An Approach to Classify Semi-Structured Objects*

Elisa Bertino[1] *Giovanna Guerrini*[2] *Isabella Merlo*[2] *Marco Mesiti*[3]

[1] Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39/41 - I20135 Milano, Italy
`bertino@dsi.unimi.it`

[2] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso 35 - I16146 Genova, Italy
`{guerrini,merloisa}@disi.unige.it`

[3] Bell Communications Research
445, South Street - Morristown (NJ), U.S.A.
`marco@research.bellcore.com`

Abstract. Several advanced applications, such as those dealing with the Web, need to handle data whose structure is not known a-priori. Such requirement severely limits the applicability of traditional database techniques, that are based on the fact that the structure of data (e.g. the database schema) is known before data are entered into the database. Moreover, in traditional database systems, whenever a data item (e.g. a tuple, an object, and so on) is entered, the application specifies the collection (e.g. relation, class, and so on) the data item belongs to. Collections are the basis for handling queries and indexing and therefore a proper classification of data items in collections is crucial. In this paper, we address this issue in the context of an extended object-oriented data model. We propose an approach to classify objects, created without specifying the class they belong to, in the most appropriate class of the schema, that is, the class closest to the object state. In particular, we introduce the notion of weak membership of an object in a class, and define two measures, the conformity and the heterogeneity degrees, exploited by our classification algorithm to identify the most appropriate class in which an object can be classified, among the ones of which it is a weak member.

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

* Work partially supported by the Italian MURST under the Interdata Project.

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 [5], of data coming from heterogeneous information sources [24] 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.

Currently, the research activity concerning the management of semi-structured data is moving along three directions [14]: (1) techniques for gathering all kinds of information (HTML pages, images, multimedia documents and so on) from various information sources (like the Web) and for extracting structural information from them [3, 19]; (2) development of data models able to represent such kinds of information and extension of traditional database techniques to manage them; (3) development of query execution techniques able to exploit the structural information extracted from data and of techniques to export data on the Web [2].

In the data model area the research community has proposed two main approaches to model semi-structured data [10, 14]. 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 [13, 26]. 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 [4, 12] 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. To overcome the drawbacks of the lack of "schema" information, proposals following the second approach have been recently extended with the introduction of some flexible "schema mechanism" [11, 17], able to represent information on the data, and yet leaving a high degree of freedom with respect to the data entered into the database.

An important issue, quite independent from the modeling approach adopted, is to capture the existence of some regularity in the data, i.e., typing data or, as in our approach, classifying them. Automatic typing or classification is crucial in order to achieve effective storage and retrieval. However, limited work has been carried out in the context of semi-structured data. In this paper, we address such issue by defining a classification approach for data, whose structure is not known, with respect to classes in an inheritance hierarchy of an object-oriented database. Therefore we assume the existence of an a-priori defined schema and allow one to create objects, whose class is not known, which are automatically classified in the existing schema. Once an object has been classified, it can be effectively considered part of the database. Applications can, therefore, access and modify such an object exploiting all the database features.

It is important to remark that the problem of automatically classifying information has also been investigated in other areas. However, semi-structured data have features requiring the development of specific classification techniques. In particular, the problem of automatic classification has been dealt with in the context of frame-based terminological languages [25], which use automatic classification techniques both at the terminological and assertional levels (schema and data, respectively), for correctly positioning a concept in the taxonomy and for determining the most specific concept for an instance. Classification approaches, typical of such languages, rely on determining subsumption relations among concepts. Subsumption, however, takes into account also attribute values, rather than only considering the similarity of the structure, as in our approach, which then results in simpler and more efficient algorithms. The problem has also been investigated in the software engineering area. In [7] a CASE tool is proposed that starting from a set of object examples derives a schema suited for handling those objects.

In the context of semi-structured data the problem of automatic typing has been addressed in [23]. However, the goal of that work substantially differs from ours, since their main aim is to extract schema information from data, that is, to extract structure from raw data. They deal with the problem of how to avoid the proliferation of types by defining a distance among types, but they do not address how their framework could exploit some a-priori knowledge on the data schema. We remark that 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 [21]). Moreover, in [23] it is not specified whether the insertion of new objects, once the schema is set, can result in schema modifications and attribute domains are not kept into account.

Our classification approach has been proposed in the context of a reference data model [8]. The reference 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 [4, 13, 26], in that it captures all the kinds of data heterogeneity that can be represented in those models. Moreover, we remark that, though tailored to a given data model, our approach to automatic classification is highly independent from the particular type system and it can be easily adapted to other object-oriented data models and type systems supporting union types. In fact, union types represent a common extension to a traditional object type system to meet the flexibility requirements for managing semi-structured data.

In our model, a *semi-structured object* is an object that has been created without specifying the class it belongs to. To this purpose, our model supports a new operation in which the class to which the object belongs to may not be specified.

In the context of semi-structured data, the assumption that for each object there is a type exactly describing it is 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. If an object is a weak member of no class, it is inserted in a repository of unclassified objects. As the schema evolves the repository is periodically examined, trying to classify objects contained in it.

As it is outlined in [23], addressing the problem of extracting structure from semi-structured data leads to approximate typing or classification, since heuristic techniques are exploited. The conformity and heterogeneity degrees are measures that allow one to classify a given object in the schema, inserting it into a class which is as close as possible to the actual object structure.

Among the possible applications of our classification technique, we would like to mention its use in supporting content-based search on the Web. The idea is to record the content of HTML pages in an existing database. Through an information extraction tool one can delimit a (semi-structured) object representing the relevant information of the page. Then, our automatic classification tool determines the most appropriate class to insert the object in. Once the object, corresponding to the HTML page, is inserted in the database, such information can be used to support content-based data retrieval through a query language. We believe that such an approach could represent a relevant improvement to the well-known techniques, based on pattern matching, adopted by the most popular Web search engines.

The remainder of the paper is organized as follows. In Section 2 we review the concepts of the reference data model that are relevant to this work. In Section 3 we introduce the notion of weak membership, whereas in Section 4 we discuss the proposed classification approach and we present an algorithm to automatically classify objects according to our notion of weak membership. Finally, Section 5 concludes the paper and discusses future work. Appendix A presents the formal definitions of some concepts introduced throughout the paper.

---

[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 Reference Data Model

The *reference data model*, defined as extension of the *basic object-oriented data model* [18], 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 (we refer the reader to [8] for details on the reference data model). The `spring` and union types enrich the original object-oriented data model with the flexibility required to manage semi-structured data, and make the type system of our model more flexible than those of existing data models for semi-structured data [13, 26]. In order to provide a safe object-oriented data model, in [8] subtyping relationship and class refinement are addressed.

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.

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. Our union type definition is similar to the one proposed in [13], but it is not identical since we impose the restriction that the types of the union type components be neither the `spring` type nor union types. This restriction ensures efficiency in terms of space allocation and type safety, and simplifies classification of semi-structured objects. Subtyping rules for union types are similar to those proposed in [13]. 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, we have modified the record type definition of the basic object-oriented data model 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, we only need to use the label appearing in the union type definition.

Example 1. Let `person` be a class name. Let `record-of(a:integer, union-of (b:string, c:person))` be a record type. Let X be a variable of this type. In order to access component `b`, we simply write X.`b`. $\diamond$

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(a:integer, union-of(a:string,c:person))`.

A legal value, for a record type, has the form $(a_1 : v_1, \dots, a_n : v_n)$, 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_p be the identifier of an object belonging to the class `person`, then $(a : 5, b :\text{'}rose\text{'})$ and $(a : 8, c : i_p)$ 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 [9]. 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, Objects and Semi-structured Objects

Our model supports a quite standard notion of class, with some differences arising from the introduction of the union and `spring` types. Each class, moreover, 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*

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

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

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 (component $m+1$ 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-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-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 people, and in particular name, age, birthday and love, where name may be a `string`, or a `record` with two components, first name (`f-name`), surname (`s-name`), and love may assume any value (a person may love another person or an animal or anything else). Let `date` be a class of the database schema and i_d be the identifier of an object instance of class `date`. We may define a class `person` whose structural type is:

```
record-of(union-of(nameS:string, nameR:record-of(f-name:string,
        s-name:string)), age:integer, birthday:date,love:spring).
```

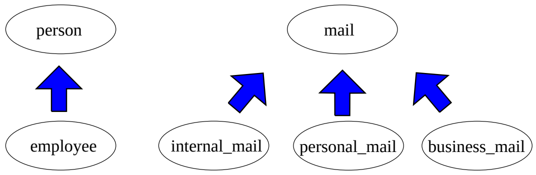


Fig. 1. A chunk of a database schema

The following objects are instances of class `person`: $o_1 = (i_1, v_1, \texttt{person})$, where $v_1 = (\texttt{nameS} :'X', \texttt{age} : 25, \texttt{birthday} : i_d, \texttt{love} :'rose')$, and $o_2 = (i_2, v_2, \texttt{person})$, where $v_2 = (\texttt{nameR}:(\texttt{f-name} :'Max', \texttt{s-name} : 'X'), \texttt{age} : 25, \texttt{birthday} : i_d, \texttt{love} : i_p)$. Note that, even if their states are legal values for the structural type of class *person*, they have different structures. ◇

In our model, finally, we denote by **semi-structured object** an object created without specifying the class it belongs to. The object is called semi-structured because it is inserted in the database without any a-priori information about its class. The object can be an instance of several classes of the schema, or of no class. In other words, the object state may be a legal value for the structural type of more than one class, or it may be a legal value for no structural type associated with any class in the schema.

Example 3. Suppose we want to classify e-mails in a database in order to make their retrieval easier. Suppose we have a mailer that allows to associate some structural information with e-mails and that we have an extraction information tool able to take that structural information out from them. Now we want to create a database to store e-mails using our model. The idea is that when a new e-mail is sent or received through the mailer, the extraction information tool takes out information (with a structure) from the e-mail and tries to insert this semi-structured object in the database. Suppose we have created the following classes in the database:[3]

- *stype*(`mail`) =
  `record-of(union-of(receiverS:string, receiverP:person), body:spring);`
- *stype*(`internal_mail`) =
  `record-of(subject:spring, sender:person, union-of(receiverS: string, receiverP:employee), body:string);`
- *stype*(`personal_mail`) =
  `record-of(subject:person, sender:person, union-of(receiverS: string, receiverP:person), body:string);`
- *stype*(`business_mail`) =

[3] We do not present classes `person` and `employee` and the other classes of the schema because they are irrelevant for the example.

```
record-of(logo:string, sender:person, union-of(receiverS:
string, receiverP:employee), body:string).
```

The meaning of the previous classes is intuitive. The idea is to have a class for e-mails and to refine the personal, internal and business e-mails in distinct subclasses. Figure 1 shows a chunk of the database schema, showing the inheritance relationships among classes. Suppose the extraction information tool generates objects o_1, ..., o_4 from some e-mails arrived in the mailbox, whose states are, respectively:

1. (`receiverS`:'*Elena F ...*', `body`:'*Dear Monica ...*'),
2. (`subject`:i_p, `sender`:i_p),
3. (`sender`:i_p, `receiverP`: i_e, `body`:'*Hello Ugo ...*'),
4. (`subject`:'*Summer in ...*', `attachment`:'*photo.jpg*')

where i_p is an identifier of an object of class `person` and i_e is an identifier of an object of class `employee`. The first object is an instance of class `mail`, whereas the others have less attributes or, as in the last case, has an attribute not in the schema. In the following we will see how our classification approach handles these situations. $\diamond$

3 Weak Membership

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,[4] 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. In order to classify a semi-structured object, we need a criterion to bind such object to a class. The notion of instance could be used for this purpose, but its definition is too restrictive to be used for semi-structured objects. In order to achieve the flexibility needed to classify semi-structured objects, we propose a weaker notion, referred to as *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. The idea of classifying semi-structured data in an existing a-priori database schema could seem too restrictive. By contrast we believe that our automatic classification, based on the notion of weak membership, represents a compromise between the flexibility of semi-structured data and the rigidity of object-oriented schemas and allows one to benefit from all the features of object-oriented database systems to manage this kind of information.

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

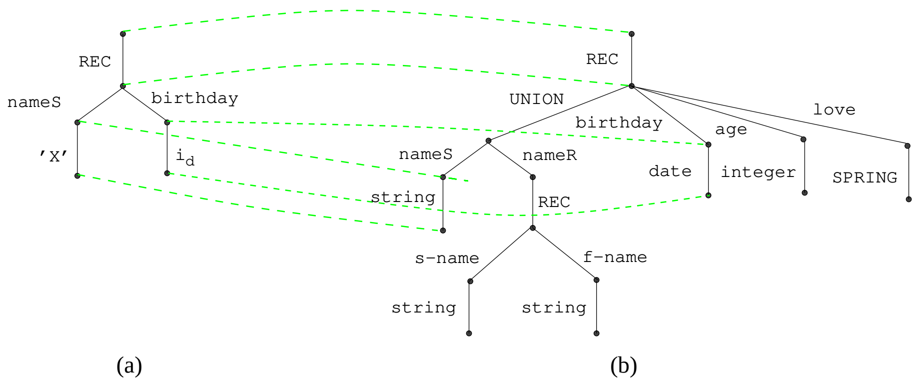


Fig. 2. (a) Object value expression, (b) class structural expression and their simulation relation

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 [22]. 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 $\mathcal{LT} = \{\text{LIST, REC, SET, UNION, SPRING}\} \cup \mathcal{AN} \cup \mathcal{PRED}$. The following definition states the notion of *class structural expression*.

Definition 4. (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 $\varphi_t : E_t \to \mathcal{LT}$ is the edge labeling function.* □

Since the class structural expression is a tree associated with a type of the model (the structural type of a class), we have developed an inductive system to map any type of the model into a labeled tree [8]. Figure 2(b) shows the class structural expression associated with class `person` of Example 2. Note that `string`, `date`, and `integer` symbols are predicates which represent the set of legal values for the corresponding types. Note also that we have not generated the structural expression associated with the object type `date` since we are interested in shallow[5] comparison among objects and classes.

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

Definition 5. (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, $\varphi_v : E_v \rightarrow \mathcal{LV}$ is the edge labeling function.* □

Similarly to what has been done for the class structural expression, an inductive system has been defined in [8] to map values of the model into labeled trees. Figure 2(a) shows the object value expression associated with a semi-structured object whose state is ($\texttt{nameS} :'X', \texttt{birthday} : i_d$). According to our shallow approach, we have not generated the object value expression associated with the state of the object identified by i_d.

We also introduce the notion of refinement among structural expressions, which is used by our classification algorithm. Intuitively, a structural expression (that is, a tree whose edges are labeled in $\mathcal{LT}$) ε' is a refinement of a structural expression ε if the two trees are isomorphic, but some of the labels of ε' are class names corresponding to subclasses of the corresponding labels in ε. Let $\mathcal{PRED}^o$ denote the subset of $\mathcal{PRED}$ corresponding to object types, and $\leq_{ISA}$ denote the inheritance relationship on object types, then, the notion of refinement among structural expressions is defined as follows.

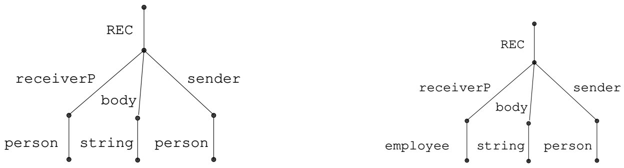
Definition 6. (Structural expression refinement). *Let $\varepsilon = (V_t, E_t, \varphi_t)$ and $\varepsilon' = (V_t, E_t, \varphi'_t)$ be two structural expressions. ε' is a refinement of ε if*

$$\forall e \in E_t : \varphi_t(e) = \varphi'_t(e) \vee (\varphi_t(e) \in \mathcal{PRED}^o \wedge \varphi'_t(e) \in \mathcal{PRED}^o \wedge \varphi'_t(e) \leq_{ISA} \varphi_t(e)). \;\square$$

The following example illustrates the notion of structural expression refinement.

Example 4. The structural expression presented in Figure 3(b) is a refinement of the one presented in Figure 3(a) because the two trees have the same structure and the labels are all equals except `person` and `employee` which are in the $\leq_{ISA}$ relation. ◇

[5] *Shallow* is used here with the same meaning as in shallow equality [16].



(a) (b)

Fig. 3. Structural expression refinement

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 $\approx_{\mathcal{L}}$ between labels). *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 the notion of *simulation*. The simulation is a particular relation among the vertices of the object value expression and the vertices of the class structural expression that takes into account the symbols used to label the edges of these expressions. The idea of simulation is used in several research areas [11, 20] and it has a solid theoretical foundation. We will use it to formally define the notion of weak membership. Our definition of simulation in an extension of the "classical" one, thus it preserves its good properties [20].

Informally, a relation $\mathcal{R}$ between the vertices of an object value expression (V_v, E_v, φ_v) and the vertices of a class structural expression (V_t, E_t, φ_t) is a simulation if the following conditions hold:

– If the label $l_v \in \mathcal{LV}$ associated with the edge $(u_1, u_1') \in E_v$, outgoing from vertex u_1, identifies a particular type (structural, basic, object), then an edge $(u_2, u_2') \in E_t$ must outgo from u_2 labeled with a symbol $l_t \in \mathcal{LT}$, for which relation $\approx_{\mathcal{L}}$ holds between l_v and l_t. Moreover, relation $\mathcal{R}$ must hold between u_1' and u_2'.
– If the label $l_v \in \mathcal{LV}$ associated with the edge $(u_1, u_1') \in E_v$, outgoing from vertex u_1, is an attribute name ($l_v \in \mathcal{AN}$) and the label associated with the edge $(u_2, u_2') \in E_t$, outgoing from vertex u_2, is $UNION$, then an edge $(u_2', u_2'') \in E_t$ must exist, outgoing from vertex u_2', with the same label of the edge (u_1, u_1'). Moreover, relation $\mathcal{R}$ must hold between u_1' and u_2''.

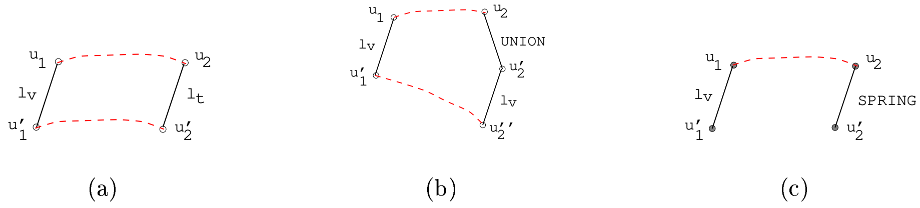


Fig. 4. Visual representation of relation among vertices of item (2) of Definition 8

- If the label $l_t \in \mathcal{LT}$ associated with the edge $(u_2, u_2') \in E_t$, outgoing from vertex u_2, is $SPRING$, then there is no condition to verify. In this situation we do not need to check other pairs in the relation whose first component is a vertex belonging to the subtree rooted at u_1, since the value associated with the subtree rooted at u_1 surely is a legal value for the type associated with the subtree rooted at u_2 (that is, the `spring` type).

The following definition formally states our notion of simulation. In the definition *root(A)* denotes the root of tree A and $u \xrightarrow{l} u'$ denotes an edge (u, u') such that $\varphi((u, u')) = l$.

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 2 the dashed lines represent the simulation between the object value expression associated with the semi-structured object, that we have introduced previously, and the structural expression associated with the structural type of class `person` of Example 2. A visual representation of relation among vertices of item (2) of Definition 8 is shown in Figure 4. The dashed lines identify the relation that must hold between the vertices of the two trees. Note that, as you can see in Figure 4(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. Consider the following example.

Example 5. 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 the above example, however, does not capture our notion of the set of legal values for the record type in the example. 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 $\forall u_2 \overset{UNION}{\longrightarrow} u_2'$ 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 $\varepsilon_t(c)\}$.* □

The above definition of membership is more flexible than the notion of instantiation. According to such definition, an object state can contain less components than those present in the structural type of a class. Such definition, however, does not allow one to identify only one class to which the object belongs. In the next section we propose an approach to establish the most appropriate class to which the object belongs.

4 Automatic Classification Approach

In the previous section we have proposed an approach to determine whether a semi-structured object is a weak member of a class. An object may be a weak member of several classes.

Example 6. Consider an object whose state consists only of the component (age: 25). Such object is a weak member of all the subclasses of class `person` in the schema of Example 2. ◇

When an object is a weak member of several classes, we need some measures to determine the most appropriate class in which we can classify the object. 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. Finally, we compute the algorithm complexity and present some examples of automatic classification.

4.1 Conformity Degree

With the first measure, referred to as *conformity degree*, we want to check how much the type of the semi-structured object is close to the structural type of a given class. In other words, we check 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. For example, if an object is a weak member of a class and it is a weak member of some subclasses of that class, we are not interested in classifying the object in the most specific class of the inheritance hierarchy if this class has several attributes which are not part of the object. 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. Informally, 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 Appendix A. Figure 5(a) shows the object structural expression associated with the object value expression shown in Figure 2(a). As we can see, this object structural expression represents the type `record-of(nameS:string, birthday:date)`. The value associated with the object value expression shown in Figure 2(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, formally defined in Appendix A, 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 5(b) shows the tree only containing the real paths of the class structural expression shown in Figure 2(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*

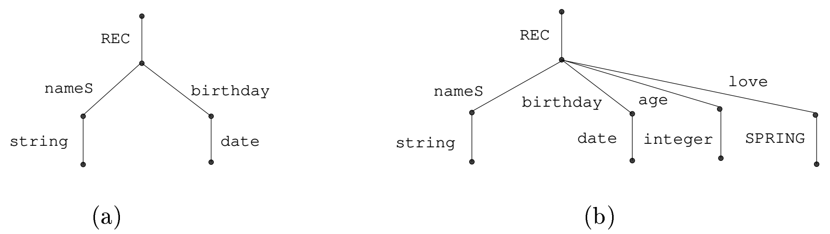


Fig. 5. (a) Object structural expression, and (b) the part of class `person` structural expression containing only the union components that appear in the object state

expression. Formally:

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

□

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 4, thus the conformity degree is 0.5.

The following proposition (proved in [8]) holds.

Proposition 1. *The following results on conformity degree and weak membership hold:*

- *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.*
- *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. In Section 2 we have also seen that for each class in the schema the set of objects belonging to a class may have different structures. For example, if in the structural type of a class there is only a union type with two components, and there is no component of `spring` type, then the extension of this class consists of a set with two kinds of objects: the ones having the first component of the union type, and the ones having the second component of the union type. Thus the heterogeneity degree is 2.[6] By contrast, if we have only a component of `spring` type in the structural type of the class, this component may assume any legal value of any type in the schema. Thus, the structure of objects belonging to this class may be highly heterogeneous. In such case, the heterogeneity degree is evaluated as the number of all value and object types introduced in the schema (these sets are 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 types. The heterogeneity degree of other types (belonging to the basic type system) is 1 since they do not generate heterogeneous extensions. In computing the heterogeneity degree we take into account that we perform a shallow comparison among the class structural expression and the object value expression. That is, if the type of an attribute in a class c is an object type $\bar{c}$, in calculating the heterogeneity degree we do not take into account the heterogeneity degree associated with the class $\bar{c}$, rather we state that its heterogeneity degree is 1. 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.*

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

□

Note that the heterogeneity degree, like the class structural expression, does not depend on the database instances but only on the schema. Thus, the heterogeneity degree and the class structural expression may be computed at schema definition time. This is important in order to define efficient algorithms to classify objects in the schema.

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. In addition, for classes with the same

[6] Note that, since the types of union type components are constrained to belong to the basic type system, their heterogeneity degree is always 1.

conformity and heterogeneity degrees, we take into account the inheritance hierarchy, by choosing the most specific class in the hierarchy.

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. Otherwise,
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,
4. We delete from $WMS_{H\text{-}min}$ all the classes having a subclass in that set, and any class c such that $WMS_{H\text{-}min}$ contains a class c' whose object structural expression[7] is a refinement of the object structural expression of c, according to Definition 6. If the resulting set is a singleton, then 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, first of all we find out the set of classes having the highest conformity degree from the classes which the object is a weak member of. 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. To select a class among the remaining classes, we choose those classes which are most specific in the inheritance hierarchy as well as those classes whose attribute domains most closely matches the attribute values of the object.

Note that, if the resulting set of the algorithm is not a singleton then an arbitrary class is selected in which the object is inserted. An alternative approach, which however is left for future investigation, would be to classify the object in all the classes in that set.

4.4 Complexity of the Classification Algorithm

In this section we present the complexity of our algorithm. The following notation is used:

- C is the set of classes of the schema with respect to the object is being classified;

[7] We recall that the object structural expression (formally defined in Appendix A) is the structural expression representing the portion of the class structural expression actually present in the object.

– k the number of classes in C;
– $dim(o)$ is the dimension of the object value expression associated with the object being to classified, that is, the number of vertices of the tree;
– $dim(c)$ is the dimension of the class structural expression associated with a class $c \in C$, that is, the number of vertices of the tree.[8]

The first step of the algorithm is the computation of the set of classes of which the object is a weak member. According to Definition 9, this is equivalent to determine whether a simulation exists between the object value expression and the class structural expression. In the computation of the simulation relationship (Definition 8), at each step, for each edge $u_1 \xrightarrow{l_v} u'_1$ of the object value expression we check whether an edge $u_2 \xrightarrow{l_t} u'_2$ exists such that one of the conditions of Definition 8 holds. Note that, since the time required to check whether one of the conditions of Definition 8 holds is constant, we have to compute how many times this step is iterated. Since, for both the object value expression and the class structural expression, the outgoing edges from a given vertex having an attribute name as label are distinct, no backtracking is needed during the iteration process, which, at most, repeats the check of the properties as many times as the number of edges of the object value expression. Since the number of edges in a tree is equal to the number of the vertices minus one, we can conclude that the first step has a cost in $O(dim(o))$ for each class, that is, in $O(k{*}dim(o))$ for all the classes of the schema. In the second step, first the conformity degree is computed for each class of which the object is a weak member. Because the number of such classes is at most k, the third step has a cost in $O(k * max_{c \in C}\{dim(c)\})$. After that, the set of classes with respect to which the object has the highest conformity degree, $WMS_{C\text{-}max}$, is computed. This step has a cost in $O(k)$. In the third step, the set of classes with respect to which the object has the lowest heterogeneity degree, among the ones in $WMS_{C\text{-}max}$, is computed. Such step has a cost in $O(k)$. In fact, the heterogeneity degree is an information which can be associated with a class when it is created, without overhead for the algorithm. Finally, in the fourth step, we compare all the remaining classes testing for inheritance and refinement. Supposing testing for subclassing constant [6], the step can be executed in $O(k * max_{c \in C}\{dim(c)\})$.

Therefore, $O(k * max\{dim(o),\ max_{c \in C}\{dim(c)\}\})$ is the total cost of the algorithm. Thus, the classification algorithm solves the problem in a time that is linear in the dimension of the entities involved in the classification process.

4.5 An Example of Classification

In this section we present some examples of the application of our classification algorithm. We classify the semi-structured objects that we have presented in Example 3. Note that the other components of the schema are not relevant for

[8] Note that given an object o and a class c the object structural expression ($\varepsilon(o, c)$) is a subtree of the tree associated with the structural type of a class of which the object is a weak member, thus $dim(c)$ is an upper-bound of the dimension of $\varepsilon(o, c)$.

the example, we only need to know that the number of different (value or object) types defined in the schema is 20. Based on the number of different types in the schema, we can compute the heterogeneity degree of the classes presented in Example 3.

- $H^{\circ}(stype(\texttt{mail})) = 20 * 2 * 1 = 40$,
- $H^{\circ}(stype(\texttt{internal\_mail})) = 20 * 1 * 2 * 1 = 40$,
- $H^{\circ}(stype(\texttt{personal\_mail})) = 1 * 1 * 2 * 1 = 2$,
- $H^{\circ}(stype(\texttt{business\_mail})) = 1 * 1 * 2 * 1 = 2$.

When the algorithm presented in Section 4.3 is applied to object o_1, it computes, in step 1, the set $WMS = \{$`mail`, `internal_mail`, `personal_mail`, `business_mail`$\}$. Each attribute in the object state, indeed, is an attribute in the structural type of each class of the set WMS and the values are of the correct types. Since $WMS \neq \emptyset$ the second step of the algorithm is applied and the set $WMS_{C\text{-}max} = \{$`mail`$\}$ is determined. The classes `internal_mail` and `personal_mail` have been removed from the set $WMS_{C\text{-}max}$ because they have an attribute, `sender`, not in the object state, whereas the class `business_mail` has been deleted since it has two attributes, `sender` and `logo`, not in the object state. At this point, since the set WMS_{C-max} is a singleton, the object is classified in class `mail`.

When the algorithm is applied to classify object o_2, it computes, in step 1, the set $WMS = \{$`internal_mail`, `personal_mail`$\}$. It does not consider classes `mail` and `business_mail` because they do not contain the `subject` attribute. Then, the set of classes with the highest conformity degree ($WMS_{C\text{-}max}$) is determined, but this set is equal to the previous one (the two classes have the same attributes). Therefore, the set of classes with the lowest heterogeneity degree $WMS_{H\text{-}min} = \{$`personal_mail`$\}$ is computed. Thus, the object is classified in class `personal_mail`.

When the algorithm is applied to classify object o_3, it computes, in step 1, the set $WMS = \{$`internal_mail`, `personal_mail`, `business_mail`$\}$. It does not consider class `mail` because it does not contain the `sender` attribute. Then, the set of classes with the highest conformity degree ($WMS_{C\text{-}max}$) is determined, but this set is equal to the previous one (the three classes have the same number of additional attributes). Therefore, the set of classes with the lowest heterogeneity degree $WMS_{H\text{-}min} = \{$`personal_mail`, `business_mail`$\}$ is computed. Class `personal_mail` is not a subclass of `business_mail`, but if we consider the object structural expressions associated with object o_3[9] we find out that `business_mail` is a refinement of `personal_mail`, thus the object is classified in class `business_mail`.

When the algorithm is applied to classify object o_4, it determines that the object is not weak member of any class. Thus, the object is put in the repository of unclassified objects.

[9] The object structural expression associated with o_3 with respect to `personal_mail` and `business_mail` is shown in Figure 3.

5 Conclusions

In this paper we have proposed an approach to classify objects whose structure is not necessarily a type present in the database schema. The proposed technique, which is currently being implemented, is based on the notion of weak membership and on conformity and heterogeneity degrees, and allows one to automatically classify an object in the class whose structural type best fits the object state.

Our classification approach is totally based on the object structure. An alternative approach could be to classify objects according to their response to messages that they receive. We did not investigate such an approach because it is more related to object-oriented programming languages than to databases. In object-oriented databases the structure, rather than the behavior, is regarded as the most relevant information associated with objects. However, such an approach could still represent an interesting research direction.

Another interesting research direction is the development of suitable information extraction tools. Two approaches are possible. The first one is to define "*prototype documents*" with respect to which the documents are compared. The comparison of a given document against a prototype document allows one to infer structural information from the document. The second approach is to use decision trees with rules that specify conditions on the attribute types. A path in the decision tree may thus represent a particular type to which a set of objects may belongs to. We plan to investigate those approaches as future work.

We are extending the work presented in this paper along several other directions. First, we would like to consider the possibility 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 conformity and 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 [15]. In this way we can achieve a more accurate classification. 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. We would like to consider the possibility that the schema evolves, as a consequence of object classification. 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, and then to verify whether a simulation exists between the tree associated with the query and the tree associated with the HTML page. If the simulation exists then the HTML page is a possible answer for the query. Finally, we plan to investigate how semi-structured objects can be handled by application programs and queried, in the context of semi-structured data, by revisiting type checking notions.

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, S. Cluet, and T. Milo. Correspondence and Traslation for Heterogeneous Data. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 351–363, 1997.
3. S. Abiteboul, R. Motwani, and S. Nestorov. Inferring Structure in Semistructured Data. *In Proc. Workshop on Management of Semistructured Data, SIGMOD Record*, 26(4):39–43, 1997.
4. 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.
5. 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.
6. R. Agrawal, A. Borgida, and H. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 253–262, 1989.
7. P. L. Bergstein and K. J. Lieberherr. Incremental Class Dictionary Learning and Optimization. In P. America, editor, *Proc. Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, pages 377–396, 1991.
8. E. Bertino, G. Guerrini, I. Merlo, and M. Mesiti. An Object-Oriented Data Model for Semi-Structured Data. Technical Report DISI-TR-99-06, University of Genova, Department of Computer Science (DISI), 1998.
9. 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.
10. P. Buneman. Semistructured Data. In *Proc. of 6th ACM SIGACT-SIGMOD-SIGART Symposium on PODS*, pages 117–121, 1997. Tutorial.
11. 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.
12. 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.
13. 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.
14. S. Cluet. Modeling and Querying Semi-Structured Data. In M. T. Pazienza, editor, *Information Extraction. LNAI 1299*, pages 192–213, 1997.
15. O. Deux et al. The Story of o_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.
16. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
17. R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. Twentythird Int'l Conf. on Very Large Data Bases*, pages 436–445, 1997.

18. 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.
19. 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.
20. M. Henzinger, T. Henzinger, and P. Kopke. Computing Simulation on Finite and Infinite Graphs. In *Proc. of 20th Symposium on Foundations on Computer Science*, pages 453–462, 1995.
21. S. Holzner. *XML Complete*. McGraw-Hill, 1998.
22. R. Milner. An Algebraic Definition of Simulation between Programs. In *Proc. of the 2nd IJCAI*, pages 481–489, London, UK, 1971.
23. 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.
24. 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.
25. C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK System Revisited. Technical Report KIT - Report 75, Technische Universitat Berlin, 1989.
26. F. Rabitti. *The Multos Document Model*, volume Human Factors in Information Technology of *6*, chapter 3, pages 17–52. North-Holland, 1990.

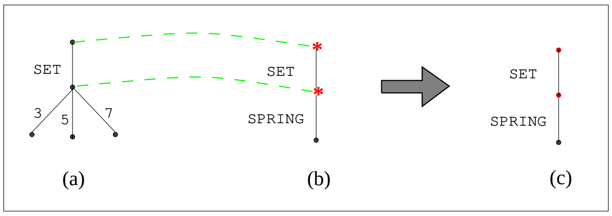


Fig. 6. Presence of $SPRING$ label in a part of class structural expression

A Additional Formal Definitions

In this section we present the formal definition of object structural expression and two figures that illustrate two particular cases, that is, the `spring` type and the null value, discussed in Section 4. The figures illustrate how to obtain, starting from the object value expression (a) and the class structural expression (b), the object structural expression (c). In the following definition, given a tree A and a vertex u, we denote by $vertex(A)$ the set of vertices of A and with $tree(u, A)$ the subtree of A rooted at u.

Definition 12. *(*Object structural expression*). Let o be a semi-structured object and c be a class in the schema such that o is a weak member of c, that is, a simulation $\mathcal{R}$ exists between $\varepsilon_v(o) = (V_v, E_v, \varphi_v)$ and $\varepsilon_t(c) = (V_t, E_t, \varphi_t)$, then we define* object structural expression*, denoted by $\varepsilon(o, c)$, the following tree:*

$$(V, E, \varphi)$$

where:

- $V = \overline{V} \cup \overline{\overline{V}} \cup \overline{\overline{\overline{V}}}$
  - $\overline{V} = \{u \mid (u_1, u) \in \mathcal{R}\}$,
  - $\overline{\overline{V}} = \{u \mid (u_1, u_2) \in \mathcal{R},\ (u_2, u) \in E_t \text{ and } \varphi_t((u_2, u)) = SPRING\}$
  - $\overline{\overline{\overline{V}}} = \bigcup_{(u_1,u_1') \in E_v \ s.t.\ \varphi_v((u_1,u_1'))=NULL} \{u \mid (u_1', u_2') \in \mathcal{R} \text{ and } u \in vertex(tree(u_2', \varepsilon_t(c)))\}$
- $E = \overline{E} \cup \overline{\overline{E}}$
  - $\overline{E} = \{(u_1, u_2) \mid u_1, u_2 \in V, (u_1, u_2) \in E_t\}$,
  - $\overline{\overline{E}} = \{(u_1, u_2) \mid u_1, u_2 \in V \text{ and } \exists u \ s.t.\ (u_1, u), (u, u_2) \in E_t, \varphi_t((u_1, u)) = UNION)\}$
  - $\varphi((u_1, u_2)) = \begin{cases} \varphi_t((u_1, u_2)) \ \text{if } (u_1, u_2) \in \overline{E} \\ \varphi_t((u, u_2)) \ \ \text{if } (u_1, u_2) \in \overline{\overline{E}} \text{ and } (u, u_2) \in E_t \end{cases}$

□

We introduce the following definitions to formally state the concept of real path.

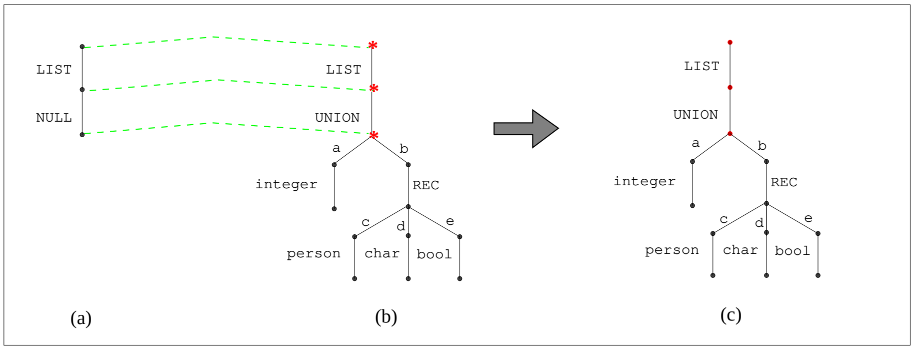


Fig. 7. Presence of $NULL$ label in a part of object value expression

Definition 13. (Labeled path). *Let $A = (V, E, \varphi)$ be a labeled tree on $\mathcal{LT}$, the sequence:*

$$u_1.l_1.u_2.l_2.\cdots.l_{n-1}.u_n$$

is a labeled path, where:

- $u_1, \ldots, u_n \in V$,
- $\forall i,\ 1 \leq i \leq n-1,\ (u_i, u_{i+1}) \in E$ *and* $\varphi((u_i, u_{i+1})) = l_i$. □

Definition 14. (Maximal labeled path). *Let $A = (V, E, \varphi)$ be a labeled tree on $\mathcal{LT}$, a labeled path*

$$u_1.l_1.u_2.l_2.\cdots.l_{n-1}.u_n$$

is called maximal if $u_1 = root(A)$ and u_n is a leaf of A. □

Now we are able to define the notion of real paths of a tree. In the following definition we denote by $path(A)$ the set of maximal labeled paths of the tree A and with $\pi_2(\mathcal{R}) = \{u \mid (\bar{u}, u) \in \mathcal{R}\}$.

Definition 15. (Real paths). *Let o be a semi-structured object and c be a class in the schema such that o is a weak member of c, that is, a simulation $\mathcal{R}$ exists between $\varepsilon_v(o)$ and $\varepsilon_t(c)$, then we define real-path the following set:*

$$\begin{aligned} &real\text{-}path(\varepsilon_t(c)) = \\ &\quad \{\omega \mid \omega \in path(\varepsilon_t(c)) \text{ and } (\omega = \ldots REC.u_1.UNION.u_2.l.u_3\ldots \Rightarrow \\ &\quad (u_3 \in \pi_2(\mathcal{R})) \text{ or } (\{u \mid (u_2, u) \text{ is an edge of } \varepsilon_t(c) \text{ and } u \in \pi_2(\mathcal{R})\} = \emptyset))\} \end{aligned}$$

□