Paper Number 255

Extracting Schema from Semistructured Data

SVETLOZAR NESTOROV*
Stanford University
evtimov@db.stanford.edu

SERGE ABITEBOUL†
INRIA-Rocquencourt
Serge.Abiteboul@inria.fr

RAJEEV MOTWANI‡
Stanford University
rajeev@cs.stanford.edu

Abstract

Of late, especially with the advent of the world-wide web, there has been an increasing interest in semistructured data in the database community. Semistructured data is characterized by the lack of any fixed and rigid schema, although typically the data has some implicit structure. While the lack of fixed schema makes extracting semistructured data fairly easy and an attractive goal, presenting and querying such data is greatly impaired. Thus, a critical problem is the discovery of the structure implicit in semistructured data and, subsequently, the recasting of the raw data in terms of this structure.

In this paper, we consider a very general form of semistructured data based on labeled, directed graphs. We show that such data can be typed using the greatest fixpoint semantics of monadic datalog programs. We present an algorithm for approximate typing of semistructured data. We establish that the general problem of finding an optimal such typing is NP-hard, but present some heuristics and techniques based on clustering that allow efficient and near-optimal treatment of the problem. We also present some preliminary experimental results.

1 Introduction

Data found over the network is generally fairly irregular. For instance, the home-pages of members of a group may contain some similar information (e.g., name, email, address, photo) but some of these may be missing in particular pages, and extra information may be present in others. For another example, consider cartographical data servers. These typically have thousands of records with hundreds of properties, most of which are null for any given object. Indeed, irregularities are often the norm in data found over the network. Furthermore, they arise naturally when one integrates data originating from several distinct (structured) sources that provide information about a common set of entities but represent these entities differently. Recently, the term semistructured data [1, 6] has emerged to describe data that has some structure but which is neither regular, nor known a-priori to the system. It is precisely for this reason that most semistructured

*Supported by the Community Management Staff's Massive Digital Data Systems Program, NSF grant IRI–96–31952, ARO grant DAAH04–95–1–0192, and grants of IBM and Hitachi Corp.

†Work performed in part while the author was visiting Stanford CS department.

‡Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership Award, an ARO MURI Grant DAAH04-96-1-0007, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

data models are self-describing. We will employ a fairly standard model for semistructured data that is based on a labeled directed graph [16, 8].

Database systems come equipped with nice graphical interfaces and efficient access methods. Both features are based primarily on the existence of some regularity in the data, i.e., on a typing of the data. Query formulation is facilitated by QBE-style [19] interfaces that, by using existing structure, allow users and application programs to learn about the data set and access data more conveniently. Also, performance is greatly improved by taking advantage of the existing structure, e.g., via indexes [18]. Thus, although we often have to deal with data sets found on the network that have no explicit structure and are fairly irregular, we would very much prefer to work with regular, structured data. This is the prime motivation of the present work on the extraction of implicit structure in semistructured data.

Clearly, the implicit structure in a particular data set may be of varying regularity. Indeed, we should not expect in general to be able to perfectly type a data set. The size of a *perfect* typing (a notion that we will study) may be quite large, e.g., be roughly of the order of the size of the data set, which would prohibit its use for query optimization and render it impractical for graphical query interfaces. Thus, we consider *approximate* typings, i.e., an object does not have to fit its type definition precisely. We study the trade-off between the quality of a typing and its compactness. More precisely, the typing problem and its trade-off can be formulated as follows. Suppose we have selected a type description language and a measure for type sizes, as well as a distance function over data sets. The problem then is: given a data set I, find a typing τ and a data set J of typing τ, such that the size of τ is smaller than a certain threshold and the distance between I and J is minimized. In other words, we want to find τ which is small enough and such that I presents as few defects as possible with respect to τ. (The dual problem is the minimization of the size of τ for a given threshold on the distance between I and J.)

The first key issue is the choice of a description language for types. Our typing is inspired by the typing found in object databases [9] although it is more general since we allow objects to live in many incomparable classes, i.e., have multiple roles [3]. This aspect is also a clear departure from previously proposed typings for semistructured data [10, 15, 7, 17]. We believe (and some early experiments support this belief) that multiple roles are essential when the data is fairly irregular. We define a typing in terms of a monadic datalog program. The intensional relations correspond to object classes and the rules describe the inner structure of objects in classes. The greatest fixpoint semantics of the program defines the class extents.

We first consider the issue of computing perfect typings. We start with an obvious perfect typing that consists of having one class for each distinct object. This yields a first monadic data program. A run of this program on the data set will naturally group similar objects and thus provide a (possibly much) coarser classification of objects that yields a (possibly much) more compact perfect typing. In fact, this typing is the coarsest possible if we insist on exact fit.

This perfect typing may still be much too large unless the data is extremely regular. Therefore, we present a technique for computing an approximate typing of an appropriate size. This means that the data set is now allowed to be imperfect with respect to the typing. It may present extra information (edges that should not be present) or lack some information (missing edges). We will see that extra edges are easily handled with a greatest fixpoint approach, whereas missing edges are much more difficult to deal with. The crux of our technique is to merge similar classes so as to decrease the size of the typing. To this end, we employ a clustering algorithm [12, 11] on the classes. Intuitively, until the typing is of acceptable size (for some application dependent notion of "acceptable size"), we perform class merges that introduce a minimal error. We consider various optimization strategies to compute this approximation and issues such as the choice of the distance function.

For a concrete example, Figure 1 shows the approximate typing produced by our method for the DBG dataset consisting of various information about the members of Data Base group at Stanford. The exact notation will be explained later in the paper. For this example, it suffices to say that each label with an arrow and superscript correspond to a link to or from a type. This typing has only 6 types and provides very good summary of the actual contents of the DBG dataset. In contrast, the perfect typing for this dataset consists of 53 different types. Note that we have also given the intuitive meaning before each of the 6 types.

In contrast to our work, previous proposals on typing semistructured data [10, 15, 7] have focused on perfect typing and implicitly assumed that each object has a unique role. We have already mentioned some motivation for approximate typing and will later discuss further motivations for multiple roles.

We also present some some experimental results. The focus of our experiments is the quality of the typing results rather than than the time performance.

The rest of this paper is organized as follows. Section 2 introduces our notation and provides the intuition for choosing the specific form of typing. Section 3 gives a summary of our method for extracting the typing from the data. Section 4 deals with perfect typing and Section 5 with the issue of computing an approximate typing. Section 6 addresses recasting the original data within the approximate typing. Section 7 provides some experimental results.

2 The Typing

In this section, we present the model of semistructured data and the types that are used in the present paper. We assume some familiarity with relational databases and more particularly with the datalog query language [18, 2].

We model semistructured data in the style of [8, 16] as a labeled directed graph. The nodes in the graph represent objects and the labels on the edges convey semantic information about the relationships between objects. The sink nodes (nodes without outgoing edges) in the graph represent *atomic objects* and have values associated with them. The other nodes represent *complex objects*. In a standard manner, we represent the graph using two (base) relations defined as follows:

link(FromObj, ToObj, Label): Relation *link* contains all the edge information. Precisely, $link(o_1, o_2, \ell)$ corresponds to an edge labeled ℓ from object o_1 to o_2. Note that there may be more that one edge from o_1 to o_2, but, in our model, for a particular ℓ, there is at most one such edge labeled ℓ.

$$
\begin{array}{rcll}
\textit{project}: & \tau_1 & = & \overleftarrow{Project}^{3}, \overleftarrow{Project}^{4}, \overleftarrow{Project}^{5}, \overrightarrow{Project\_Member}^{3}, \overrightarrow{Project\_Member}^{4}, \\
 & & & \overrightarrow{Name}^{0}, \overrightarrow{Homepage}^{0} \\
\textit{publication}: & \tau_2 & = & \overleftarrow{Publication}^{3}, \overleftarrow{Publication}^{5}, \overrightarrow{Author}^{3}, \overrightarrow{Name}^{0}, \overrightarrow{Conference}^{0}, \overrightarrow{Postscript}^{0} \\
\textit{db-person}: & \tau_3 & = & \overleftarrow{Project\_Member}^{1}, \overleftarrow{Group\_Member}^{5}, \overrightarrow{Project}^{1}, \overrightarrow{Birthday}^{5}, \overrightarrow{Degree}^{6}, \\
 & & & \overrightarrow{Years\_At\_Stanford}^{0}, \overrightarrow{Email}^{0}, \overrightarrow{Home\_Page}^{0}, \overrightarrow{Title}^{0}, \\
 & & & \overrightarrow{Name}^{0}, \overrightarrow{Original\_Home}^{0}, \overrightarrow{Personal\_Interest}^{0}, \overrightarrow{Research\_Interest}^{0} \\
\textit{student}: & \tau_4 & = & \overleftarrow{Project\_Member}^{1}, \overleftarrow{Student}^{4}, \overleftarrow{Group\_Member}^{5}, \overrightarrow{Project}^{1}, \overrightarrow{Advisor}^{4}, \\
 & & & \overrightarrow{Email}^{0}, \overrightarrow{Title}^{0}, \overrightarrow{Home\_Page}^{0}, \overrightarrow{Name}^{0}, \overrightarrow{Nickname}^{0} \\
\textit{birthday}: & \tau_5 & = & \overleftarrow{Birthday}^{3}, \overrightarrow{Namex}^{0}, \overrightarrow{Month}^{0}, \overrightarrow{Day}^{0}, \overrightarrow{Year}^{0} \\
\textit{degree}: & \tau_6 & = & \overleftarrow{Degree}^{3}, \overrightarrow{Major}^{0}, \overrightarrow{School}^{0}, \overrightarrow{Name}^{0}, \overrightarrow{Year}^{0}
\end{array}
$$

Figure 1: Optimal typing program for DBG data set

atomic(Obj,Value): This relation contains all the value information. The fact $atomic(o, v)$ corresponds to object o being atomic and having value v.

We also require that (i) each atomic object has exactly one value, i.e. Obj is a key in relation *atomic*, and (ii) each atomic object has no outgoing edges, i.e., the first projections of *link* and *atomic* are disjoint.

In the following, we consider that the data comes in as an instance over *link* and *atomic* satisfying these two restrictions. We use the term *database* here for such a data set. An example of a database is given in Figure 2. Note that this data happens to be very regular.

In this paper, we consider that a typing is specified by a datalog program of a specific form (to be described shortly). The only two extensional relations (EDB's) of the typing program are *link* and *atomic*. The intensional relations (IDB's) are all monadic and correspond to the various types defined by the program. For instance, we can consider the following typing program $\mathcal{P}_0$ for the database of Figure 2:

$$
\begin{array}{lcl}
person(X) & \text{:-} & link(X, Y, \textit{is-manager-of}) \ \& \ company(Y) \ \& \\
 & & link(X, Y', name) \ \& \ atomic(Y', Z) \\
company(X) & \text{:-} & link(X, Y, \textit{is-managed-by}) \ \& \ person(Y) \ \& \\
 & & link(X, Y', name) \ \& \ atomic(Y', Z)
\end{array}
$$

The intuition is that g, j are persons, m, a are companies and the other objects are atomic.

Syntax Typing programs are more precisely defined as follows. The EDB's are *link* and *atomic*. The IDB's are all monadic. Furthermore, each IDB is defined by a single rule of the form:

$$c(X) \ \text{:-} \ A_1 \ \& \ \ldots \ \& \ A_p$$

link	FromObj	ToObj	Label
	g	*m*	is-manager-of
	j	*a*	is-manager-of
	m	*g*	is-managed-by
	a	*j*	is-managed-by
	g	*gn*	name
	j	*jn*	name
	m	*mn*	name
	a	*an*	name

atomic	Obj	Value
	gn	"Gates"
	jn	"Jobs"
	mn	"Microsoft"
	an	"Apple"

Figure 2: Data and Types

for some p, where the A_i, called the typed links, are defined as follows. Each *typed link* has one of the following forms:

1. $link(Y, X, \ell)$ & $c'(Y)$
2. $link(X, Y, \ell)$ & $c'(Y)$
3. $link(X, Y, \ell)$ & $atomic(Y, Z)$

where ℓ is some constant (a label), X is the variable in the head of the rule and Y, Z are variables not occurring in any other typed link of the rule.

We will discuss some limitations introduced by this typing further on.

Notation Suppose that the types (IDB's) of the program are $type_1, ..., type_n$. Think of the atomic objects as belonging to $type_0$. The following notation for typed links greatly simplifies our presentation and will be used throughout the paper:

- $link(Y, X, c)$ & $type_j(Y)$ is denoted by $\overleftarrow{c}^j$.
- $link(X, Y, c)$ & $type_j(Y)$ is denoted by $\overrightarrow{c}^j$.
- $link(X, Y, c)$ & $atomic(Y, Z)$ is denoted by $\overrightarrow{c}^0$.

The direction of the arrow over the label denotes whether the edge is incoming (left) or outgoing (right). The superscript denotes the type of the object at the other end of the edge.

Semantics The semantics of a datalog program $\mathcal{P}$ of the form described above for a database D is defined as the *greatest fixpoint* of $\mathcal{P}$ for D. (See, e.g., [4].) More precisely, let M be an instance over the schema of $\mathcal{P}$ such that M coincides with D on $\{link, atomic\}$. Then M is a *fixpoint of* $\mathcal{P}$ *for* D, if for each IDB c, $\mathcal{P}(M)(c) = M(c)$. It is the *greatest* fixpoint, if it contains any other fixpoint of $\mathcal{P}$ *for* D.

Note that this definition is correct because for a given database D and a datalog program $\mathcal{P}$, there is unique greatest fixpoint of $\mathcal{P}$ for D [4]. For the data of Figure 2, and for the program $\mathcal{P}_0$, the greatest fixpoint consists of $\{person(g), person(j), company(a), company(m)\}$, which is as expected. Note that for this program, a least fixpoint semantics would fail to classify any object. The intuition behind the choice of the greatest fixpoint semantics is that we want to classify consistently as many objects as possible. The fact that it is a fixpoint indicates that the type of an object is justified by the types of objects connected to it.

Justifications of the above definition are also as follows. First, consider some relational data represented with *link* and *atomic* in the natural way: the entries of the tables are represented by atomic objects, the tuples by complex objects, and the labels are the attributes of relations. Consider the typing program corresponding to this schema also in a natural manner: one type is used for each relation. Then the previous typing would correctly classify the tuples. (This is assuming that no two relations have the same set of attributes for, in that case, their tuples would become indistinguishable.) Observe that for relational data, (i) the typing program is not recursive and thus the greatest fixpoint and the least fixpoint coincide and (ii) the data graph is bipartite in the sense that edges only go from complex objects to atomic ones.

For a second justification, consider some ODMG data [9] (ignoring collections such as lists or bags that are beyond our framework). For the natural representation of this data with $link$ and $atomic$, the natural typing program would correctly classify the objects.

Observe that the typed links allow to describe locally the structure of the objects in a class c. With typed links, one can state that there is some edge labeled ℓ going to (coming from) an object in some other class c' or going to (coming from) an atomic object. Note also that the language is quite restricted. For instance, it is straightforward to see that the typing rules can be expressed in first-order logic with 2 variables (FO^2) which is a very restricted subset of first-order logic. For instance, the rule for $person$ can be rewritten equivalently as:

$$person(X) \Leftrightarrow \exists Y(link(X,Y,is\text{-}manager\text{-}of) \wedge company(Y)) \wedge \exists Y(link(X,Y,name) \wedge \exists X(atomic(Y,X)))$$

that uses only two distinct variables. The fact that we limit ourselves to a framework such as the FO^2 logic may be an asset since that logic features nice properties [5], e.g., satisfaction is decidable for FO^2. On the other hand, observe that there are natural "typing" informations that fall outside our scope. For instance, one cannot express in FO^2, so with our rules, some simple restrictions on the cardinality of certain kinds of links, e.g., that companies have a unique name. Also, observe that even some rules that use only two variables are not allowed in our typing programs, e.g., the rule

$$\begin{aligned} person(X) \;\; :\text{-} \;\; & link(X, Y, is\text{-}manager\text{-}of) \;\&\; company(Y) \;\&\; link(Y, X, is\text{-}managed\text{-}by) \;\&\; \\ & link(X, Y', name) \;\&\; atomic(Y', Z) \end{aligned}$$

can be expressed using two variables only but is outside our framework.

Clearly, one could consider richer typing languages, and in particular, unrestricted monadic datalog programs. In the present paper, we focus on the previously defined simple types based on typed links.

Remark 2.1 *In the present paper, we ignore the value of the atomic objects and assign all of them to the same type. In practice, however, it is often easy to separate the atomic values into different sorts, e.g., integer, string, gif, sound, etc. Indeed, one can also apply (application specific) analysis techniques to enrich the world of atomic types with domains such as names, dates or addresses. It is straightforward to extend the framework to handle multiple atomic types.*

A more intense extension to our framework would be to consider some a priori knowledge of the typing. This may often occur in practice for instance if we attempt to integrate data with a known structure to semistructured data discovered on the net.

Finally, one may want to use in the typing, specific atomic values or ranges of atomic values. This would for instance allow to classify differently objects with values "Male" or "Female" in a sex *subobject. These are interesting extensions that should be considered in future work.*

Defect: Excess and Deficit In the case of relational and object data that are very regular and with the proper typing program, we obtain a *perfect* classification of the objects. In general, one should not expect this to happen. Suppose we have a program $\mathcal{P}$ that proposes a typing for a database D. We need a measure of how well $\mathcal{P}$ types D.

A first measure is the number of ground facts in D that are not used to validate the type of any object. We call this measure the *excess* since it captures the number of facts that are in excess. More precisely, let M be the greatest fixpoint of $\mathcal{P}$ for D. A ground fact $link(o, o', \ell)$ in D is in excess if there exist no class c, c', such that o is in $M(c)$, o' in $M(c')$ and the definition of c or c' stipulates that there is an ℓ-link from c to c'. The number of such ground facts forms the *excess*.

Excess is rather easy to capture with our datalog programs and the greatest fixpoint semantics. The deficit, i.e., some information that may be missing, is much less so. To define the deficit, we need also to be given a typing assignment τ in addition to a program P and a database D, that associates a set of objects to each type. The *deficit* of τ is the minimum number of ground facts that must be added to D (invented) in order to make all type derivations in τ possible. (A subtlety is that τ does not have to be a typing since the addition of these facts may bring some objects to more classes than specified by τ.)

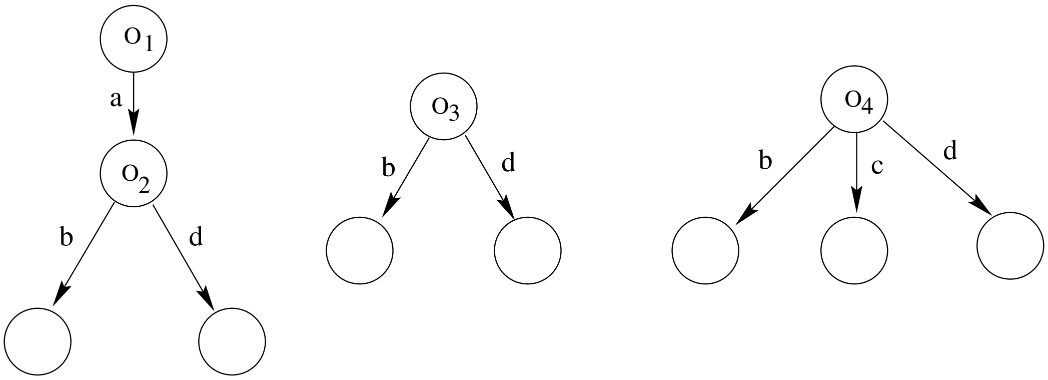

Figure 3: Example database

Example 2.2 *Suppose we are given the database shown on Figure 3 and the following typing program:*

$$\begin{array}{rcl} type_1 & = & \overrightarrow{a}^{2} \\ type_2 & = & \overleftarrow{a}^{1}, \overrightarrow{b}^{0}, \overrightarrow{c}^{0} \\ type_3 & = & \overrightarrow{b}^{0}, \overrightarrow{d}^{0} \end{array}$$

Consider two type assignments, τ_1 and τ_2, that both map o_i to $type_i$, for $i = 1, 2, 3$. They differ in that τ_1 maps o_4 to $type_2$ and τ_2 maps o_4 to $type_3$. Then the defect of τ_1 with respect to the given database and program is 2 because we have to "invent" one base fact, namely $link(o_1, a, o_4)$ and have to disregard $link(o_4, d, o_0)$. Thus, the excess is 1 and the deficit is 1 adding up to a defect of 2. For τ_2, we have only excess of 1 because we have to disregard $link(o_4, c, o_0)$. Thus, the defect is 1.

To conclude this section, note that the greatest fixpoint semantics may lead to excess but cannot yield deficit. Intuitively, deficit is more related to negation and hypothetical reasoning.

3 Method Summary

The goal of this work is to be able to approximately type a large collection of semistructured data efficiently. We are therefore led to making simplifying assumptions and introducing heuristics to be able to process this large collection in an effective way. In this section, we present the technique in rather general terms. The various steps are detailed in the following sections.

Our method for the approximate typing of semistructured data consists of three stages. As we shall see, there are several alternatives to be considered at each of the three stages. In order to decide which choices are most appropriate for a given database, we need some information about the data. It should be stressed that these choices remain primarily empirical and that the general process should entail using user feedback and adapting the technique to the particular application domain.

The gist of the first stage is to assign *every* object to a single *home type*. We use the minimal number of home types such that every object fits its home type *perfectly* (with no defect). The process of partitioning objects into a collection of home types is similar in spirit to bisimulation [13]. (However, some of the possible variations for this stage yield collections that differ significantly. For example, we could decide to have 'selected' objects with multiple home types.)

In the second stage, we address the optimization problem of reducing the number of types, and thus having objects that fit their home types with some defect, while incurring the lowest cumulative defect. This stage is the hardest both computationally and conceptually. We show that the general optimization problem is NP-hard even for a simple class of semistructured data corresponding to bipartite graphs. There are, however, techniques and heuristics adapted from *k-clustering* [12, 11] that allow efficient and near-optimal treatment of the problem. We also discuss the sensitivity of the solution with respect to the final number of types.

The third and final stage of our method is about recasting the original data within the chosen types. Ideally, the greatest fixpoint semantics of the typing program (consisting of the chosen types) should be employed. However, some of the techniques described in the second stage do not mix well with the fixpoint semantics. For example, some objects may be assigned to more than one particular home type, i.e., the objects don't have all typed links required by their home types. We present ways of resolving the incompatibilities and discuss some additional variations.

4 Stage 1: Minimal perfect typing

In this section, we present an algorithm for deriving a perfect (with no defect) typing program from semistructured data. In this program, every complex object has a type that is based on its local picture. The resulting object partitioning of the minimal perfect typing program is related to the partition obtained through bisimulation. We discuss this relationship towards the end of this section

4.1 Assuming a unique role

In this section, we assume that each object lives in a unique class. We will remove this restriction later.

Given some database D, the *minimal perfect* typing program $\mathcal{P}_D$ is constructed as follows:

1. First construct a program $\mathcal{Q}_D$ as follows. Let $o_1, ..., o_N$ be the complex objects. For each complex objects o_k, assign a unique type predicate $type_k$. The rule for $type_k$ will contain $\overset{\leftarrow i}{\ell}$ iff there is an edge labeled ℓ from o_i to o_k, and $\overset{\rightarrow i}{\ell}$ if there is an edge labeled ℓ from o_k to o_i. And the rule for $type_k$ will contain $\overset{\rightarrow 0}{\ell}$ iff there is an edge labeled ℓ from o_i to some atomic object.

2. Compute the *greatest fixpoint* M of $\mathcal{Q}_D$ for D. Let $\equiv$ be the equivalence relation on the types $\{type_k \mid k \in [1..N]\}$ defined by $type_i \equiv type_j$ if $M(type_i) = M(type_j)$. The types of $\mathcal{P}_D$ will be

the equivalence classes of $\equiv$, say $\tau_1, \ldots, \tau_n$.

3. The new program $\mathcal{P}_D$ is obtained by choosing for each τ_i, a type $\boldsymbol{type_k}$ in τ_i and replacing, in the rule r for $\boldsymbol{type_k}$, each type $\boldsymbol{type_j}$ by its equivalence class $[\boldsymbol{type_j}]$ according to $\equiv$. The home type of o_k becomes $[\boldsymbol{type_k}]$.

Remark 4.1 *The following property is useful in finding the equivalence classes of types (Step 2 above):* $\boldsymbol{type_i} \equiv \boldsymbol{type_j}$ *iff* $o_j \in M(\boldsymbol{type_i}) \wedge o_i \in M(\boldsymbol{type_j})$

The following example illustrates the algorithm for finding the *minimal perfect* typing program.

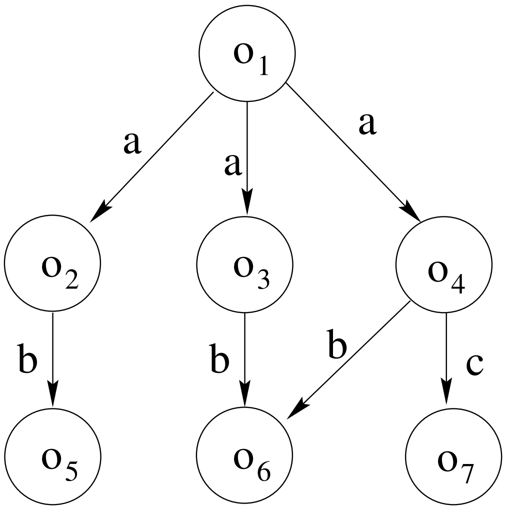

Figure 4: Simple database

Example 4.2 *Consider the simple database D in Figure 4. The program $\mathcal{Q}_D$ constructed in (1) of the algorithm is:*

$$
\begin{aligned}
\boldsymbol{type}_1 &= \overrightarrow{a}^{2}, \overrightarrow{a}^{3}, \overrightarrow{a}^{4} \\
\boldsymbol{type}_2 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0} \\
\boldsymbol{type}_3 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0} \\
\boldsymbol{type}_4 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0}, \overrightarrow{c}^{0}
\end{aligned}
$$

The greatest fixpoint M for $\mathcal{Q}_D$ obtained in (2) is:

$$M(\boldsymbol{type}_1) = \{o_1\}, M(\boldsymbol{type}_2) = M(\boldsymbol{type}_3) = \{o_2, o_3, o_4\}, M(\boldsymbol{type}_4) = \{o_4\}.$$

Let $[\boldsymbol{type}_1] = \tau_1$, $[\boldsymbol{type}_2] = [\boldsymbol{type}_3] = \tau_2$ and $[\boldsymbol{type}_4] = \tau_3$. The program $\mathcal{P}_D$ is:

$$
\begin{aligned}
\tau_1 &= \overrightarrow{a}^{3}, \overrightarrow{a}^{2} \\
\tau_2 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0}, \overrightarrow{c}^{0} \\
\tau_3 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0}
\end{aligned}
$$

The home type object partition for this program is:

- τ_1 *is the home type for* o_1.
- τ_2 *is the home type for* o_2 *and* o_3.
- τ_3 *is the home type for* o_4.

Computational Efficiency There is a straightforward method for computing the greatest fixpoint for a program $\mathcal{P}$. First, assign every type to every object and call this database M^{all}. Then compute $\mathcal{P}(M^{all} \cup link \cup atomic)$. Keep applying $\mathcal{P}$ to the result of the last application until no change occurs. The algorithm outlined above for finding the natural perfect typing program is not very efficient if implemented in the obvious way. There are several stages that must be handled carefully.

Computing the greatest fixpoint (Step 2 above) can potentially take double-quadratic time with respect to the size of the database (number of objects). One possible improvement is to notice that in the case when $\mathcal{P}$ is not recursive the greatest fixpoint coincides with the least fixpoint. There are many other possible improvements in the general case to the computation of the greatest fixpoint notably by using the fact that our programs are monadic, and also by using some differentiation techniques [18].

Also, it would be interesting to compare this cost to that of running a "bisimulation" style computation [8, 17, 13]. (Indeed, more generally, it would be interesting to consider in more depth the connection to approaches based on bisimulation.) Intuitively, two nodes are *bisimilar* if after the (possibly infinite) unfolding from each vertex and after duplicate elimination for subtrees, the two resulting (possibly infinite) regular trees are identical. A subtlety is that we do consider here both incoming and outgoing edges, which leads to also introducing edges corresponding to incoming edges when unfolding a vertex. Bisimulation turns out to be relatively easy to compute – the "relative" refers to the complexity of the definition. First, we consider that all objects are in a unique class c_0. At some stage, suppose that the objects are separated in a partition $\pi_1, ..., \pi_m$. If for some classes π_i, π_j and some label ℓ, there are objects in π_i that have ℓ edge going to objects in π_j and some that do not, one can split π_i in two. (Similarly, if some objects in π_i have incoming ℓ edges from π_j and some that do not.) This yields a more refined partition. Ultimately, this provides a partition of the set of objects and a type based on this partition.

4.2 Multiple roles

As the result of the minimal perfect typing so far, each object has its *home* type based on the object's local picture. Note however that the types defined by the minimal perfect typing program may still overlap. The reason is that the program does not contain negation. Thus, objects that have more typed links than required for a given type will also be assigned to it even though it is not their home type. This is in the style of ODMG inheritance but somewhat richer since our description of the locality of an object includes its outgoing edges (as in ODMG) but its incoming edges as well.

In the context of semistructured data, it seems often compulsory to remove the home type assumption that states that, for each object, there is a type that fully describes it. Objects may have multiple roles and each role may come equipped with a set of possibly overlapping attributes. For example a person may be an employee, a soccer player, a foreigner, a friend, etc., and each of its possible roles may come equipped with a pattern of incoming and outgoing edges. We want to avoid the combinatorial explosion of introducing employee-soccer-player-foreigner, employee-foreigner-friend, etc. Indeed, forcing each object to be in a single type would artificially increase the number of types or the error of the typing.

At this stage, we can identify complex types that can be expressed as a conjunction of several simpler types. By simpler we mean having less typed links in their definition. The home objects for the complex types can then be assigned to each of the simpler types that cover the complex one. Thus, at the end of this operation we will have an overlapping collection of types. The following example illustrates the main idea behind this.

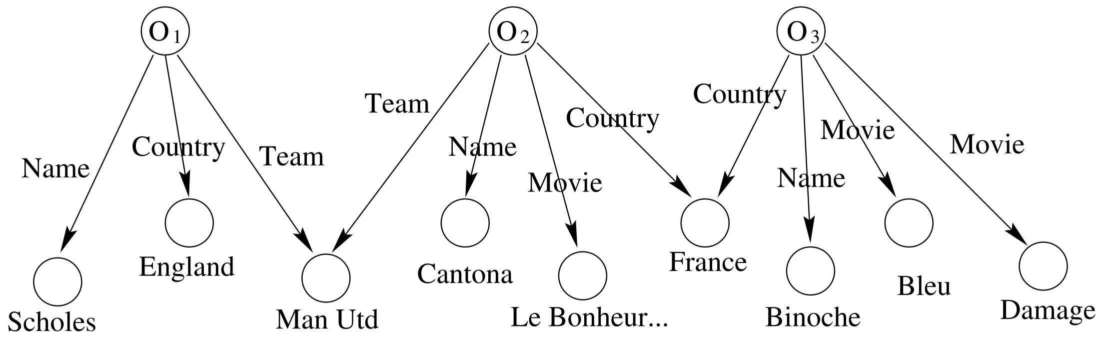

Figure 5: Soccer and movie stars

Example 4.3 *Consider the database in Figure 5. Its natural perfect typing program is:*

$$\begin{aligned} type_1 &= \overrightarrow{Name}^0, \overrightarrow{Country}^0, \overrightarrow{Team}^0 \\ type_2 &= \overrightarrow{Name}^0, \overrightarrow{Country}^0, \overrightarrow{Team}^0, \overrightarrow{Movie}^0 \\ type_3 &= \overrightarrow{Name}^0, \overrightarrow{Country}^0, \overrightarrow{Movie}^0 \end{aligned}$$

In the greatest fixpoint $type_1$ *contains* o_1 *and* o_2*;* $type_2$ *contains* o_2*;* $type_3$ *contains* o_2 *and* o_3*. Thus, even if we delete* $type_2$ *every object will still be assigned to at least one type. In that case,* o_2 *will loose its original home type but will be assigned two home types, namely* $type_1$ *and* $type_3$*.*

Remark 4.4 *Because of the special form of the type predicates it can be shown that eliminating types corresponding to multiple role objects is not very hard. It can be done in at most* $O(n^2)$ *where* n *is the number of types. Of course, there is still the problem of choosing a particular set of types for the cover. However, we expect that in real situations the choice will be obvious.*

It should be observed that although the introduction of new very general types may sometimes be useful, overdoing it may lead to some "atomization" of the information. Intuitively, one would like to avoid *describing a person as some object that is in a class* has-name *and in a class* has-address *and in a class* has-spouse.

5 Stage 2: Clustering

In most cases, the *minimal perfect* typing program will have too many types to be useful as a summary of the data set. There will be many 'similar' types that intuitively can be collapsed into one thus dramatically reducing the size and complexity of the typing program. In this section, we outline how to transform the typing program to reduce the number of types while keeping the defect (excess + deficit) low.

5.1 The general problem

The optimization problem that we consider in this stage is similar to k-clustering [11]. Every home type along with its weight (the number of objects having this type as their home type) is a point on a hypercube. The dimensions of the hypercube are the different typed links found in the minimal program from Stage 1. The general form of the problem, however, is more complex than k-clustering because deciding to coalesce several types has the effect of projecting all points on the hypercube on several of it diagonals and thereby reducing the dimensions. Consider the following example.

Example 5.1 *Consider the following four rather similar types:*

$$\begin{array}{lllllll} \tau_1 & :- & \overset{\rightarrow 0}{a}, \overset{\rightarrow 3}{b} & \qquad & \tau_2 & :- & \overset{\rightarrow 0}{a}, \overset{\rightarrow 4}{b} \\ \tau_3 & :- & \overset{\rightarrow 0}{a}, \overset{\leftarrow 1}{b} & \qquad & \tau_4 & :- & \overset{\rightarrow 0}{a}, \overset{\leftarrow 2}{b} \end{array}$$

Initially, all 4 types are different. However, if we coalesce either τ_1 and τ_2 or τ_3 and τ_4, the remaining two types become identical. Of course, in this case, it doesn't matter which pair is chosen first. However, there are situations where the order of coalescing has a significant effect on the quality of the result.

5.2 Distance function between types

There are many ways to define the distance between two types. We argue that while the fine tuning of the parameters of a specific function is very domain specific, the general properties of the distance function are universal.

Consider two types τ_1 and τ_2 and their definitions. The simplest and most natural distance function seems to be the *Manhattan path* between the two type points on the binary hypercube defined by the typed links in their definitions. In simpler terms, the distance is the number of typed links in the symmetric difference between the bodies of their rule definitions. We denote this distance, that is the basis of more complex functions considered later, by $d(\tau_1, \tau_2)$.

Example 5.2 *Consider the following three types:*

$$\begin{array}{lcl}\tau_1 & :- & \overset{\rightarrow 0}{a}, \overset{\rightarrow 2}{b} \\ \tau_2 & :- & \overset{\rightarrow 0}{a}, \overset{\leftarrow 1}{b} \\ \tau_3 & :- & \overset{\rightarrow 2}{b}, \overset{\leftarrow 1}{b}, \overset{\leftarrow 3}{b}\ .\end{array}$$

For τ_1, τ_2, the symmetric difference consists of $\{\overset{\rightarrow 2}{b}, \overset{\leftarrow 1}{b}\}$, so $d(\tau_1, \tau_2) = 2$. For τ_1, τ_3, the symmetric difference consists of $\{\overset{\rightarrow 0}{a}, \overset{\leftarrow 1}{b}, \overset{\leftarrow 3}{b}\}$, so $d(\tau_1, \tau_3) = 3$. And $d(\tau_2, \tau_3)$ is also 3.

Although this simple distance function appears to be very natural, it does not take into account the weight of the types (the number of objects having the type as their home type). We need to use a more complex *weighted distance* δ that should be a function of the Manhattan distance d, and the weights of the two types w_1 and w_2. We need to assume that the distance is not symmetric. Intuitively, this is because $\delta(w_1, w_2)$ measures the cost of moving type objects of type τ_2 to τ_1. It seems desirable to have the following properties for such a distance:

increasing in d This property is based on the intuition that it is better to collapse 'similar' types, i.e., such that there are very few typed links in one and not in the other.

decreasing in w_1 This is based on the intuition that the expected noise around some class of object should be proportional to the number of objects in the class. In other words, if the class has a very large extent, we may expect a lot of objects that almost fit in it but not quite and should be willing to correct them.

increasing in w_2 The intuition behind this last property is that large collections of similar objects are likely to form types and thus should not be moved to other types (unless the other type is much bigger and thus the previous property kicks in).

These three properties are clearly related to the overall goal of minimizing the defect.

There are several possible functions that seem reasonable choices even though some of them don't satisfy all three properties listed above:

$$\begin{array}{ll}\delta_1(w_1, w_2, d) = d * w_2 & \delta_2(w_1, w_2, d) = L^d/(w_1 * w_2) \\ \delta_3(w_1, w_2, d) = (w_1 * w_2)^{1/d} & \delta_4(w_1, w_2, d) = (w_2/w_1)^{1/d} \\ \delta_5(w_1, w_2, d) = L^d * w_2 & \end{array}$$

where L is the total number of different typed links in the typing program obtained at the end of Stage 1. Clearly, the choice of a distance function seriously affects the results of the typing. The following example shows that deciding on the particular parameters for a given function is domain specific.

Example 5.3 *Suppose at the end of Stage 1 there are only three types.*

- *100000 objects of type* $\tau_1 = \vec{a}^0 \vec{b}^0$
- *1000 objects of type* $\tau_2 = \vec{a}^0 \vec{b}^0 \vec{c}^0$
- *100 objects of type* $\tau_3 = \vec{a}^0 \vec{b}^0 \vec{\ell_1}^0 \dots \vec{\ell_k}^0$

Suppose that we want to end up with only two types at the end of Stage 2. We implicitly assume that one extra type will be the empty set allowing us to chose not to type some objects by assigning them to the empty set type. For $k = 1$*, the best solution will be to move* τ_3 *to* τ_1*. Similarly, for a big* k*, e.g.,* $k > 15$*, the best solution is to move* τ_2 *to the* τ_1*. In between, there is a range for* k *such that the best solution is to move* τ_3 *to the empty set type, i.e, to not classify those 100 objects with a home type* τ_3*. The two cut-off points depend on the distance function that is chosen and are clearly application dependent.*

Note that the distance function δ_1 resembles our definition of defect introduced in Section 2. While it measure the defect exactly for a single coalescing when we have a series of type coalescing it only provides an upper bound on the defect of the final program.

Clustering algorithm Since finding the optimal k types is NP-hard we have to employ heuristics in order to solve the problem. In our experiments we used a greedy algorithm because of it lower time complexity and implementation ease. Furthermore, under certain assumptions, the greedy algorithm gives an $O(\log n)$-approximation of the best solution [11].

To conclude this section, we consider a special case that is somewhat easier and an alternative to the clustering in general.

Bipartite graphs An important special case is when all typed links point to atomic objects which happens when the graph is bipartite. This is the case for relational data or when the data comes from a file of records. Then each type is defined by the set of labels on the outgoing links, i.e. the attributes in the relational case. The problem is much simpler. However, even in this simple case, one can show that finding the best typing with k types (for some fixed k), where “best” is defined by minimizing the defect, is still NP-hard.

Variation to k-clustering A different approach is to first consider the types after Stage 1 without their weights. Using some measure of the relative importance of an attribute within a set of attributes (e.g. the *jump* function [14]) we can find the best k clusters of the types and only use the weights within a cluster to determine its type definition corresponding to its center. However, this approach may run into problems if there are many outliers and the hypercube is densely populated.

6 Stage 3: Recasting

In the third stage we allow objects to be in types other than their home type(s) if they satisfy the appropriate type predicates. At this stage however we do not account for the excess or deficit. Thus, at the end of the third stage we have typed approximately all objects with k types at some defect cost. Note that the first stage is independent of the choice of k. Thus, we can support a sliding scale mechanism where the scale is k and the result is the best k types and the corresponding defect. Infact, our experiments suggest that using this approach yields better results and provides additional insight into the data. We present more detailed discussion of this approach in Section 7.

When we allow objects to be assigned to types other than there home type(s) we actually have several options depending on whether we only classify objects based on their actual typed links or the ones suggested by their home type assigned at the end of Stage 2.

The typing rules for objects that have not been used to derive the typing program are rather simple. First we assign the new objects to all types that it satisfies completely. If the object cannot be assigned any type precisely than we assigned it to the closest type to it, in terms of the simple distance function d. Of course, if we have many new objects we may wish to reconsider the the current typing program. Deciding how many new objects is too many and recomputing efficiently the typing program are open problems.

7 Experimental results

In this section we present some preliminary experimental results. While performance (in terms of time) is an important consideration in our work, the main focus of the experiments was the quality of the results. Indeed, understanding when the various options in our algorithm work best and how is a prerequisite for designing efficient data structures and optimization. In this performance study we used extensively synthetic data. We also show some results on a operational data set. Note the using synthetic data is attractive for the purpose of evaluating the quality of the typing in several ways. First, we are able to compare the types produced by our algorithm with the *intended* type in the data specification. Second, we are able to measure the effects of various perturbation of the data on the the typing results.

7.1 Generating Synthetic Data

The main idea behind the synthetic data is to use type definition with probability attached to their typed links and then produced random instances according to those probabilities. The following example illustrate the data generation process.

Example 7.1 *Consider the following simple type specification with attached probabilities. There are two types in addition to the standard atomic type. Objects of the first type have a link labeled 'a' to an atomic objects with probability 0.9 and 'b' link to atomic with probability 0.5. Objects of the second type have a*

link	FromObj	ToObj	Label
	C1	*A1*	a
	C1	*A5*	b
	C2	*A3*	a
	C3	*C1*	c
	C3	*A2*	b
	C4	*A1*	b
	C5	*C2*	c

Figure 6: Synthetic Data

link labeled 'c' to an objects of the first type with probability 0.8 and 'b' link to atomic with probability 0.9. Figure 6 gives a small database that might be generated from this type specification.

The results of running our typing algorithm for several synthetic data sets are captured in Table 1. The distance function used in the clustering stage is the weighted Manhattan distance. The clustering is done by a greedy algorithm.

We run experiments on 4 different synthetic datasets (DB Nos. 1,3,5,7). For each dataset we denote whether its corresponding graph is bipartite and whether the intended types are overlapping, i.e, have typed links in common. We also consider a slight perturbation of each dataset (DB Nos. 2,4,6,8) where we delete randomly a few links in the graph and then add some randomly labeled links.

The main observation from the results is that slight perturbation of the dataset results in a dramatic increase of the number of perfect types while the effect on the optimal approximate typing is relatively small. Another observation is that datasets with bipartite graphs are much easier to handle compared to regular graphs.

7.2 Sensitivity Analysis

There is clearly a trade-off between the defect and the simplicity of the typing program. For example, the minimal perfect typing program has no defect but has too many types. On the other side of the scale, if we choose to have only one type the defect will be huge unless we are dealing with very regular data. We conjecture that for non-random semistructured data there is usually an optimal number (or a small range) of types. Figure 7 show the defect and the total distance used in the algorithm as a function of the number of types in the approximate typing. The distance function is the weighted Manhattan distance between the types. As expected, there is a small range of types (6-10) the yields optimal tradeoff between number of types and defect. The optimal typing with 6 types is shown in Figure 1

The existence of optimal range of number of types suggests that an interactive approach to typing

Synthetic Data							*Typing*		
DB No	*Bipartite ?*	*Overlap ?*	*Perturb ?*	*Intended Types*	*Objects*	*Links*	*Perfect Types*	*Optimal Types*	*Defect*
1	Y	N	N	10	1500	2909	30	10	225
2	Y	N	Y	10	1500	2958	52	10	307
3	Y	Y	N	6	950	2409	19	6	239
4	Y	Y	Y	6	950	2442	35	6	283
5	N	N	N	5	400	726	317	5	181
6	N	N	Y	5	400	749	341	5	310
7	N	Y	N	5	400	775	375	5	291
8	N	Y	Y	5	400	795	381	5	333

Table 1: Synthetic Data Results

semistructured data will work best. Instead of deciding in advance on a fixed number of types in the approximate typing it is better to explore several different values ranging from as many as in the minimal perfect typing to perhaps just 1. Note that the algorithm can be adapted such that the we find sequentially the best fit with k types starting from the number of types in the perfect typing. Thus, the algorithm can find the optimal tradeoff point and suggest a 'natural' typing (or a small set). If the results are unsatisfying because of too much defect or too many types. The algorithm can keep reducing the number of types or revert to a typing with more types but less defect. However, we feel that having hard limits on the number of types or the defect, without having knowledge of the data is unreasonable.

8 Conclusions

In this paper, we presented a method for extracting schema from semistructured data. The schema is in the form of a monadic datalog program with each intensional predicate defining a separate type. We asserted that in the context of semistructured data it is imperative to allow for some defect when objects are typed. This assertions was supported by the preliminary experimental results on both operational and synthetic data. The perfect typing (with no defect) was shown to be much bigger than the approximate typing produced by our method. Indeed, in some cases the perfect typing was of roughly the same size as the data which precludes its practical use. In contrast, the size of approximate typing can always be reduced to a desired range. Our experiments suggested that even better results can be obtained by considering the defect as a function of the number of types in approximate typing and choosing an optimal range.

Acknowledgments We would like to thank Jeff Ullman, Peter Buneman, Victor Vianu, for discussions on this topic.

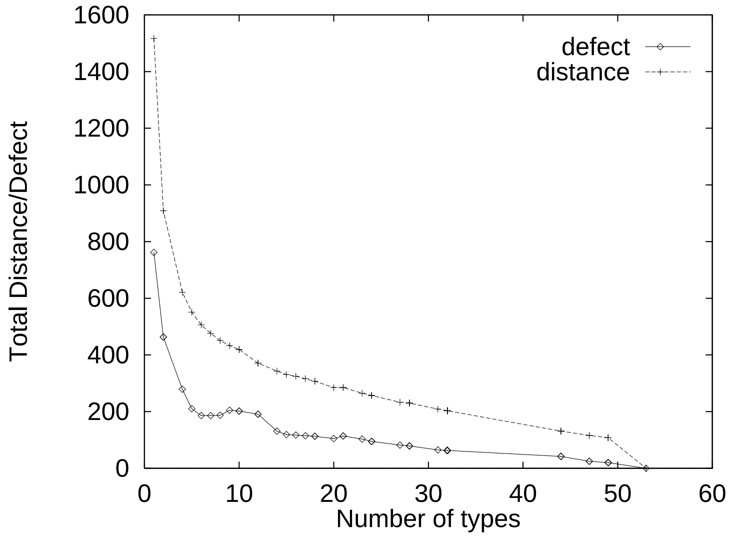

Figure 7: Sensitivity Graph for DBG data set

References

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of ICDT*, pages 1–18, Delphi, Greece, January 1997.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading,Massachusetts, 1995.

[3] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of VLDB*, pages 39–51, Taipei, Taiwan, 1993.

[4] K.R. Apt. *Logic Programming, Handbook of Theoretical Computer Science*. J. Van Leeuwen, Elsevier, 1991.

[5] E. Borger, E. Graedel, and Y. Gurevich. *The classical decision problem*. Speinger-Verlag, Berlin Heidelberg, 1997.

[6] P. Buneman. Semistructured data: a tutorial. In *Proceedings of PODS*, Tucson, Arizona, May 1997.

[7] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Addind structure to unstructured data. In *Proceedings of ICDT*, pages 336–350, Delphi, Greece, January 1997.

[8] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference*, pages 505–516, Montreal, Canada, June 1996.

[9] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1994.

[10] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[11] D.S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22:148–162, 1982.

[12] M.R. Korupolu, C.G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California, January 1998.

[13] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[14] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.

[15] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of ICDE*, pages 79–90, Birmingham, U.K., April 1997.

[16] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases (DOOD)*, pages 319–344, Singapore, December 1995.

[17] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *Proceedings of VLDB*, pages 227–238, 1996.

[18] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I,II*. Computer Science Press, Rockville, Maryland, 1989.

[19] M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16:324–343, 1977.