and dialect found in the e-dictionary.

The corresponding set of methods may be easily recognized (from the foregoing): *insert-and-save, dictionary-entry, spelling-checking, forms-by-dialects, hyphenation, flective-forms*, etc. Each of these is called from a method implementing the equivalent function in the context of superordinated classes in a document and is done by consulting the e-dictionary.

Note that, besides the information mentioned, a punctuation following a word may be stored, too, which, otherwise, could not be a separate object in a document. "Following" punctuation is a space, in case there is no punctuation following the word. By this approach, it is easy to extract a *sentence* as a part of a logical structure of a text. This way a notion of "sentence beginning" could be formalized (for the purpose of capitalizing the beginning letter in visualization): a new sentence begins at the beginning of a section, paragraph, etc. or "behind" a word having a nontrivial "following" punctuation. Thus, the first word in a sentence does not have to be stored, internally, with a capital first letter (which is convenient for retrieving, indexing, and similar), except when it is always written that way (e.g., proper names).

4.2.2. Class CHAPTER. As a subclass of the basic class DOCUMENT-PART, this class, together with information necessary for visualization, may contain the following attributes:

Title: TITLE
Numaration: INT
Contents: {* SECTION}
Key-words: {WORD}

There is no need for specific methods (except for redefinition of existing insertion and visualization methods present in the class DOCUMENT-PART). Attribute *Key-words* is of importance for retrieval of such chapters that deal with specific matter specified in a user query.

4.3. The object model graph. It is pointed out that two relationships exist between classes: *IS-A* and *IS-PART-OF*. On the basis of the former it is possible to define *class inheritance graph*, having class names as vertices and unlabeled edges representing *IS-A* hierarchy. The latter, along with structure of defined classes, defines *class composition graph.* Each edge in this graph is labeled by the attribute name that indicates *IS-PART-OF* relation, along with constructor that should be used (curled braces are used to denote set-constructor, and an asterisk for pointer to an object). For example, as class BOOK possesses attribute *Contents* which is set of pointers to CHAPTER-objects, there exists an edge from BOOK vertex to CHAPTER vertex, labeled by $\{*\ \mathit{Contents}\}$, as indicated by a figure:

$$\boxed{\text{BOOK}} \xrightarrow{\{*\ \text{Contents}\}} \boxed{\text{CHAPTER}}$$

There could be several edges connecting two vertices (e.g. — BOOK and TEXT-PARAGRAPH are connected with edges labeled as *Preface, Epilogue, Appendix*).

Object model graph is a superposition of the two graphs. It contains vertices with names of the classes defined in the model, and two kinds of edges connecting them: one inherited from *class inheritance graph* (unlabeled edges) and second from *class composition graph* (labeled as explained). Formal description of object model graph as an interpretation of object model will be given in the next section.

The figure on the next page represents a part of object model graph (neither all the classes, nor all the relationships between them are included).

Labels on edges that have to be labeled are replaced (only for purpose of representing in this figure) with numbers. The correspondences are as folows:

$1 = \{*\ \mathit{Contents}\}$	$5 = \{*\ \mathit{Index}\}$
$2 = \mathit{Title}$	$6 = \{\mathit{Key\text{-}words}\}$
$3 = \{\mathit{Author}\}$	$7 = \mathit{Title\text{-}page}$
$4 = \mathit{Abstract}$	$8 = \mathit{Cataloging}$

5. Formal description of the object model

In previous sections, notion of a class was used informally, and in the same sense as a notion of a type. Further, methods indicated with each class were also informally specified.

In this section, a formal description of the object model of text will be presented, together with extended class algebra as a basis for manipulative methods of creation, retrieval, deletion and changing documents.

5.1. Types and classes. *Type* in the object model of text is a triple (T, T_v, T_o), where T – is a *type identifier*, T_v – *value type*, set of values, and T_o – *operation type*, set of operations on values from T_v. The type (T, T_v, T_o) will be also written as $T(T_v, T_o)$.

Type in the object model of text is defined by the following recursive definition:

DEFINITION 1. (TYPE) **1.** *Primitive type* is a type whose value type T_v is a primitive pascal type, i.e., $T_v \in$ {integer, real, char, boolean, scalar, subrange}. T_o is a set of defined arithmetic, character, boolean and comparison operations, which includes usual operations on specific types.

2. Primitive type is a *type*;

3. If $T(T_v, T_o), T_1(T_{v_1}, T_{o_1}), \ldots, T_n(T_{v_n}, T_{o_n})$ are types, then

a) $STT(STT_v, STT_o)$ is a *structured tuple type*, over the types $T_1(T_{v_1}, T_{o_1}), \ldots, T_n(T_{v_n}, T_{o_n})$,

b) $SST(SST_v, SST_o)$ is a *structured set type* over the type T,

c) $SPT(SPT_v, SPT_o)$ is a *structured pointer type* over the type T,

if

STT, SST, SPT are type identifiers; SST and SPT may be empty;

$STT_v = T_{v_1} \times \cdots \times T_{v_n}$ is a value type, i.e., a set of n-tuples of values from $T_{v_1}, \ldots, T_{v_n}$ (T_i may be, and usually is substituted for T_{v_i}, if not empty);

$SST_v = \{T_v\}$ is a value type, i.e., a set of finite, ordered sets of values of the type T_v;

$SPT_v = *T_v$ is a value type, i.e., a set of pointers to values of the type T_v;

STT_o, SST_o, SPT_o are operation types, which include specific operations on those structured types, i-th component (element, pointer, pointed object) selection operation, and a composition of a selection operation and operations from the corresponding operation type.

4. Structured type is a *type*;

5. Type can be constructed by application of items 2, 3. and 4. of this definition only. The following restrictions hold: if type T is obtained by item 3.c) of this definition, then the same item cannot be applied to it in order to obtain a new type; if type T is obtained by item 3.b) of this definition, then neither the same item nor the item 3.c) can be applied to it in order to obtain a new type.

EXAMPLE 1. For the type
$Word$({ e-dictionary words }, { part-of-speech, form, prefix, ending, length}),
according to the items 3.c), 3.b) of Definition 1, also a type is

$Index$($\{*Word\}$, { pointer to ith word in index, ith word in index,
operations on $Word$ applied to ith word in index,
create index, retrieve index}).

Informally, *class* is an abstract set of objects with common properties. Common properties of objects of a class are defined by a *class declaration*, and each specific set of objects with those properties is an *extension of the class*. The same term *class* will be used for both class declaration and class extension, whenever distinction between the two is not of importance, or meaning of the term is unambiguously clear.

DEFINITION 2. (CLASS) 1) *Class declaration* is a triple

$$(class\_id, attribute\_domain\_list, set\_of\_methods),$$

usually written as $class\_id(attribute\_domain\_list; set\_of\_methods)$. $Class\_id$ is an identifier of a class, $attribute\_domain\_list$ is a list of components separated by commas, each of which being of the form $attribute\_name : domain$, and methods are predefined and user defined operations over objects of the class ('messages' those objects respond to). Domain is structured out of classes in the following way:

(i) *Primitive class declaration* corresponds to a primitive type. Thus, if $T(T_v, T_o)$ is a primitive type, then $T(T : T_v; T_o)$ is a *primitive class declaration.* Set of methods of a primitive class is a set of operations of the corresponding primitive type. For example, $INT(INT : \{set\_of\_integers\}; \{+, -, *, \div, <, >, =, \leq, \geq\})$ may be a primitive class declaration.

(ii) Primitive class declaration is a *class declaration*; *class associated* with an only attribute of a primitive class, and with its corresponding domain, is the primitive class itself; *type* of a primitive class and of its only domain, is the corresponding primitive type.

(iii) If

$$C_1(a_1^1 : D_1^1, \ldots, a_{n_1}^1 : D_{n_1}^1; M_1),$$
$$C_2(a_1^2 : D_1^2, \ldots, a_{n_2}^2 : D_{n_2}^2; M_2),$$
$$\ldots,$$
$$C_k(a_1^k : D_1^k, \ldots, a_{n_k}^k : D_{n_k}^k; M_k)$$

are class declarations of classes $C_1, \ldots, C_k$, with corresponding types $T_1(T_{v_1}, T_{o_1})$, $\ldots$, $T_k(T_{v_k}, T_{o_k})$, respectively, then

$$C(a_1 : D_1, \ldots, a_k : D_k; M)$$

is a *class declaration*, where

(a) C is a class identifier

(b) $a_i, 1 \leq i \leq k$ are (different) attribute names and $D_i, 1 \leq i \leq k$ are the corresponding *domains* (of attributes a_i) and

$D_i \equiv C_i$ of type T_i, or

$D_i \equiv *C_i$ (*pointer to a class* C_i) of structured pointer type $*T_i$,

or

$D_i \equiv \{C_i\}$ (*set over a class* C_i) of structured set type $\{T_i\}$, or

$D_i \equiv \{*C_i\}$ (*set of pointers to a class* C_i) of structured set pointer type $\{*T_i\}$.

A *class associated* with an attribute a_i (and the domain D_i) is C_i. *Type* of the class C is a structured tuple type $T(T_{v_1} \times \cdots \times T_{v_k}, T_o)$.

(c) M is a set of methods, and it is a set of operations T_o of the type of the class.

2) An arbitrary set of values of the type of a class is a *class extension* (of that class); an element of a class extension is an *instance* or an *object* of the class extension (of the class).

3) If a set of attributes of a class S contains a set of attributes of a class S', then the class S' is a *superclass* of the class S, and the class S is a *subclass* of the class S'; if the class S' has attributes $a_1, \ldots, a_n$, defined on domains with associated classes $C_1, \ldots, C_n$, respectively, then each attribute a_i, in the subclass S, is defined either on a domain with the same associated class C_i as in the superclass (which means, on domain C, $\{C\}$, $*C$ or $\{*C\}$), or on a domain whose associated class is a subclass (direct or indirect) of the class C_i. If classes S and S' have the same sets of attributes, then superclass/subclass relationship has to be explicitly defined.

Each class has at least one superclass (assuming an empty-attribute class and multiple inheritance) and zero or more subclasses.

REMARK 1. a) Although a class declaration contains a list of methods, in addition to a list of attributes and their domains, in what follows, this notion will assume just *class_id* with *attribute_domain_list*. Methods will be considered separately.

b) In object model implementations, relationship between classes (superclasses, subclasses, disjunctive, overlapping classes) are explicitly cited with class declarations. If a class S' is a superclass of a class S, then only specific (or redefined) attributes of the class S are specified in its declaration. For example, in declaration of the class BOOK, attributes of its superclass NL-DOCUMENT are not mentioned (they are assumed), but only specific and redefined attributes of the class BOOK (e.g., *Preface, Contents*, etc.).

5.2. Methods and extended class algebra. A set of methods of a specific class may contain three types of methods:

I *User defined methods* – procedures that objects of a specific class respond to. They may have side effects. For example, such a method for a class BOOK may be *index-generating*.

II *Attribute methods* (component-element-object selection)

Let *comp* ('component') be an arbitrary component in a class structure. For example, in a structure of the class BOOK, *comp*s are BOOK.$Date$, BOOK.$Index$, BOOK.$Index^i$ (i-th component of attribute $Index$), BOOK.$*Index^i$, BOOK.$Author^i$, but BOOK.*Acknowledgement* is not a *comp* of the class BOOK, since *Acknowledgement* is an attribute of the class ARTICLE but is not an attribute of the class BOOK. Property of being *comp* of a class is defined in a similar way as for nested relational models ([6], [7]), although there are significant differences between nested relational and object models: our model consists of extended set of type and class constructors, and it supports generalization structure, in addition to aggregation one (presented by nested relational models).

Since the definition of a class is recursive, class structure (and type structure) may be of unlimited complexity (in case of cyclic definitions even infinite), so a class component definition is recursive, too.

DEFINITION 3. (*comp*) For the class declaration

$$class\_id(a_1 : D_1, \ldots, a_n : D_n)$$

(i) *class_id* is a *comp* with a domain *class_id* itself;

(ii) if C is a *comp* with a domain D, and D is an identifier of a class with a class declaration $D(a'_1 : D'_1, \ldots, a'_k : D'_k)$, then $C.a'_j (1 \leq j \leq k)$, is a *comp* with a domain D'_j;

(iii) if C is a *comp* of the form $(C \equiv) C'.c$ (c does not contain a dot), with a domain $\{D\}$ (D as in (ii)), then $C'.c^i$ is a *comp* with the domain D; c^i is *element* of the attribute c;

(iv) if C is a *comp* of the form $(C \equiv)C'.c$, with a domain $*D$ (D as in (ii)), then $C'.*c$ is a *comp* with the domain D; $*c$ is then a reference of the attribute c;

(v) if C is a *comp* of the form $(C \equiv)C'.c$, with a domain $\{*D\}$ (D as in (ii)), then $C'.c^i, C'.*c^i, C.*c$ are *comps* with domains $*D, D$, and $\{D\}$, respectively; $*c^i$ is element reference of the attribute c.

Each *comp* may be represented as $class\_id.c_1.c_2 \ldots c_n$ $(n \geq 1)$, where each c_i has the form *attr_name*, **attr_name*, $attr\_name^j$, or $*attr\_name^j$.

EXAMPLE 2. BOOK.$*Versions^3.Author$ is a set-valued *comp*;

BOOK.$Versions^3.Author$ is not a *comp*;

BOOK.$Cataloging^3$is a *comp*.

III *General manipulative methods.*

In almost all the methods defined in an object model of text, it is necessary to access a specific document, part of a document or a property of theirs. By operations of an extension of the relational algebra, it is possible to address a specific component or its property, for purposes of retrieving or further processing. Those operations are equivalent to operations of the relational algebra [1], but they are defined over a semantically richer model, i.e., on classes instead of relations. Since the operations are common for all the classes in the model, they may be defined in a metaclass of the model.

An operation is *unary* if it operates over a single class or a portion of a class hierarchy rooted at that class (regardless of operation parameters that can be treated as other, non-class operands). The similar holds for *binary* operations.

As an analogue to an operation Op of the relational algebra, in the object model it is possible to define two corresponding operations, *class operation* Op_c, and *extended class operation* Op_c^e. First of them takes into account only classes the operation is specified for, and the second one is applied to subclasses of those classes, too.

DEFINITION 4. (PROJECTION) Let $C_{i_1}, \ldots C_{i_m}$ be *comps* of the class declaration (1), with domains $D'_{i_1}, \ldots, D'_{i_m}$, respectively. Let all the *comps* C_{i_j} be of the form $C_{i_j} = c.c_{i_j}$, (have the common leading part). If a domain of a *comp* c is a class D' with a class declaration including attributes $a'_{i_1}, \ldots, a'_{i_m}$, and each c_{i_j} is of the form $a'_{i_j}, {a'}^i_{i_j}, *a'_{i_j}$ or $*{a'}^i_{i_j}$, then

a) *class projection* of the class declaration (1) over *comps* $C_{i_1}, \ldots, C_{i_m}$, $\Pi_c((1), C_{i_1}, \ldots, C_{i_m})$ is a class with a class declaration $P_c(a'_{i_1} : D'_{i_1}, \ldots, a'_{i_m} : D'_{i_m})$;

b) applied to a class extension r with the class declaration (1), it produces a set of projections of instances of the class extension r on attributes $a'_{i_1}, \ldots, a'_{i_m}$ (or the corresponding attribute references, attribute elements or attribute element references). Those attributes may be attributes of the class with the class declaration (1), as well as the attributes of classes which are domains of the attributes of the class (1), i.e., attributes of their class domains, etc.;

c) if a class projection $\Pi_c((1), C_{i_i}, \ldots, C_{i_m})$ is to be permanent, position of its class declaration in a class hierarchy may be determined in the following way: find the closest to the class (1), its superclass C' (if one exists), such that its set of attributes is contained in the set $\{a'_{i_1}, \ldots, a'_{i_m}\}$; then new class P_c is a subclass of the class C', and all the (direct) subclasses of the class C' become direct subclasses of the new formed class P_c. If such a superclass does not exist, new formed class is just a subclass of the root class OBJECT.

The operation of *extended class projection*, Π_c^e is defined the same way as the class projection, except that the part b) of the definition now has the following form: Π_c^e, applied to a class extension r with the class declaration (1), produces a set of projections on attributes $a'_{i_1}, \ldots, a'_{i_m}$ (or the corresponding attribute elements or references), of instances of the class extension r and of class extensions of all of its subclasses.

EXAMPLE 3. For the class BOOK and the *comp*s BOOK.$Preface$, BOOK.$*Contents^1$, class projection

$$\Pi_c(\text{BOOK}, \text{BOOK}.\textit{attributes\_of}\_\text{NL\_DOCUMENT},\ \text{BOOK}.Preface,$$
$$\text{BOOK}.*Contents^1)$$

is a class with class declaration

$$P_c(\textit{attr\_domain\_list\_of}\_\text{NL-DOCUMENT}, Preface : \text{TEXT-PARAGRAPH},$$
$$Contents : \text{CHAPTER})$$

(preface and the first chapter, including inhereted attributes). This class may be now a superclass of the class BOOK, and a subclass of the class NL_DOCUMENT. The new class extension is a projection, on the corresponding attributes, of all the instances of the class BOOK, but not of its subclass TEXTBOOK. The corresponding extended class projection, Π_c^e, produces the same class declaration, but the corresponding class extension includes, except for projection of instances of the class BOOK, projections of instances of the class TEXTBOOK, too.

DEFINITION 5. (RESTRICTION) Class and extended class restrictions, $rest_c$(*class_declaration*, *condition*) and $rest_c^e$(*class_declaration*, *condition*), respectively, are defined as follows:

For the class declaration (1), let C_1, C_2 be two *comp*s, whose domains have the same associated class, D'. Let c be a constant from a domain with the associated class D', and *Exp* an expression whose operands are *comp*s, constants and user defined methods with a value from a domain with the same associated class D'. Let $\{\sigma_1, \ldots, \sigma_k\}$ be a set of binary boolean-valued operations over the class D', where σ_1 is *equality*, and others may be $<, \leq, >, \geq$, if ordering is defined on the domain D', or other operations, e.g., *contains*, *contains-any*, *leading*, *trailing*, etc., over STRING class. Then *condition* is a boolean-valued expression of the form: $C_1 \sigma_i C_2$, $C_1 \sigma_i c$ or $C_1 \sigma_i Exp$.

Class restriction $rest_c((1), condition)$ (over a class with the class declaration (1)) is an operation that

a) produces a class declaration

$$R_c(a_1 : D_1, \ldots, a_n : D_n);$$

b) applied to a class extension r of the class with class declaration (1), produces a class extension r' of the class with class declaration (2), consisting of instances of the class extension r satisfying *condition*;

c) If the class R_c is to persist, its superclass is class *class_id* with the class declaration (1).

Extended class restriction $rest^e_c((1), condition)$ differs in item b), only. Applied to a class extension r of the class *class_id*, it produces a class extension r' of the class R^e_c with the class declaration (2), consisting of all the instances of the class r satisfying *condition*, and all the instances, satisfying condition, of union compatible projections of all the subclasses of the class *class_id*. Union compatible classes, similarly to union compatible relations, are classes with the same number of attributes, and whose attributes, in the same order, are defined on the same domains.

REMARK 2. Reason for such a position of classes R_c, R^e_c, in a class hierarchy is the fact that, although classes *class_id* and $R_c(R^e_c)$ have the same sets of attributes, integrity conditions that classes have to maintain are stronger for the class R_c than for the class *class_id*. Integrity aspect of the object model is not a subject of this paper.

EXAMPLE 4. Let *Forms*(WORD) be a user defined method, which, for a given word of a class WORD, generates (or reads from a dictionary), all the forms of that word. The class restriction operation

$$rest_c(\text{ BOOK, BOOK}.\mathit{Title}\ \mathit{contains\_anyForms}(\text{'tekst'}))$$

produces a class declaration R_c (same as for the class BOOK) and retrieves all the books whose title contains any of the word 'tekst' forms. Notice that textbooks with this property do not participate in the retrieval. The corresponding extended class restriction operation, $rest^e_c$, retrieves all the books whose title contains any of the word 'tekst' forms, including textbooks, but with attributes of the class BOOK only.

DEFINITION 6. (UNION) Class union, $\cup_c$, and extended class union, $\cup^e_c$, are operations defined on union compatible classes, in the following way:

Let r be a class extension with a class declaration $R(a_1 : D_1, \ldots, a_n : D_n)$ and s a class extension with a class declaration $S(b_1 : D_1, \ldots, b_n : D_n)$, where classes R and S are union compatible. *Class union*, $\cup_c$, and *extended class union*, $\cup^e_c$, are operations which

a) produce a class U_c (i.e., U^e_c) with class declaration

$$U_c(a_1\_alias\_b_1 : D_1, \ldots, a_n\_alias\_b_n : D_n)$$

(*alias* is another name for the same attribute);

b) extensions of classes U_c, U_c^e are defined by the following formulas:

$$r \cup_c s \stackrel{\text{def}}{=} \{\, t \mid t \text{ is an instance of the class } r \text{ or } t \text{ is an instance of the class } s \,\};$$

$$\begin{aligned} r \cup_c^e s \stackrel{\text{def}}{=} \{\, t \mid\; & t \text{ is an instance of the class } r \text{ or of an union compatible projection} \\ & \text{of any of its subclasses} \,\} \cup \{\, t \mid t \text{ is an instance of the class } s \\ & \text{or of an union compatible projections of any of its subclasses} \,\}. \end{aligned}$$

If necessary, duplicates elimination is based on identity of objects (instances), and not on identifiers equality [4].

c) if the class U_c (U_c^e) is to persist, position of its class declaration in a class hierarchy is determined in the following way: the closest (to the class R) common superclass of the classes R, S, is assigned to the class $U_c(U_c^e)$ as its (direct) superclass (if there is no other common superclass, common superclass is the class OBJECT). Classes R, S are assigned the new class U_c as their direct superclass.

Similar considerations hold for operations of class and extended class intersection and difference.

Definition 7. (Cartesian product) Let r be a class extension of a class with class declaration $R(a_1 : D_1, \ldots, a_n : D_n)$ and s a class extension of a class with class declaration $S(b_1 : D_1', \ldots, b_n : D_m')$. *Class Cartesian product*, $\times_c$ and *extended class Cartesian product*, $\times_c^e$, are operations that

a) produce classes with class declarations $CP_c, (CP_c^e)(a_1 : D_1, \ldots, a_n : D_n, b_1 : D_1', \ldots, b_m : D_m')$;

b) extensions of classes CP_c, CP_c^e are defined in the following way:

$$r \times_c s \stackrel{\text{def}}{=} \{\, t \mid t \text{ is an instance of the class } r \,\} \times \{\, t \mid t \text{ is an instance of the class } s \,\}$$

$$\begin{aligned} r \times_c^e \stackrel{\text{def}}{=} \{\, t \mid\; & t \text{ is an instance of the class } r \text{ or a compatible projection} \\ & \text{of any of its subclasses} \,\} \times \{\, t \mid t \text{ is an instance of the class } s \\ & \text{or a compatible projection of any of its subclasses } \,\}; \end{aligned}$$

c) if a class $CP_c(CP_c^e)$ is to be permanent, then it is a direct subclass of classes R and S.

Set of class and extended class operations is *extended class algebra.*

Remark 3. Class and extended class join operations, although not explicitly defined in the extended class algebra, may be expressed in terms of Cartesian product and restriction class and extended class operations. For example, expression

$$proj_c(R_c, R_c.Author),$$

where $R_c = rest_c(CP_c, CP_c.Author = CP_c.Author', CP_c.Title <> CP_c.Title')$, and $CP_c = \text{BOOK} \times_c \text{BOOK}$, is equivalent to joining the class BOOK with itself on equal values of the attribute *Author*, restricting to different titles and projecting the result onto the attribute *Author*,which means, finding sets of authors published more than one book.

THEOREM 1. *The set of classes is closed under the operations of the extended class algebra.*

Proof. Proof of the theorem is based on a graph theoretic interpretation of the object model and the corresponding extended class algebra. The graph theoretic interpretation chosen is similar to the one of [6], but extended with different kinds of type and domain constructors. Object model is represented by an *object model graph* which is a superposition of two graphs: *class inheritance graph* and *class composition graph*.

Class inheritance graph is a finite, oriented, weakly connected graph $\Gamma' = (A, R')$, where:

1. set of vertices A is a set of class identifiers;

2. set of edges R' contains an oriented edge (S, S') if and only if class S' is a subclass of the class S;

Class composition graph is a finite, oriented, labeled graph $\Gamma'' = (A, R'')$, where:

1. set of vertices A is a set of class identifiers;

2. set of edges R'' contains an oriented edge (S, S'), if the class S has an attribute a with a domain S', $\{S'\}$, $*S'$ or $\{*S'\}$; label of each edge is initially empty;

3. for each class S and each of its attributes a, labels of edges exiting the vertex S are updated in the following way:

a) if domain of a is S', ($\{S'\}$, $*S'$, $\{*S'\}$), component a ($\{a\}$, $*a$, $\{*a\}$, respectively) is added to the label of the edge (S, S');

b) components of a label are separated by commas.

Object model graph built up of a *class inheritance graph* $\Gamma' = (A, R')$ and a *class composition graph* $\Gamma'' = (A, R'')$, is a finite, oriented, weakly connected, labeled graph $\Gamma = (A, R)$, where

1. set of vertices A is a set of class identifiers;

2. set of edges R is union of sets of edges R' and R'';

3. edge labeling is the same as in the set R''; for unlabeled edges from R', empty label is assumed;

4. if there is an unlabeled edge (C', C) (on the *class inheritance graph*), than for each edge (C', D_i') (on the *class composition graph*), labeled by $a, \{a\}, *a$ or $\{*a\}$, there is an edge (C, D_i'') (on the *class composition graph*), labeled by $a, \{a\}, *a$ or $\{*a\}$ (not necessarily identically as in the edge (C', D_i')), where the vertex D_i'' is either D_i' or there is a path in the *class inheritance graph* beginning in the vertex D_i' and ending in the vertex D_i''. If vertex D_i'' is the same as D_i', and a corresponding label component is the same, it can be discarded from label of the edge (C, D_i'); if the label of the edge (C, D_i') becomes empty, the edge itself may be discarded.

Attribute methods (*comps*) and operations of the extended class algebra, have the following interpretation:

1) *comps* of a class *class_id* with class declaration (1), are *paths* in the *class composition graph* starting in vertex *class_id*:

(i) a *comp class_id* is the vertex *class_id* itself; the corresponding path ends in the same vertex *class_id* and is of 0 length;

(ii) if C is a *comp* with a domain D, and D is an identifier of a class with a class declaration $D(a'_1 : D'_1, \ldots, a'_k : D'_k)$, then a *comp* $C.a'_j (1 \leq j \leq k)$, is a path of the length length(C)+1, and containing all the edges of the interpretation of the *comp* C and an edge (D, S), whose component label is $a'_j, \{a'_j\}, *a'_j$, or $\{*a'_j\}$, if domain D'_j of the attribute a'_j is $S, \{S\}, *S$ or $\{*S\}$, respectively;

(iii) if C is a *comp* of the form $C'.c$, with a set structured domain $\{D\}$ (pointer structured domain $*D$; set pointer structured domain $\{*D\}$), then interpretation of a *comp* $C'.c^i$ ($C'.*c$; $C'.c^i, C'.*c^i, C'.*c$, respectively) is the same as for the *comp* C.

2) Operations of class and extended class projection, restriction, union, Cartesian product, may be interpreted by updating the *object model graph* in the following way:

(i) add to the *object model graph* (to both *class composition graph* and *class inheritance graph*) a vertex *result_class_name*;

(ii) add to the *class composition graph* (and the *object model graph*) edges, appropriately labeled (according to item 1(ii) of the interpretation of *comps*), between the *result_class_name* vertex and the corresponding classes which are associated with domains of attributes of the class *result_class_name*

(iii) update the *class inheritance graph*, (and thus the *object model graph*), in the following way:

a) In case of projection, let C' be a class (if one exists) from the definition of class and extended class projection, to be promoted into a direct superclass of the result class $P_c(P^e_c)$ (now generally called *result_class_name*). Delete edges (C', C_i), for each direct subclass C_i of the class C'. Add an (unlabeled) edge $(C', result\_class\_name)$; add (unlabeled) edges $(result\_class\_name, C_i)$, for each previous direct subclass C_i of the class C'. If such a class C' does not exist, just add an (unlabeled) edge (OBJECT, *result_class_name*).

b) In case of restriction of a class *class_id*, add an (unlabeled) edge $(class\_id, result\_class\_name)$.

c) In case of union (of union compatible classes R and S), let C be a class from Definition 6. of union, to be promoted into a superclass of a result class. Delete edges (C, R) and (C, S); add (unlabeled) edges $(C, result\_class\_name)$ and $(result\_class\_name, R)$, $(result\_class\_name, S)$. Similar holds for intersection and set difference operations.

d) In case of Cartesian product (of classes R and S), add (unlabeled) edges $(R, result\_class\_name)$, $(S, result\_class\_name)$.

All the updating mentioned on the *object model graph* preserves its basic characteristics of being an interpretation of an (updated) object model, including the item 4. of interpreting superclass/subclass relationship on object model graph. Thus, set of classes obtained retain their structural and inheritable characteristics (in terms of Definition 2.3) of superclasses and subclasses), which means that set of classes is closed under the defined operations of the extended class algebra.

REMARK 4. In case of dropping a class, except for deleting a corresponding vertex in all the three graphs, all the edges exiting the vertex are also deleted (on all the three graphs), and class inheritance graph (thus object model graph, too) is rearranged in accordance with superclass/subclass definition (Definition 2.3)). Specifically, it means that all the direct superclasses of the class dropped become direct superclasses of all the direct subclasses of the class dropped (with appropriate interpretation). It further implies that, for an expression of the extended class algebra, it is possible to retain, as a persistent class, only a result class of the overall expression, while ignoring intermediate results (as if they had been made permanent and then have been dropped). This way, potential explosion of classes is controlled.

6. Advantages and deficiencies of the object model of text

From the considerations in the previous sections, it is possible to conclude the following advantages of the object approach and the object methodology to document processing:

— **Segmentation of logical structure**: logical structure, which is very important for sophisticated processing, is clearly reflected through the structure of objects constituting a document. That way, a logic of a text is also reflected through internal structure that a document is given in a system using the object model, thus eliminating all the problems connected to recognition of a text structure. By this model, even a word, as the finest part of a logical structure, is completely clearly representable.

— **Representability of all the document layers**: graphical and logical structures are treated equitably, so the model proposed equally supports both preparation for visualization and intelligent processing of a document. The model provides for storing all the relevant information about a document. Internal representation of a document is of no importance and has no reflection on semantics of operations that are to be performed.

— **Extendibility of the model**: using the object approach, the model is easily extendible by new classes of documents or by adapting structure of a class to meet specific needs. This comprises that the model can be extended in the sense of specialization for a specific natural language, or its specific usage.

— **Modularity** of a document: document may be composed out of parts (of an arbitrary complexity in its logical organization) that are independently stored. It enables multi-user access to documents that are composed out of many modules that can be independently updated and changed. Such an approach provides for existence of different versions of a document sharing some common parts while different parts are saved and updated separately. There is no unnecessary repetition of data in a database.

— Updated **internal referencing**: all the objects inside internal organization of a document reference each other by means of unique object identifiers. In case of changes in contents of an object, references to it are kept updated. In visualizing a document, internal reference is mapped either into the contents of the referenced object, or into some kind of association to it (ordinal numeral, bibliographic reference indicator, etc.).

— Efficient **retrieval** of documents and document parts that contain certain information indicated in a user query (intelligent retrieval).

— **Using a document as a hypertext**: structure of a document described by this model is marked up, which provides for easy realization (in visualization) of all the aspects of a hypertextual organization and document presentation. For example, it is sufficient for each word to provide for a pointer to an object bound to the word, so as to gather all the information necessary for hypertextual usage of the document. An object bound to a word is to be presented in case additional explanation is needed, or the corresponding path for hypertextual presentation of the document is chosen. The model is naturally expanded for purpose of organizing and presenting **multimedia documents**, by adding recorded sound and picture as document parts. Further, methods for visualization and presentation need to be redefined.

Despite all the significant advantages, there are couple of deficiencies (the authors are aware of) of the model proposed. The first one concerns efficiency. Namely, there is a need to provide for efficient insertion and storing documents in accordance with the object model. Development of existing (commercial) word processing systems indicates that even such systems will offer and ask users to strictly mark up structure of their documents if they wish to have their documents used for anything more than just visualizing on a printer. Converting such marked up representations into the model proposed would be then easily accomplished and would present a possibility of converting existing document holdings into the corresponding form.

Another problem concerns organization of objects in a database: segmentation and refinement of logical structure of a document up to the word level brings to "explosion" of number of identifiers and objects. Two compromise solutions are possible: first one consists in word identification relative to a document, diminishing possibilities of document modularity, while second does not consider words as "real" objects having their own identifiers, but identifiers are assigned only to those words which "demand" for identifiers (e.g., there are other objects referencing them). Still, objects of the type WORD point to an e-dictionary, anyway. The second solution implies something "larger" granulation of logical structure (words would not be accessible "directly" but, in larger units, e.g., paragraphs), which would require additional processing in situations where accessing words is necessary.

We believe that advantages of the model significantly override deficiencies, so that the object model proposed is a good solution to managing NL documents.

7. Conclusion

Development of object methodology offers a number of advantages and is becoming a dominant approach in information processing in general, and especially in textual data processing. This methodology offers attractive possibilities for overriding starting problems in natural language input processing. In an internal representation of a document in a textual database, sufficient knowledge can be recorded for processing this kind of information. The methodology offers a possibility to use, in a very flexible way, necessary knowledge about language, stored in a computerized form (e-dictionary, electronic orthography [5]); a possibility to uniformly approach NL documents through a standard interface offering processing functions. For the Serbian language specifically, this approach offers the possibility of resolving a great number of problems (two-alphabets and mixed writings, flection, dialects, nonstandardized forms, foreign words in a text, etc.), providing for equitable usage of our language with other languages.

The object model of text developed and presented in the paper represents a frame for consistent application of object methodology to managing natural language documents. Structural and manipulative aspects are described and defined. Integrative aspect of the model has not been considered, but is important since it has impact even on structural part of the model. The model can be applied to different kinds of data, and especially to both documents and processing rules, for example, morphological, syntactic, semantic rules, rules for optimization textual queries, for accessing distributed, heterogeneous textual bases, etc.

There are several directions in which the model could be extended. One that readily comes to mind, approved by needs of natural language document processing, is definition of an union type, which may gather different classes into a single type. For example, it would be nice to be able to define an attribute *Title* to be a set consisting of words *or* numerals. Consistent extension of class definition, superclass/subclass relationship and class algebra is necessary, including this and some other features, interesting in theoretical and useful in practical senses.

REFERENCES

[**1**] Codd,E.F., *Relational completeness of database sublanguages*, in Database Systems, Courant Computer Science Symposia, ed. Rustin,R, Prentice-Hall, Engl.Cliffs, N.J. 1971, 65–98.

[**2**] Deux, O., et al., *The O_2 System*, in Communications of the ACM, Oct. 1991, 35–48.

[**3**] Duchene,H., et al., *Vodak kernel data model*, in Advances in Object-Oriented Database Systems, K.R.Dittrich (ed.), Springer-Verlag, 1989.

[**4**] Kim,W., *Introduction to Object-Oriented Database*, MIT Press, Cambridge, 1990.

[**5**] Nenadić,G., Vitas,D., *Orthography as Frame for Natural Language Standardization in Computing Environment*, Standardization and Quality in Automated Technologies, June 1995.

[**6**] Paredaens,J., Gucht,D., *Converting nested algebra expression into flat algebra expression,* in ACMTODS 17(1), March 1992, 65–93.

[**7**] Roth,M.A., et al., *Extended algebra and calculus for nested relational databases*, in ACMTODS 13(4), Dec. 1988, 390–417.

[**8**] Stonebraker,M., et al., *The Implementation of POSTGRES*, in IEEE Transaction on Knowledge and Data Engineering, March 1990.

[**9**] Vitas,D., Pavlović-Lažetić,G., Krstev,C., *Electronic Dictionary and Text Processing in Serbo-Croatian*, Linguistische Arbeiten, 293, Max Niemeyer Verlag, Tübingen 1993 (1), 225–231.

(received 14.09.1995.)

Faculty of Mathematics, Studentski trg 16, Beograd, Yugoslavia