Handling Semi-Structured Data through an Extended Object-Oriented Data Model

Giovanna Guerrini *Isabella Merlo* *Marco Mesiti*

Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova
Via Dodecaneso 35 - I16146 Genova, Italy
{guerrini,merloisa,mesiti}@disi.unige.it

Abstract In traditional database applications the structure of data is pre-defined, and data are entered into the database specifying the schema element (relation or class, depending on the paradigm) they belong to. New emerging database applications, expecially those related to the Web, are characterized by data that have an irregular, heterogeneous, partial structure that quickly evolves. In this paper we adapt an object-oriented data model to this kind of data, by providing it with more flexible type system and by weakening the conditions for membership of an object to a class. An approach to classify objects, created without specifying the class they belong to, in the most appropriate class of the schema, is also proposed.

1 Introduction

In the last few years, there has been in the database community a growing interest in the management of *semi-structured* data [1]. Semi-structured data are data whose structure is not regular, is heterogeneous, is partial, has not a fixed format and quickly evolves. Moreover, the distinction between the data described by the structure and the structure itself is blurred. Those characteristics are typical of data available on the Web [3], of data coming from heterogeneous information sources [18], and so on. The lack of a fixed a-priori schema and of information on the data structures makes it difficult handling semi-structured data through conventional database technology.

The research community has proposed two main approaches to model semi-structured data [5,9]. The first one is a more traditional approach and consists of adapting existing data models to deal with semi-structured data. In particular, extensions to the object-oriented data model have been proposed with less restrictive type systems [8,20]. The second approach, by contrast, does not have any notion of type and schema to avoid any restriction on the structure of the data to be stored in the database. The basic idea of this approach [2,7] is to use a labeled graph to store structural information together with data they refer to. An advantage of the first approach over the second one is the existence of a structure containing information on the type of data separated from the data themselves. This is important for efficiently querying data and for developing adequate storage structures and indexing techniques.

In this paper we consider an extended object-oriented data model, well-suited for representing semi-structured data. This data model includes some new types ensuring a highly flexible type system. In particular, its modeling power is comparable to that of the best known data models for semi-structured data, such as [8,20], in that it captures all the kinds of data heterogeneity that can be represented in those models. It enriches a traditional object-oriented type system with a `spring` type, allowing to express that the domain of an attribute is not specified, and with *union types*, allowing one to model attributes that can take values from different domains.

The idea of handling semi-structured data in an existing a-priori database schema could seem too restrictive. Thus, in our approach, we introduce the notion of*weak membership* that represents a compromise between the flexibility of semi-structured data and the rigidity of object-oriented schemas. In our model, a *semi-structured object* is an object that has been created without specifying the class it belongs to and whose structure may not exactly match any of the classes in the database schema. In the context of semi-structured data, the assumption that for each object there is a type exactly describing it is indeed too strong. Thus, in our model we do not make such assumption and we rely on a notion of weak membership. Such notion is weaker than the classical notion of class membership, since we only require the components[1] in the object state be a subset of the components of the *structural type* of the class,[2] rather than requiring the components of the object state be exactly all and only those appearing in the structural type of the class, as in traditional object-oriented data models.

According to our notion of weak membership, an object can be a weak member of no class, of just one class or of several classes, even not related by inheritance hierarchies. To determine the *most appropriate* class for an object, among the ones of which the object is a weak member, we use two measures: the *conformity degree*, measuring the similarity degree between the type of the semi-structured object and the structural type of the class, and the *heterogeneity degree* of the class, measuring how much the extension of the class is heterogeneous.

The remainder of the paper is organized as follows. In Section 2 we present the basic concepts of the extended data model. In Section 3 we introduce the notion of weak membership, whereas in Section 4 we discuss the proposed classification approach. Finally, in Section 5 we conclude by discussing some directions of ongoing and future work.

---

[1] A *component* of a record value or of a record type is one of the slots composing it.

[2] The structural type of a class is the record type containing the attributes of the class and their respective domains.

2 Data Model

Our data model [15] is defined as the extension to the object-oriented data model presented in [11] (denoted in the following as the *basic object-oriented data model*) and is based on a type system which consists of three kinds of types: *value types*, *object types*, and the `spring` *type*. **Value types** are classical types such as *basic value types* (`integer`, `bool`, `real`, etc.) and *structured types* (built by means of `record`, `set` and `list` constructors). The reference data model adds to this set of types the *union type*, that we will discuss in more details below. **Object types** are types corresponding to classes (class names). Finally, the **`spring` type** is a new type, not present in the basic object-oriented data model, allowing one to specify that an attribute does not have any specific domain. Because of the relevance of this type in handling semi-structured data, we will also discuss it in more details below. It is important to remark that the reference data model, as the basic object-oriented data model, supports all the common features of object-oriented data models such as object identity, user-defined operations, classes, inheritance.

In the remainder of this section we first discuss the new types added to the basic data model and then we introduce the notions of class and object as supported by the model. Moreover, we discuss subtyping and inheritance rules.

2.1 Union Types

A **union type** consists of a set of types belonging to the basic type system each one associated with a distinct label. Let $T_1, \ldots, T_n$ be value types of the basic object-oriented data model or object types and $a_1, \ldots, a_n$ be distinct labels, then the type *union-of(*$a_1 : T_1, \ldots, a_n : T_n$*)* is a union type. Legal values for a union type are pairs $l : v$, where l is the label of a union type component, and v is a legal value for the type associated with l.

As a consequence of the introduction of union types, record types are modified to allow one to omit the label associated with a component whose type is a union type. In this way, in order to access that component, one only needs to use the label appearing in the union type definition.

Example 1. Let `Abstract` be a class name representing fundamental features of paper abstracts. Let `record-of (title:string, union-of(keyword: list-of (string), abstract:Abstract))` be a record type. Let X be a variable of this type. In order to access the `abstract` component, we only write X`.abstract`. $\diamond$

To avoid ambiguities in accessing a component of a record type, we impose that the labels of record type components and the labels of union type components be all distinct. That is, we disallow record types such as `record-of(keyword: string, union-of(keyword: list-of(string),abstract:Abstract))`.

A legal value, for a record type, has the form $(a_1{:}v_1, \ldots, a_n{:}v_n)$, where a_i is the label of a record type component or the label of a union type component appearing in the record type definition, and v_i is a legal value for the type associated with a_i in the corresponding record type definition. For example, let i_a be the identifier of an object belonging to the class `Abstract`, then both (`title` :'*Querying Semi- Structured Data*', `keyword` :['*query*', '*semi-structured data*']) and (`title` :'*Modelling and Querying...*', `abstract`:i_a) are legal values for the type of the previous example.

2.2 Spring Type

The **`spring type`** is the common supertype of value types and object types. The introduction of this type allows us to manage data without knowing their actual type. Each legal value of each type of the model is a legal value for the `spring` type. Note that our notion of `spring` type is different from the notion of `Object` type, supported by some systems like GemStone [4]. The first difference is that in our model we have both value types and object types, whereas those systems only support object types. The `spring` type in our model is not an object type and is not a value type, rather it is a common supertype of all (value and object) types of the model. Another relevant difference is that in our model the `spring` type cannot be directly instantiated, that is, no objects or values can be proper instances of the `spring` type. In other systems, by contrast, objects can be proper instances of the `Object` type.

2.3 Classes and Objects

Our model supports a quite standard notion of class, with some differences arising from the introduction of union and `spring` types. Each class has a structural type, which is a record type describing the state of the class instances, formally defined as follows.

Definition 1. (Structural type of a class). *Given a class c, defined as*

$$\begin{aligned} &\textit{class } c\ \{a_1 : T_1, \ldots, a_m : T_m, \\ &\qquad \textit{union-of}(a_1^1 : T_1^1, \ldots, a_1^p : T_1^p), \ldots, \textit{union-of}(a_n^1 : T_n^1, \ldots, a_n^p : T_n^p)\ \} \end{aligned}$$

the record type $\textit{record-of}(a_1 : T_1, \ldots, a_m : T_m, T_{m+1}, \ldots, T_{m+n})$*, where, for* $k = 1, \ldots, n$*:* $T_{m+k} = \textit{union-of}(a_k^1 : T_k^1, \ldots, a_k^p : T_k^p)$*, is the structural type of class c, denoted by* $stype(c)$*.* □

Note that, as specified in the definition above, the class contains some fixed attributes $(a_1, \ldots, a_m)$, and some other components for which one out of some possible alternatives, specified through a union type, can be chosen (components $m+1$ to $m+n$).

The notion of object supported by the model, formalized by the following definition, is also quite standard.

Definition 2. (Object). *An object is a triple* $o = (i, v, c)$ *where i is an object identifier, v is a record value (the object state) and c is the most specific class to which o belongs.* □

Finally, the following definition states the conditions for an object to be an instance of a class.

Definition 3. (Instance). *An object o is an* instance *of a class c if $o.v$ is a legal value for $stype(c)$.* □

Definition 3 above requires that the following conditions hold:

(1) for each component $a : v$ of the object state, a component $a : T$ exists in $stype(c)$ such that v is a legal value for T or a component $union\text{-}of(a_1 : T_1, \ldots, a_p : T_p)$ exists such that $a : v$ is a legal value for that component, that is, $\exists\ i, 1 \leq i \leq p$, such that $a = a_i$, and v is a legal value for T_i;
(2) for each component $a : T$ in $stype(c)$, a component $a : v$ exists in the object state such that v is a legal value for T, and for each component $union\text{-}of(a_1 : T_1, \ldots, a_p : T_p)$ in $stype(c)$ a component $a : v$ exists in the object state such that $a : v$ is a legal value for that component, that is, $\exists\ i, 1 \leq i \leq p$, such that $a = a_i$ and v is a legal value for T_i.

Condition (1) above requires that each component in the object state corresponds either to an attribute of the class (and in this case the component value must be a legal value for the attribute domain) or to one of the components of a union type in the structural type of the class (and in this case the component value must be a legal value for the union type component domain). Condition (2) above, by contrast, requires that the object state contains a component for each attribute of the class and a component for each union type in the structural type of the class (corresponding to one of the components of the union type).

The following is an example of classes and objects in our model.

Example 2. Suppose we wish to model information about articles that we have gathered from the Web. In this case articles may have some common features, whereas other features may be typical of particular ones. Suppose we want to model the title, the author name, the abstract, the date in which the article is published and the test. The name may be a `string`, or a `record` with two components, author first name (`a-name`) and surname (`a-sname`), and the text may contain anything (for example, images, tables, or simple strings). Let `Date` be a class of the database schema and i_d be the identifier of an object instance of class `Date`. Let i_a and i'_a be two object identifiers of class `Abstract`. We may define a class `Article` whose structural type is:

```
record-of(union-of(authorS:string, authorR:record-of(a-name:
          string, a-sname:string)), title:string,
          abstract:Abstract, published:Date, text:spring).
```

The following objects are instances of class `article`: $o_1 = (i_1, v_1,$ `Article`$)$, where v_1 is (`authorS`:'Buneman', `title`:'Semi-Structured Data', `abstract`:i_a, `published`:i_d, `text`:'*The topic...*'), and $o_2 = (i_2, v_2,$ `Article`$)$, where the value v_2 is (`authorR`:(`a-name`:'Serge', `a-sname`:'Abiteboul'), `title`: 'Querying...', `abstract`: i'_a, `published`:i_d, `text`:`record-of(...)`). Even if their states are legal values for the structural type of `Article`, they have different structures. ◇

2.4 Subtyping and Inheritance

The notion of *subtyping* has been extended to manage union types and the `spring` type. In the following definition the ISA hierarchy denotes the inheritance relationship among classes established by the user. The ISA relationship is denoted as a function that for each class in the schema returns the set of its direct superclasses. The function ISA^* denotes the transitive closure of the ISA relationship.

The subtype relationship, denoted by $\leq_T$, is defined as follows. Note that the subtype relationship for basic types is the identity.

Definition 4. (Subtypes). *Given T_1, T_2 types of the reference model, T_2 is subtype of T_1 (denoted as $T_2 \leq_T T_1$) if and only if one of the following conditions holds:*

1. $T_1 =$ `spring`;
2. $T_1 = T_2$;
3. $T_1 \in ISA^*(T_2)$;
4. $T_2 = set\text{-}of(T_2')$, $T_1 = set\text{-}of(T_1')$ *and* $T_2' \leq_T T_1'$;
5. $T_2 = list\text{-}of(T_2')$, $T_1 = list\text{-}of(T_1')$ *and* $T_2' \leq_T T_1'$;
6. $T_2 = record\text{-}of(a_1 : T_1'', \ldots, a_m : T_m'', T_{m+1}'', \ldots, T_{m+n}'')$, $T_1 = record\text{-}of(a_1 : T_1', \ldots, a_m : T_m', T_{m+1}', \ldots, T_{m+n}')$ *and for each* i, $1 \leq i \leq m+n$, $T_i'' \leq_T T_i'$;
7. $T_2 = union\text{-}of(a_1 : T_1'', \ldots, a_m : T_m'')$, $T_1 = union\text{-}of(a_1 : T_1', \ldots, a_m : T_m', a_{m+1} : T_{m+1}', \ldots, a_{m+n} : T_{m+n}')$ *and for each* i, $1 \leq i \leq m$, $T_i'' \leq_T T_i'$;
8. $T_2 = record\text{-}of(a'' : T'')$, $T_1 = union\text{-}of(a_1 : T_1', \ldots, a_n : T_n')$, *and* j $(1 \leq j \leq n)$ *exists s.t.* $a'' = a_j$ *and* $T'' \leq_T T_j'$. □

Note that in the reference model there exists a unique root type, the `spring` type. That type is the supertype of all types in the model and ensures that given two types their common supertype surely exists.

3 Weak Membership

In order to achieve the flexibility needed to handle semi-structured data, we weaken the notion of instance of traditional object-oriented data models in the notion of *weak membership*, only requiring condition (1), stated after Definition 3. Thus, the structural type of a class may have more components than those appearing in the object state. In such a case, we need some exception-handling mechanism to manage accesses to components not present in the classified object.

In order to formally define the notion of weak membership and to define a method to check whether an object is a weak member of a class, we extend a well-known theoretical notion, the simulation relation [16]. First, we provide an abstract representation of the structural type of a class, the *class structural expression*, and an abstract representation of the object state, the *object value expression*. Then, to verify whether the object is a weak member of the class, we check whether a particular simulation exists between those two expressions.

Intuitively, the class structural expression is a tree labeled with symbols representing the attributes of the class and their types, whereas the object value expression is a tree labeled with symbols representing the attributes of the object and their values. In the remainder of this section, we first present the formal definitions concerning class and object expressions (Subsection 3.1) and then the weak membership notion is formally defined (Subsection 3.2).

3.1 Class and Object Expressions

In the following the set $\mathcal{PRED}$ denotes a set of predicates where each predicate represents the set of legal values for basic value types and object types. A predicate $p \in \mathcal{PRED}$ applied to a value v holds if and only if v belongs to the set of instances associated with the type p, where the type p may be a basic value type or an object type. Moreover, given the set $\mathcal{AN}$ of attribute names, $\mathcal{LT}$ denotes the set of tree labels, that is $\mathcal{LT}$ = {LIST, REC, SET, UNION, SPRING} $\cup\ \mathcal{AN} \cup \mathcal{PRED}$. The following definition states the notion of *class structural expression.*

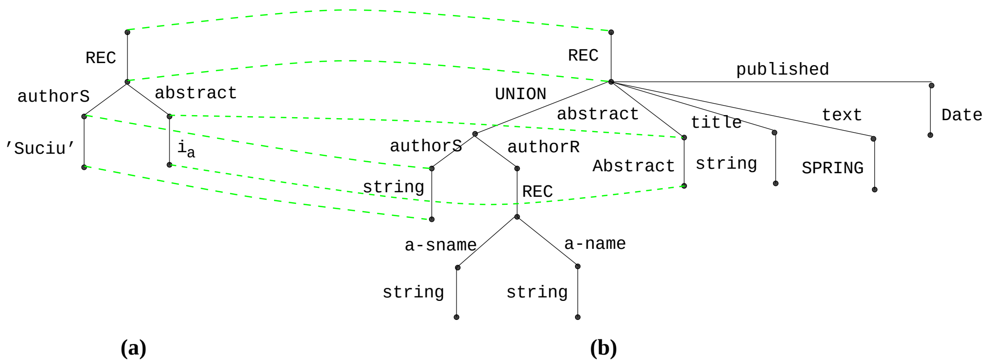


Figure1. (a) Object value expression, (b) class structural expression and their simulation relation

Definition 5. (Class structural expression). *Given a class c, the class structural expression of c (denoted by $\varepsilon_t(c)$) is a tree (V_t, E_t, φ_t), labeled on $\mathcal{LT}$, where V_t is a set of vertices, $E_t \subseteq V_t \times V_t$ is a set of edges and, $\varphi_t : E_t \rightarrow \mathcal{LT}$ is the edge labeling function.* □

Figure 1(b) shows the class structural expression associated with class `Article` of Example 2. Note that `string`, `Date`, and `Abstract` symbols are predicates which represent the set of legal values for the corresponding types.

In the following definition, stating the notion of *object value expression*, $\mathcal{LV}$ denotes the set of labels of object value expressions, that is, $\mathcal{LV}$ = {LIST, REC, SET, UNION, NULL} $\cup\ \mathcal{AN} \cup \mathcal{V}$, where $\mathcal{V}$ denotes the set of legal values for basic value types and object identifiers.

Definition 6. (Object value expression). *Given an object o, the object value expression of o (denoted by $\varepsilon_v(o)$) is a tree (V_v, E_v, φ_v), labeled on $\mathcal{LV}$, where V_v is a set of vertices, $E_v \subseteq V_v \times V_v$ is a set of edges and, $\varphi_v : E_v \to \mathcal{LV}$ is the edge labeling function.* □

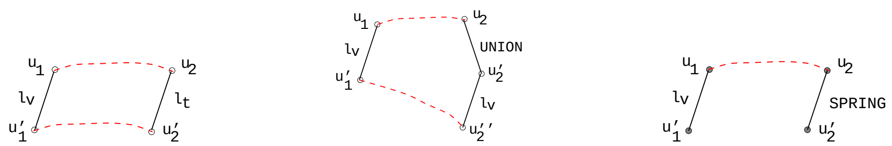
Figure 1(a) shows the object value expression associated with a semi-structured object whose state is (`authorS`:'*Suciu*', `abstract`:i_a). According to our shallow approach, we have not generated the object value expression associated with the state of the object identified by i_a.

3.2 Simulation Relation

Before defining the relation between the class structural expression and the object value expression we introduce a mapping between labels in set $\mathcal{LV}$ and labels in set $\mathcal{LT}$, that is used to identify a set of cases to be managed in the same way.

Definition 7. (Relation $\approx_{\mathcal{L}}$). *A relation $\approx_{\mathcal{L}}$ holds between a label $l_v \in \mathcal{LV}$ and a label $l_t \in \mathcal{LT}$ (denoted by $l_v \approx_{\mathcal{L}} l_t$), if and only if one of the following conditions holds: (1) $l_v = NULL$ and $l_t \neq SPRING$; (2) $l_v, l_t \in \{LIST, REC, SET, UNION\} \cup \mathcal{AN}$ and $l_v = l_t$; (3) $l_t \in \mathcal{PRED}$ and l_t holds on l_v.* □

We are now able, using relation $\approx_{\mathcal{L}}$, to introduce our definition of *simulation*, which is a variation of the classical notion [6]. In the definition *root(A)* denotes the root of tree A and $u \xrightarrow{l} u'$ denotes an edge (u, u') such that $\varphi((u, u')) = l$.



(a) (b) (c)

Figure2. Visual representation of relation among vertices of item (2) of Definition 8

Definition 8. (Simulation). *A binary relation $\mathcal{R}$ from the vertices of $A_v = (V_v, E_v, \varphi_v)$ labeled on $\mathcal{LV}$ to the vertices of $A_t = (V_t, E_t, \varphi_t)$ labeled on $\mathcal{LT}$, is a simulation if and only if the following conditions hold:*

1. *$root(A_v)\ \mathcal{R}\ root(A_t)$;*
2. *if $u_1\ \mathcal{R}\ u_2$, then $\forall u_1 \xrightarrow{l_v} u'_1$ in E_v, $\exists\ u_2 \xrightarrow{l_t} u'_2$ in E_t, such that one and only one of the following conditions holds:*
   (a) $l_v \approx_{\mathcal{L}} l_t$ and $u'_1\ \mathcal{R}\ u'_2$,

(b) $l_t = UNION$, $\exists u'_2 \xrightarrow{l'_t} u''_2$ *in* E_t *such that* $l_v = l'_t$ *and* $u'_1 \; \mathcal{R} \; u''_2$,
(c) $l_t = SPRING$. □

In Figure 1 the dashed lines represent the simulation between the object value expression associated with the semi-structured object, whose state has been introduced previously, and the structural expression associated with the structural type of class `Article` of Example 2. A visual representation of relation among vertices of item (2) of Definition 8 is shown in Figure 2. The dashed lines identify the relation that must hold between the vertices of the two trees. Note that, as you can see in Figure 2(c), we do not require the relation to hold between vertices u'_1 and u'_2.

For determining weak membership, we do not consider every simulation. For example, consider the object value expression associated with the object state (`a`: 5, `b`:'*rose*') and the class structural expression associated with the structural type `record-of(union-of(a:integer, b:string))`. According to Definition 8 a simulation exists between them. The simulation in this example, however, does not capture our notion of the set of legal values for the record type. The idea of the union type is, instead, that of choosing *one* out of some possible alternatives. Thus, in the definition of weak membership, we leave out this kind of simulations, as formally stated by the following definition.

Definition 9. (Weak membership). *An object o is a weak member of a class c if a simulation* $\mathcal{R}$ *exists between the object value expression associated with o* $(\varepsilon_v(o))$ *and the class structural expression associated with c* $(\varepsilon_t(c))$*, such that* $\forall u_2 \xrightarrow{UNION} u'_2$ *labeled edge of* $\varepsilon_t(c)$ *at most one pair* $(u, u') \in \mathcal{R}$ *exists such that* $u' \in \{\bar{u} \mid (u'_2, \bar{u})$ *is an edge of* $\varepsilon_t(c)\}$. □

4 Automatic Classification Approach

In the management of semi-structured objects we want to emphasize the role of the class as a repository that contains objects whose states have the same type,[3] rather than as a template for creating objects. In this context, we allow applications to create objects without specifying the class they belong to. Then, it is the system that automatically classifies those objects in an appropriate class. The definition of weak membership, presented in the previous section, does not allow one to identify only one class to which the object belongs. In this section we propose an approach to establish the most appropriate class the object can be classified in. If no class exists of which the object is a weak member, we insert it into a repository of unclassified objects. As the schema evolves the repository is periodically examined, trying to classify objects contained in it.

In the remainder of this section we propose two measures to select the most appropriate class where we can classify a given object, among those of which the object is a weak member. We also outline an algorithm using those measures to automatically classify semi-structured objects.

[3] Note that, in our model, this condition does not mean that all objects instances of a class have the same structure (cfr. Example 2).

4.1 Conformity Degree

With the first measure, referred to as *conformity degree*, we determine how much the type of the semi-structured object is close to the structural type of a given class, that is, how many components the class has in addition to those of the object. In case an object is a weak member of more than one class, we select the classes that have the minimal number of additional components with respect to the components in the object state. To formally define the conformity degree, we introduce an additional data structure, referred to as *object structural expression*, representing the actual type of the object. This data structure, intuitively, is a subtree of the tree associated with the structural type of a class of which the object is a weak member. It is associated with a legal type of our type system and allows the actual type of the object to be compared with the structural type of the class, since the object structural expression is built starting from the class structural expression. To generate this structure we start from the existing simulation between the object value expression and the class structural expression and extract the vertices of the class structural expression that appear in the second component of the simulation. Then, we add to this set of vertices other vertices to handle two particular cases: the presence of null values in the object state and the presence of `spring` types in the structural type of the class. The edges and the labeling function of this tree are created accordingly. For further details on the formal definition of the object structural expression, that will be denoted by $\varepsilon(o, c)$, we refer the reader to [15]. Figure 3(a) shows the object structural expression associated with the object value expression shown in Figure 1(a). As we can see, this object structural expression represents the type `record-of(authorS:string, abstract:Abstract)`. The value associated with the object value expression shown in Figure 1(a) is a legal value for that type. Moreover, to formally define the conformity degree, we must take into account that when there is a union type in the structural type definition of a class, only one of its components may appear in the object state. Thus, we consider the *real paths* of a class structural expression. Real paths are paths that do not contain any edge labeled by $UNION$ followed by an edge labeled by l ($l \in \mathcal{AN}$) where l is an attribute not appearing in the object state. Figure 3(b) shows the tree only containing the real paths of the class structural expression shown in Figure 1(b). The following definition formalizes the notion of conformity degree.

Definition 10. (Conformity degree). *Let o be a semi-structured object and c be a class such that o is a weak member of c. We define the conformity degree of o with respect to c (denoted by $C^\circ(o, c)$), as the ratio of the number of paths of the object structural expression and the number of real paths of the class structural expression. Formally:*

$$C^\circ(o, c) = \frac{card(path(\varepsilon(o, c)))}{card(real\text{-}path(\varepsilon_t(c)))}$$

□

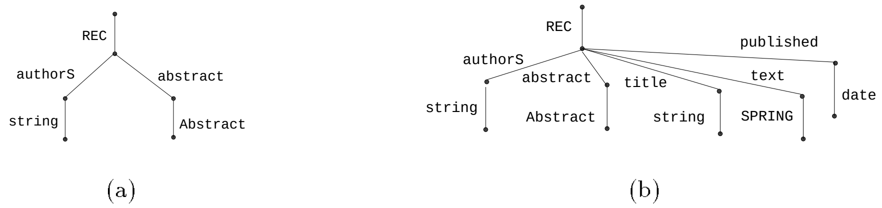


Figure3. (a) Object structural expression, and (b) the part of class `Article` structural expression containing only the union components that appear in the object state

In the previous example, the number of paths of the object structural expression is 2, the number of real paths of the class structural expression is 5, thus the conformity degree is 0.4.

The conformity degree is always a number between 0 (low conformity) and 1 (high conformity). If a semi-structured object is an instance of a class, the conformity degree is 1 and if a semi-structured object is a weak member of a class and the conformity degree is 1, then the object is an instance of the class.

4.2 Heterogeneity Degree

With the second measure, referred to as *heterogeneity degree*, we want to check how much the extension of a class is heterogeneous. By using the heterogeneity degree, we can insert a given object in the class with the most homogeneous extension. The advantage of having classes with a homogeneous extension is that more efficient query execution strategies and storage organizations are possible. In Section 2 we have seen that, because of the presence of union and `spring` types in the type system, several structures may correspond to the same type. This fact affects the heterogeneity of the extensions of classes in which union types and `spring` types are used. We have evaluated that the presence of an union type in the class definition generates an heterogeneity degree equal to the number of types present in the union type. By contrast, the presence of a `spring` type generates an heterogeneity degree equal to the number of all value and object types of the schema (denoted by $\mathcal{VT}$ and $\mathcal{CI}$, respectively). The heterogeneity degree of a record type is the product of the heterogeneity degree of its components, while the heterogeneity degree of a set type (list type) is the heterogeneity degree of its component type. The heterogeneity degree of other types (belonging to the basic type system) is 1 since they do not generate heterogeneous extensions. The following definition states how the heterogeneity degree of a class is computed.

Definition 11. *(*Heterogeneity degree*). Let $T = stype(c)$ be the structural type of class c, then the heterogeneity degree associated with c is the value returned* *by the following function applied to T.*

$$H^\circ(T) = \begin{cases} 1 & \textit{if } T \textit{ is a basic value type or object type} \\ n & \textit{if } T = \textit{union-of}(a_1 : T_1, \ldots, a_n : T_n) \\ card(\mathcal{VT}) + card(\mathcal{CI}) & \textit{if } T = \texttt{spring} \\ \prod_{i=1}^{m+n} H^\circ(T_i) & \textit{if } T = \textit{record-of}(a_1 : T_1, .., a_m : T_m, T_{m+1}, .., T_{m+n}) \\ H^\circ(T') & \textit{if } T = \textit{list-of}(T') \textit{ or } T = \textit{set-of}(T') \end{cases}$$

□

4.3 Classification Algorithm

In our classification approach we look for a class such that: the semi-structured object is a weak member of the class with the highest conformity degree; the class has the lowest heterogeneity degree. The classification algorithm takes as input a semi-structured object and executes the following steps:

1. The set of classes of which the object is a weak member is computed; such set is denoted as WMS. If $WMS = \emptyset$ then the object cannot be classified and it is simply inserted in the repository of unclassified objects.
2. The set of classes $WMS_{C\text{-}max}$ is extracted from the set WMS by choosing the classes with respect to which the object has the highest conformity degree. If this set is a singleton, the most appropriate class has been found and the object is inserted in the class extension. Otherwise,
3. The set of classes $WMS_{H\text{-}min}$ is extracted from the set $WMS_{C\text{-}max}$ by choosing the classes with the lowest heterogeneity degree. If this set is a singleton, the most appropriate class has been found and the object is inserted in the class extension. Otherwise, an arbitrary class is selected in which the object is inserted.

In the previous algorithm, we find out the set of classes having the highest conformity degree (`step 2`) from the classes of which the object is a weak member(`step 1`). We use the conformity degree as the main measure in the classification approach because it allows one to identify the classes with the smallest number of attributes not present in the object state. At this point we try to minimize the heterogeneity degree (`step 3`).

5 Conclusions and Future Work

In this paper we have discussed how semi-structured data can be handled through an object-oriented data model, in particular we have discussed extensions to an object-oriented data model to make it better-suited to manage this kind of data. First, the type system has been extended to provide it with the flexibility required by semi-structured data; then, the notion of membership to a class has been weakened to allow the model to handle objects that do not completely adhere to the structure of any class; finally, an approach has been considered for classifying objects in the most appropriate class of the schema. In our opinion, this approach is a good balance between the flexibility of semi-structured data and the rigidity of a fixed, a-priori defined, object-oriented schema.

It is important to remark that the problem of automatically classifying information has also been investigated in other areas, such as that of terminological languages [19]. However, semi-structured data have features requiring the development of specific classification techniques. In the context of semi-structured data the problem of automatic typing has been addressed in [17], that investigate how to extract structure from raw data. They however do not exploit any a-priori knowledge on the schema. This knowledge, that we assume in our approach, often occurs in practice, for instance when integrating semi-structured data, discovered on the Web, with data having a known structure or when the semi-structured data have associated some kind of structural information (for example the Document Type Definition associated with an XML page [14]).

The proposed approach is currently being implemented, and could be extended along a number of directions, some of which we discuss in the following.

Extensions to the Model. A first extension that could be considered is that of classifying an object in more than one class, rather than always forcing the selection of a single class. This could be useful when there are several classes of which the object is a weak member, with the same values for the conformity and the heterogeneity degrees. Moreover, our current notion of weak membership is based on the fact that the object state contains less components than those of the class. Such notion can be extended to the case of objects whose state contains additional components with respect to those specified in the class, in the same spirit of the O_2 exceptional instances [10]. In this way we can achieve a more accurate classification method. Another possible extension could be that of allowing components to be dynamically added, or deleted, to the state of objects in the database. This could require a re-classification of the object, that is, a migration of the object in a more appropriate class.

Schema Adaptation. We would like to consider the possibility that the schema evolves, as a consequence of object classification. A first form of evolution is concerned with the generalization of attribute domains. Whenever an object could be classified in a class, if we only consider attribute names appearing in its state, but for some of its components the value does not belong to the attribute domain in the class, then the attribute domain in the class could be generalized to include that value as a legal value, so that the object can be classified in the class. For example, if an object whose state contains a component `a:`'*john*' conforms to a class `c` except for the domain of `a` in `c` that is `integer`, then the object is classified in `c` whose domain for `a` becomes `union-of(integer, string)`. In this case we use a more traditional definition of union types in which no labels are associated with types. Note that, however, this may lead to some problems in recognizing the actual type of a value.

A more radical form of evolution should be that of periodically restructuring the schema, adapting it to the actual data stored in the database. That is, if more than a given percent of the objects classified in a given class do not have a value for a given attribute, the attribute is removed from the class (and the objects providing a value for it are treated as exceptional instances). Conversely, if more than a given percent of the objects classified in a given class have an additional attribute, a subclass is added to the schema containing this additional attribute. Moreover, when a class has less than a given percent of proper instances, it is removed from the schema and its objects re-classified.

Extraction Tool. In [12,21] some techniques to extract structural information from documents are presented. These approaches are very "*problem oriented*" and it is very difficult to generalize them for any kind of documents. The big problem is the absence of a general format to represent documents that allows one to associate a semantic meaning to some parts of the documents. We think that the use of the Document Type Definition of XML may help to solve this problem. Our idea is to use *data mining techniques* [13] to split a set of documents in subsets having a quite similar structure using the structural information present in the document. In particular, two approaches are possible. The first one is to define "*prototype documents*" to which the documents are compared. If a document is similar to a prototype document, using some measure to evaluate the similarity, then it is classified in the extension of the prototype document. The latter is to use decision trees with rules that specify conditions on the attribute types. In this way a path in the decision tree may represent a particular type to which a set of objects may belongs to.

Query Language. A query language for the presented data model is being defined. This language provides the basic features of Web query languages.

The applicability of the classification approach to Web search engines, to perform content-based queries, will also be investigated. The idea is to define, starting from the query, the value to be searched on the Web, to associate a structural expression with HTML pages (seen as objects), and then to verify whether a simulation exists between the tree associated with the query and the tree associated with the object. If the simulation exists then the HTML page is a possible answer for the query. If structural information, such as a schema definition and a classification of objects into classes, are available, they can also be exploited to reduce the cost of evaluating the query, by first looking for a simulation between the tree associated with the query and the tree associated with the class.

Acknowledgments We wish to thank Elisa Bertino, who supervised our work on semi-structured data management, providing us several useful suggestions. We also wish to acknowledge the financial support for the work reported in this paper by the Italian MURST under the Interdata Project.

References

1. S. Abiteboul. Querying Semi-Structured Data. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 1–18, 1997.

2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1):68–88, 1996.
3. S. Abiteboul and V. Vianu. Queries and Computation on the Web. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 262–275, 1997.
4. R. Breitl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. Addison-Wesley, 1989.
5. P. Buneman. Semistructured Data. In *Proc. of 6th ACM SIGACT-SIGMOD-SIGART Symposium on PODS*, pages 117–121, 1997. Tutorial.
6. P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 336–350, 1997.
7. P. Buneman, S. Davidson, D. Suciu, and G. Hillebrand. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 505–516, 1996.
8. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 313–324, 1994.
9. S. Cluet. Modeling and Querying Semi-Structured Data. In M. T. Pazienza, editor, *Information Extraction. LNAI 1299*, pages 192–213, 1997.
10. O. Deux et al. The Story of o_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.
11. G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. *Journal of Intelligent Information Systems*, 11(1):5–40, 1998.
12. J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web, 1997. Available via anonymous ftp at ftp://db.stanford.edu/pub/paper/extract.ps.
13. M. Heikki. Methods and Problems in Data Mining. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 41–55, 1997.
14. S. Holzner. *XML Complete*. McGraw-Hill, 1998.
15. M. Mesiti. An Object-Oriented Data Model for Semi-Structured Data. Master's thesis, University of Genova – Department of Computer Science (DISI), April 1998. In Italian.
16. R. Milner. An Algebraic Definition of Simulation between Programs. In *Proc. of the 2nd IJCAI*, pages 481–489, London, UK, 1971.
17. S. Nestorov, S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Data. In L. M. Haas and A. Tiwary, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 295–306, 1998.
18. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. of the 11th Int'l Conf. on Data Engineering*, pages 251–260, 1995.
19. C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK System Revisited. Technical Report KIT - Report 75, Technische Universitat Berlin, 1989.
20. F. Rabitti. *The Multos Document Model*, volume Human Factors in Information Technology of *6*, chapter 3, pages 17–52. North-Holland, 1990.
21. D. Smith and M. Lopez. Information Extraction for Semi-Structured Documents. In ACM, editor, *Proc. of 6th ACM SIGACT-SIGMOD-SIGART Symposium on PODS*, pages 117–121, Tucson, Arizona, May 1997.