Schemas for Integration and Translation of Structured and Semi-Structured Data *

Catriel Beeri[1] and Tova Milo[2]

[1] Hebrew University beeri@cs.huji.ac.il
[2] Tel Aviv University milo@math.tau.ac.il

1 Introduction

The Web is emerging as a universal data repository, offering access to sources whose data organization varies from strictly structured databases to almost completely unstructured pages, and everything in between. Consequently, much research has recently focused on data integration and data translation systems [10, 6, 9, 8, 17, 13, 2, 19], whose goals are to allow applications to utilize data from many sources, with possibly widely varying formats. These research efforts have established a common *data model* of semistructured data, for uniformly representing data from any source. Recently, however, it is being realized that having a common *schema model* is also beneficial, and even necessary, in translation and integration systems to support tasks such as query formulation, decomposition and optimization, or declarative specification of data translation.

As an example, which we use for motivation throughout the paper, recently suggested tools for data translation [2, 11, 19] use the semistructured data model as a common middleware data model to which data sources are mapped. Translation from source to target formats is achieved by (1) importing data from the source to the middleware model, (2) translating it to another middleware representation that better fits the target structure, and then (3) exporting the translated data to the target system. In [11, 19], *schema* information is extensively utilized in this process. Source and target formats are each represented in a common schema language; then a correspondence between the source and target schemas can be specified in the middleware model. In many cases, the two schemas are rather similar, with differences mostly due to variations in their data models; after all, they both represent the same data. Consequently, most of the correspondence between the two schemas can be determined *automatically*, with user intervention required only for endorsement or for dealing with difficult cases. Once such a correspondence is established, translation becomes an automatic process: Data from the source is imported to the common data model and is "typed", i.e. matched to its schema, thus its components are associated with appropriate schema elements. Next, the schema correspondence is used for a translation of the data to the target structure, where each component is translated according to its type. Then data is exported to the target

* The work is supported by the Israeli Ministry of Science and by the Academy of Arts and Sciences

The above idea was in fact implemented in the TranScm translation system, whose the architecture and functionalities are described in [19]. In the present paper we elaborate on the theoretical foundations of the middleware schema model. (The model has only been informally sketched in [19] and its properties have not be investigated there.) While we refer to the TranScm system for motivation, our results are relevant to other translation systems (e.g. YAT [11]), and to other applications of semi-structured data such as data integration. Our main contribution is the presentation of schema definition languages that are both expressive and flexible, and an investigation of properties such as expressive power and the complexity of significant decision problems for the context of data translation and integration. Our schema languages are considerably more expressive than those found in recent work on schematic constraints for semi-structured data (e.g. [5, 16]), providing a combination of expressibility and tractability. Another contribution here is the extension of the data model to support order in data, not supported by the common semistructured data model, both in the data and in the schema level. The closest to our model is YAT [11], supporting order and an expressive schema language, but lacking some of the features described here. (They also do not mention expressiveness and complexity results.) Further comparison is presented in the last section.

The paper is organized as follows. In Section 2 we present our data model and two schema specification languages. We first introduce a basic schema specification language and illustrate its flexibility and expressive power, then we discuss an extension that is useful for data translation, and show it increases the expressive power. Relationships between the languages and context-free/regular grammars are established. Detection and removal of redundancy from schemas is discussed in Section 3, the closure of schemas under operations in Section 4, and the complexity of matching of data to schemas and related problems in Section 5. In Section 6 we present a preliminary study of the computation of schema information for queries and its use for optimization. Section 7 summarizes the results. All figures appear in the Appendix.

2 Data and Schemas

We define here the middleware data model, introduce our first schema specification language and illustrate its capabilities. Then we explain an extension useful in data translation. Close relationships between the languages and context-free/regular grammars are established; these are extensively used in the sequel.

2.1 The Data Model

The common model for semi-structured data [20, 6, 2] represents data by trees or forests, augmented with references to nodes. Thus, it is essentially that of labeled directed graphs. This, with the significant addition of order, is also our model. More differences and extensions are explained below.

Definition 1. *Let $\mathcal{D}$ be an infinite set of* **label values**. *A* **data graph** $G = (V, E, L, O, \omega)$ *is a finite directed node-labeled graph, where V and $E \subseteq V \times V$ are finite sets of nodes and edges, respectively; $L : V \rightarrow \mathcal{D}$ is the labeling function for the nodes; $O \subseteq V$ is the set of* **ordered nodes**; *and ω associates with each node in O a total order relation on its outgoing edges.*

A **rooted** *data graph is a graph as above, with a designated* **root** *node, denoted v_0, and where each node in the graph is reachable from v_0.*

Like in all models in the literature, node labels in our model can be used both to represent data such as integers and strings, and to represent schematic information, such as relation, class, and attribute names. While it might seem natural to represent schematic data as edge labels, and 'real' data as node labels, for convenience usually only one of the two kinds of labels is used. For our work, node labels seem to have an advantage. It is easy to convert edge labels to node labels: create a node to represent the edge, and move the label to it. Thus, our results apply to edge-labeled graphs as well.

Allowing an order to be defined on the children of some of the nodes is a significant extension, missing in most previous models. Order is inherent in data structures such as tuples and lists, and is essential for representing textual data, either as sequences of characters or words, or on a higher level by parse trees. As shown in [2], supporting order in the data model enables a natural representation of such data. By allowing order to be defined for *part of the nodes* we are able to naturally describe data containing both ordered and unordered components.

Reachability of data from one or more roots is a de-facto standard in essentially all data sources, from database systems to Web sites. In the sequel, we are mainly interested in rooted graphs; non-rooted graphs are used mainly for technical reasons. For brevity in the sequel a *data graph* will mean a rooted one (unless explicitly stated otherwise). For simplicity we assume here a single root, but all subsequent results can be naturally extended for the case of multiple roots.

2.2 Schemas

We now present the basic Schema Definition Language, ScmDL. A schema is a set of type definitions. Following the SGML/XML syntax for element definition[15, 1], the definition of a type (schema element) has two parts. The first is a regular expression describing the possible sequences of children that a node can have. We use common notation for regular expressions: ϵ denotes the empty language, and $\cdot$ and $|$ denote concatenation and alternation resp. The second part contains attributes that describe properties of a node. Three of these are boolean flags, that determine if instance data nodes of this type (i) are ordered, (ii) can be referenced from other nodes; only such nodes can have more than one incoming edge, (iii) can be roots of the data graph. The fourth attribute is a unary predicate, that determines the possible labels of data nodes of this type. We do not use any specific predicate language. We assume given some collection $\mathcal{P}$ of predicates over $\mathcal{D}$, the domain of label values, that is closed under the boolean operations, and such that satisfiability of a predicate, and whether $p(d)$ holds, for a predicate p and a label d, can be decided in time polynomial in d and p's definition.[1] We also assume that: (a) For base types such as $Int, String$, $\mathcal{P}$ contains a corresponding base predicate satisfied by the relevant subset of $\mathcal{D}$ (b) For each constant $d \in \mathcal{D}$ there is a predicate satisfied only by d (i.e. $\lambda x.x = d$), denoted by *is-d*, where in the case of strings, the quotes are omitted: e.g. *is-foo* denotes the predicate satisfied only by "foo". (c) Finally, to capture cases where no restrictions are imposed on the labels, we assume that $\mathcal{P}$ also contains a predicate, denoted *Dom*, that is satisfied by all the elements in $\mathcal{D}$ (i.e. $\lambda x.true$).

Definition 2. *Let $\mathcal{T}$ be an infinite set of type names. A* **type definition** *has the form*

$$t = \ R_t\ (label = t_{label}, ord = t_{ord}, ref = t_{ref}, root = t_{root})$$

where $t \in \mathcal{T}$, R_t is a regular expression over $\mathcal{T}$, t_{label} is a predicate name in $\mathcal{P}$, and $t_{ord}, t_{ref}, t_{root}$ are boolean values. (For additional conventions and abbreviations, see below.)

A ScmDL **schema** *is a pair $S = (T, Def)$, where $T \subset \mathcal{T}$ is a finite set of type names, and Def is a set of type definitions containing precisely one definition for each of the type names in T and using only type names in T.* [2]

We use $Lang(t)$ to denote the regular language defined by R_t, and we use $t.a$, for $a \in \{label, ord, root, ref\}$, to denote the value of the attribute a in t's definition. We say that a type t is **ordered** (is **a root**, is **referencable**) iff $t.ord$ ($t.root$,$t.ref$, resp.) is true.

It can be seen that we follow the SGML/XML style of definition [15, 1], as shown in Figure 1, in that a type definition describe both the attributes of nodes, and their children. We chose a less verbose style, for brevity.

$$\begin{array}{llll}
<!ELEMENT\ t & (R_t)> & & \\
<!ATTLIST & t & & \\
 & label & t_{label} & \#Required \\
 & ord & Boolean\ t_{ord} & \\
 & ref & Boolean\ t_{ref} & \\
 & root & Boolean\ t_{root} > &
\end{array}$$

Fig. 1. SGML/XML-style type definition

A ScmDL schema defines a set of data instances. Intuitively a data graph G **conforms** to a schema S if each of the nodes $v \in G$ can be assigned a type t s.t.

[1] This assumption is used in deriving some of our complexity results; the consequences of dropping it on these results are quite obvious.

[2] In [19] schemas are presented graphically, as the system is oriented towards a graphical interface for defining schemas and representing schemas and data.

v satisfies the requirements of t, as described in t's definition. This is formally defined below.

Definition 3. *Let G be a data graph and S be a schema. We say that G* **conforms** *to S (or is an* **instance** *of S), if there is a type assignment on its nodes, i.e. a total mapping h from the nodes in G to types in S s.t. for each node $v \in G$ the following holds:*

1. *v's label satisfies $h(v).label$,*
2. *v is ordered iff $h(v)$ is ordered,*
3. *If v is a root node then $h(v)$ is a root type,*
4. *If v has more than one incoming edge then $h(v)$ is referencable,*
5. *Finally, if v is ordered and its children are $v_1, \ldots, v_n$ (in that order) then the sequence $h(v_1), \ldots, h(v_n)$ is a word in $Lang(t)$, and if v is unordered then there exists some order on its children for which the above holds.*

The set of all data graph instances of a schema S is denoted $inst(S)$, and we say that two schemas S, S' are **equivalent** *if $inst(S) = inst(S')$.*

Notational conventions: To further simplify type definitions, we assume the convention that flags whose value is false are omitted, while those that are true are represented by their name. Thus

$$t = \quad R_t \ (label = t_{label}, ord)$$

represents a type for which $ord = true, ref = false, root = false$. Another convention is used for types of leaf node, i.e. nodes that have no outgoing edges, hence their regular expression is ϵ. Almost always, for leaves all three flags are false. Further, in essentially all cases, the labels of such nodes are values of base types, such as $Int, String$, that have a corresponding predicate. Hence, from now we consider type definitions for such leaves as given, as in the following definition,

$$Int = \quad \epsilon \ (label = Int, ord = false, ref = false, root = false)$$

and freely use $Int, String$, as type names in schemes without defining them.

2.3 Examples

Descriptive power is an obvious requirement from a middleware schema language. One should be able to describe the strict and precise formats of database sources, the variety of formats used in scientific disciplines, the very loose formats of Web pages, and also cases where the data has parts that are precisely specified, and other parts that are allowed to vary. We now illustrate the expressiveness and flexibility of the language in describing possible structures of data.

An OODB schema and its instance database are shown in Figures 2 and 3, respectively. The corresponding ScmDL schema and data graph are shown in Figures 4 and 5. Note that *article* is the root, *authors* is the only ordered

$$
\begin{array}{l}
\textit{class article public type} \\
\qquad \textit{tuple (title} \quad : \textit{string}, \\
\qquad\qquad \textit{authors} \ : \textit{list(author)}, \\
\qquad\qquad \textit{contact\_author} : \textit{author}, \\
\qquad\qquad \textit{sections} : \textit{set(string)} \ ) \\
\\
\textit{class author} \ : \ \textit{string};
\end{array}
$$

Fig. 2. OODB Schema

class name	oid	value
article	o_0	tuple(title : "From Structured Documents to Novel Query Facilities", authors : list(o_1, o_2, o_3), contact_author : o_1, sections : set("Structured documents ...",...))
author	o_1	V. Christophides
author	o_2	S. Abiteboul
author	o_3	S. Cluet

Fig. 3. An OODB Instance

node, and that the data graph conforms to the schema, with the natural type assignment.

A DTD for an SGML document and its instance document are presented in Figures 6 and 7 respectively. The corresponding ScmDL schema and data graph are presented in Figures 8 and 9. All the internal nodes are ordered, with *article* being the root.

The above were examples for homogeneous, strictly structured data. The schema in Figure 10 illustrates the opposite case, where no constraints at all are imposed on data graphs. (Recall that *Dom* is the label predicate $\lambda x.true$.) It is easy to see that all data graphs conform to the above schema with a type assignment h assigning any_{ord} to all the ordered vertices and any_{unord} to the unordered ones.

$$
\begin{array}{lll}
article & = title \cdot authors \cdot contact\_author \cdot sections & (label = is\text{-}article, root, ref) \\
title & = String & (label = is\text{-}title) \\
authors & = author^* & (label = is\text{-}authors, ord) \\
author & = String & (label = is\text{-}author, ref) \\
contact\_author & = String & (label = is\text{-}contact\_author) \\
sections & = String^* & (label = is\text{-}sections)
\end{array}
$$

Fig. 4. ScmDLschema for the OO data graph

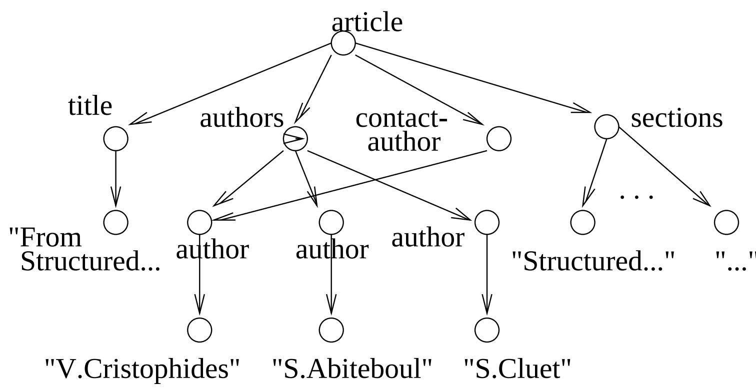

Fig. 5. Data graph of an OODB article

$$
\begin{array}{ll}
<!DOCTYPE\ article\ [& \\
<!ELEMENT\ article & (title,\ author^*, contact\_author, section^*) > \\
<!ELEMENT\ title & (\#PCDATA) > \\
<!ELEMENT\ author & (\#PCDATA) > \\
<!ELEMENT\ contact\_author & (\#PCDATA) > \\
<!ELEMENT\ section & (\#PCDATA) >
\end{array}
$$

Fig. 6. SGML DTD

$$
\begin{array}{l}
< article > \\
< title > From\ structured\ Documents\ to\ Novel\ Query\ Facilities\ \ < /title > \\
< author >\ \ V.\ Christophides\ \ < /author > \\
< author >\ \ S.\ Abiteboul\ \ < /author > \\
< author >\ \ S.\ Cluet\ \ < /author > \\
< contact\_author >\ \ V.\ Christophides\ \ < /contact\_author > \\
< section > Structured\ documents\ are\ central...\ \ < /section > \\
\ldots < /article >
\end{array}
$$

Fig. 7. SGML Document

$$
\begin{array}{lll}
article & = \ title \cdot author^* \cdot contact\_author \cdot section^* & (label = is\text{-}article, ord, root) \\
title & = \ String & (label = is\text{-}title, ord) \\
author & = \ String & (label = is\text{-}author, ord) \\
contact\_author & = \ String & (label = is\text{-}contact\_author, ord) \\
section & = \ String & (label = is\text{-}section, ord)
\end{array}
$$

Fig. 8. ScmDLschema of an SGML article

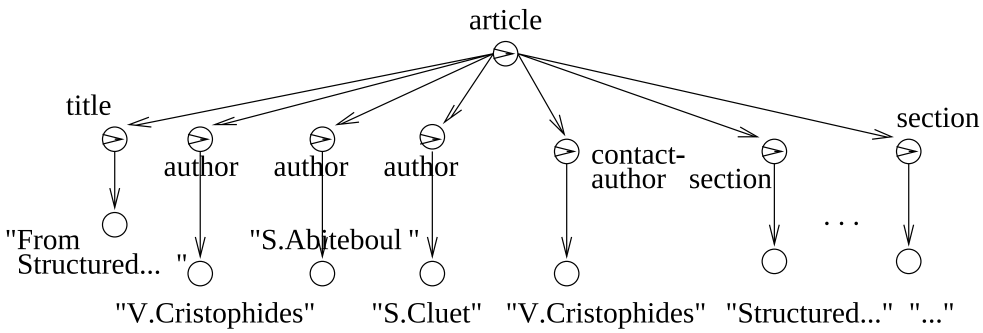

Fig. 9. Data graph of an SGML article

$$\begin{array}{lcl} any_{ord} & = & (any_{ord} \mid any_{unord})^* \ (label = Dom, ord, root, ref) \\ any_{unord} & = & (any_{ord} \mid any_{unord})^* \ (label = Dom, root, ref) \end{array}$$

Fig. 10. Schema for arbitrary graphs

Finally, we present an example for the modeling of partial constraints. Assume that we are modeling a web site with arbitrary structure, but where one of the links leads to an OODB describing articles, with structure as described above. The schema for such mixed data is presented in Figure 11. (W.l.o.g. we assume that all the nodes representing Web pages are ordered, reflecting the order of elements in the page.)

$$\begin{array}{lcll} mixed & = & any^*.(db \mid mixed).any^*. & (label = Dom, ord, root, ref) \\ any & = & any^* & (label = Dom, ord, ref) \\ db & = & article^* & (label = Dom) \\ article & = & defined\ as\ in\ Figure\ 4 & (\ldots) \end{array}$$

Fig. 11. Schema for a Web site containing an articles OODB

Note that in [5], schema information was captured using *schema-graphs*, i.e, graphs whose nodes represent types and whose edges describe the possible children that a node of a certain type may have. We argue that ScmDL is powerful enough to express all the schema constraints expressible by the *schema-graphs* of [5] and **much more**:[3] We can *require* a node to have a certain outgoing edge, while schema-graphs only allows to state that such an edge is *possible*. We also deal with order and referencebility constrains, while they don't. The following proof is omitted for space reasons.

[3] As already mentioned, the fact that they use edge labels while we use node labels does not prevent a meaningful comparison.

Proposition 1. *Every schema-graph has an equivalent* SCMDL *schema, but not vice versa.*

2.4 Virtual nodes and types

Typically, in a translation scenario, the target schema is different from that of the source. One common difference is that in the source the grouping of data into subtrees/subgraphs is less (or more) refined than in the target. For example, assume we want to convert the SGML document in Figure 7 into an OO format, as in Figure 3. In both formats an article consists of four logical components, namely the title, the authors list, the contact author, and the sections. However, while in the OO data graph (Figure 5) each of these parts occurs in a subtree under an appropriately named node, this does not hold in the data graph of the SGML document (Figure 9), which is just a sequence of elements. Note that the refined structure (implicitly) exists in the parse tree of the document, relative to the regular expression defining it. For facilitating the translation, it is convenient to make the logical structure explicit and split the sequence into its logical sub-components. That is, rather than looking at the "real" SGML instance in Figure 9, it is convenient to use a "virtual" graph with structure as in Figure 12, remembering that the *authors* and *sections* are "virtual" vertices that do not really occur in given the data.

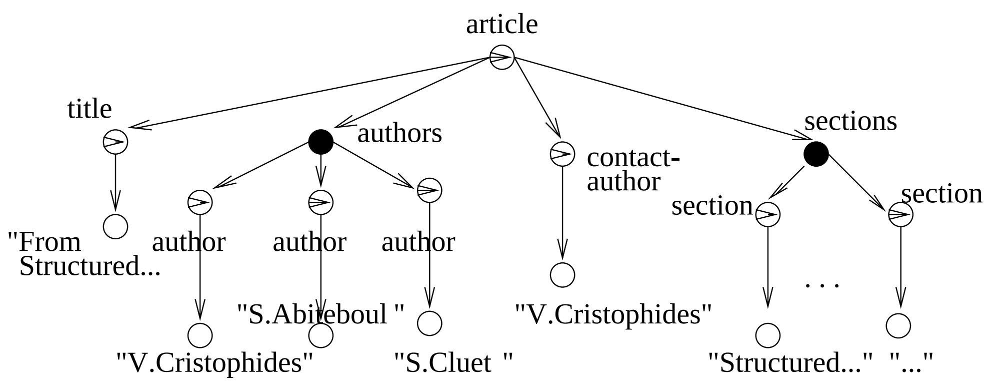

Fig. 12. Virtual version of the data graph of an SGML article

The first step is to define *virtual* graphs and schemas, and extend the notion of conformity accordingly. Observe that, since virtual nodes represent logical components that do not actually occur in the data, they cannot be serve as roots or be referenced (i.e. have more than one incoming edge). Also, since virtual nodes are introduced to represent subsets or subsequences of the children of a given node, if the node is (un)ordered so should be the virtual types used in its definition.

Definition 4. *A* **virtual** *data graph is a data graph with some nodes marked as virtual. A* **virtual** *schema is a schema as in Def 2, with one more boolean attribute,* virtual, *in the type definitions, such that:*
(i) a virtual type is neither referencable nor a root,
(ii) if a virtual type t_v occurs in the regular expression associated with a type t, then t, t_v are both ordered or both unordered.
The extended schema definition language is denoted VScmDL. *A virtual graph G_V* **v-conforms** *to a* VScmDL *schema S if there exists a type assignment h satisfying all the conditions in Definition 3, and additionally v is virtual, iff so is $h(v)$.*

Note that by the definition, a virtual node in an instance data graph is not the root, is not referencable, and is ordered iff its parent (if one exists) is. Hence we will consider in the sequel only virtual graphs where all the virtual nodes have this property. We refer to data graphs without virtual nodes, to types whose *virtual* flag is false, and to schemas that contain only such types (equivalently ScmDL schemas) as **real**.

We can now state the relationship between real data graphs and virtual graphs and schemas.

Definition 5. *Let G and G_v be a real and a virtual data graphs, resp. We say that G_V is a (possible)* **virtual** *version of G if G can be obtained from G_v by identifying each virtual node with its closest non-virtual ancestor, gluing its children directly to this ancestor, preserving the order of the children, and using the label of the ancestor for the resulting node.*

Definition 6 (conformity - revisited). *A real data graph G* **conforms** *to a* VScmDL *schema S if some virtual version G_V of G v-conforms to S by some type assignment h. This h is called a* **virtual type assignment** *for G; its restriction to nodes of G is called a* **type assignment** *for G.*

The set of virtual graphs that v-conform to S is denoted $v\_inst(S)$, the set of real graphs that conform to it is denoted $inst(S)$; two schemas S, S' are (v-) equivalent if $inst(S) = inst(S')$ (resp. $v\text{-}inst(S) = v\text{-}inst(S')$).

Continuing with the above example, the virtual graph in Figure 12 is a possible virtual version of the data graph in Figure 9. Thus the SGML article of Figure 9 conforms to the following virtual schema due to this virtual version, with the natural (virtual) type assignment.

$$\begin{array}{lcll} article & = & title \cdot authors \cdot contact\_author \cdot sections & (label = is\text{-}article, ord, root) \\ authors & = & author^* & (label = is\text{-}authors, ord, virtual) \\ sections & = & section^* & (label = is\text{-}sections, ord, virtual) \end{array}$$

Note that if a VScmDL has no virtual types (i.e. is a real schema, or equivalently, a ScmDL schema) then conformity and v-conformity coincide, and so do $inst(S)$ and $v\text{-}inst(S)$

Recalling the above SGML-to-OODB translation process, we can see that the introduction of virtual types facilitates the comparison of the source and target schemas, and the correspondence between their types. (For additional illustrations and translation examples see [19].) Similarly, the introduction of virtual nodes into the source data facilitates its matching to its schema augmented with these virtual types, and hence the translation of the data. Finding where such virtual nodes are needed is part of the conformity test. The complexity of this and related problems are discussed in the following sections. Observe that, in general, a data graph may have more than one possible virtual version and (virtual) type assignment. We shall also consider this issue in the sequel and present conditions under which a unique type assignment is guaranteed.

2.5 Schemas and Grammars

Real schemas associate types with regular grammars. It turns out that the extension of schemas to include virtual types, with the refined notion of conformity, extend the expressive power of the language. Denote by CF-ScmDL a variant of ScmDL, also without virtual types, where types can be described by context free grammars, rather than by regular expressions, and similarly use VCF-ScmDL for such schemas with virtual types. The semantics is defined exactly as in Definitions 3 and 6, except that now $Lang(t)$ is in general a context free language.

Proposition 2. *Every* VScmDL *schema has an equivalent CF-*ScmDL *schema. Conversely, every CF-*ScmDL *schema has an equivalent* VScmDL *schema, but not necessarily an equivalent* ScmDL *schema.*

The intuition here is based on the analogy between virtual nodes of a graph and internal nodes of derivation trees for CF grammars. Thus the extension of ScmDL to VScmDL adds expressive power. In contrast, enriching CF-ScmDL with virtual types does not extend the expressive power.

Proposition 3. *Every VCF-*ScmDL *schema has an equivalent CF-*ScmDL *schema. Moreover, every (V)CF-*ScmDL *schema has an equivalent CF-*ScmDL *Schema where all unordered types are associated with regular expressions.*

An immediate consequence of the second part of the Proposition (proved using Parikh's theorem[14]) is that Proposition 2 can now be refined: *"Every* VScmDL *schema has an equivalent CF-*ScmDL *where all unordered types are associated with regular expressions".* Thus the use of virtual types adds expressive power only in describing ordered types.

3 Redundant types

Since schemas allow the description of optional data, a type assignment for a given data graph may not use all types in a schema, and type assignments to different graphs may use distinct types. However, it may be the case that a schema contains some types that are redundant — they are not used by any type assignment. For example, consider the following schema.

$$\begin{array}{rll} r = & t \mid q & (label = Dom, root) \\ q = & \epsilon & (label = Dom) \\ t = & t & (label = Dom) \end{array}$$

It is easy to see that it has only one possible instance having a root of type r with one child of type q. The other alternative, namely having a child of type t, is impossible since t requires a child of type t, and so on till infinitum. Such an infinite chain of nodes of type t can exist in a finite graph only in a cycle in which an edge points back to a previous t node in the chain. But nodes of type t are not referencable, hence this is not allowed. It follows that t is redundant.

Formally, a type t is *redundant* in a schema S if there is no instance G of S having a virtual type assignment that uses t. Clearly one would like to avoid having redundant types in a schema - they make the schema larger than necessary, hence more complex to understand and handle. We prove that type redundancy can be treated and eliminated efficiently, hence from now we assume that schemas contain no redundant types.

Theorem 1. *Every* (V)ScmDL *schema S can be transformed to an equivalent* (V)ScmDL *schema S' with no redundant types, in time polynomial in the size of S.*

Corollary 1. *Given a schema S, it is possible to decide if $inst(S)$ is empty, in time polynomial in the size of S.*

4 Closure properties

In data integration one imports data from multiple sources, each described by its own schema, and it is often desirable to describe all data using a single schema. For simple scenarios, this requires performing on schemas operations that reflect the basic set theoretic operations on the data. We consider the following two questions: Given two VScmDL schemas S_1, S_2, does there always exist a VScmDL schema S_{op}, such that

1. $inst(S_{op}) = inst(S_1)$ $\mathtt{op}$ $inst(S_2)$, where $\mathtt{op}$ is one of $\cup, \cap, -$?
2. $v\_inst(S^v_{op}) = v\_inst(S_1)$ $\mathtt{op}$ $v\_inst(S_2)$, for $\mathtt{op}$ in $\cup, \cap, -$?

We will refer to the first as closure under *op* and to the later as closure under *virtual op*. We note that closure under (virtual) difference implies closure under (virtual) complement since we have shown that ScmDL can define a schema with all data graphs as its instances (and a similar VScmDL schema can be defined for virtual instances). Also, since by Proposition 1 one can test emptiness of schemas, computable closure under difference implies that testing for the equivalence and the containment of schemas (hence schema subsumption) is possible.

Proposition 4. .
(1) VScmDL *schemas are closed under union, but not under intersection, subtraction and complement. If one of them is* real, *then they are closed under intersection.*
(2) ScmDL *schemas (i.e. schemas with no virtual types) are closed under all these operations.*
(3) Finally, all schemas are closed under virtual *union, intersection, subtraction and complement.*

Note that in here we need our predicate language to be closed under boolean operations. Also, observe that for ordered types the results follow from closure properties of context-free and regular languages, but for unordered types we use the theory of semi-linear sets [14].

Initial results regarding schema closure under more general query operations are presented below.

5 Matching Data and Schema

Given a (V)ScmDL schema S, we call the problem of testing if a data graph G conforms to it the *(v-)matching problem.* Finding a corresponding virtual version (for the case of VScmDL schemas) and the (virtual) type assignment, if one exists, is called the *type derivation* problem. We are interested in the complexity of the problems as a function of the size of the input data graph G.

Theorem 2. *The matching, v-matching and type derivation problems are NP-complete. The problems remain NP-complete even if the nodes in the graph are all ordered (or all unordered).*

Proof. (sketch) For the upper bound, guess a type assignment for the nodes. Testing the local properties for each node is easy. For the relationship with its children, convert a VScmDL into a CF-ScmDL schema. Then, for each node, use the parse tree of the word for deriving the virtual nodes and their assigned types. (For real schemas, this step is simpler). Completeness is proved via reduction to satisfiability of 3NF formulas.

Despite this result, in many practical cases the above problems can be solved in polynomial time. We present some examples below.

Proposition 5. *If the data graph is a tree then the (v-)matching and type derivation problems for it can be solved in a polynomial time.*

Proof. (sketch) We determine all possible type assignments for each node, going up from the leaves. For unordered nodes we use dynamic programming to avoid considering all the possible orders of the edges.

Non-tree data can also be handled efficiently in some cases. One case is the class of schema-graphs of [5], which, as shown in the proof of Proposition 1, have very simple syntactic characterization in ScmDL. Following [5] we have that the matching and the type derivation problems for this class can be solved in polynomial time (details omitted). Next, we consider in more detail another case:

Definition 7. *We say that an CF-*ScmDL *schema S is* **tagged** *if for every type name t in S, for any two types t_1, t_2 in $Lang(t)$, their label predicates are disjoint (i.e. cannot be satisfied by the same label value).*

A VScmDL *schema is* **tagged**, *if its equivalent CF-*ScmDL *schema S', as constructed in the proof of Proposition 2, is tagged.*

Observe that relational and object oriented databases, and in general strongly typed databases, with homogeneous bulk types and tagged union are all naturally described by tagged (real) schemas.

Proposition 6. *For tagged schemas the (v-)matching and the type inference problems can be solved in polynomial time.*

Proof. (sketch) Here the algorithm works from the root down to the leaves. Again, unordered nodes pose a difficulty, since one needs to avoid considering all the possible orders for the set of children, which is addressed using dynamic programming.

Conformity of a data graph to a schema implies there is at least one type assignment h for its nodes, but there may exist more than one assignment. From the proof of Proposition 6 we have:

Proposition 7. *Instances of tagged real schemas have a single type assignment.*

Given a VScmDLschema S, we call the problem of testing if every instance of S has a unique (virtual) type assignment, the *unique (virtual) assignment* problem. Observe that having a unique virtual assignment always implies having a unique type assignment, but not vice versa.

Theorem 3.
(1) The unique (virtual) assignment problem is in general undecidable.
(2) The problem is decidable for real schemas.

Proof. (sketch) We prove (1) by reductions to the problem of deciding the emptiness of the intersection of two context free languages (for unique assignment), and of testing ambiguity of context-free languages (for unique virtual assignment). The proof for (2) involves intricate analysis of the possible causes for non unique typing in real schemas.

6 Schemas and Queries

Often in data integration or translation one may be interested in only a part of the data in a source, defined by a query. For example, a query in an integration system is broken into components that are pushed to the sources; only the corresponding results are translated and integrated. A natural issue that arises here is to derive the schema of a query result. This can, e.g., facilitate the translation of the retrieved data. In the opposite direction, query evaluation on a source can benefit by using schema information to prune the search. This well known idea of classical database systems is attracting attention in systems that manipulate heterogeneous and semi-structured data [16, 5, 11]. In this direction, one would like to derive schema information for node variables in a query, and use it to restrict the search.

Query languages for semi-structured data, like their classical ancestors, have a body and a head [18, 12, 11]. In the body, node variables are introduced, operated upon by predicates, and related by edges and paths. We consider here the component that seems to be the more difficult to treat, for deriving schema information, namely generalized path expressions [10, 18, 3, 6, 12], of the form $x_0\ P_1\ x_1\ P_2\ x_2 \ldots P_n\ x_n$ where the x_i's are variable names, and the P_i's are regular expressions or path variables. Intuitively, given a data graph G, such expressions search for nodes $v_0, \ldots, v_n$ s.t. the path between v_{i-1} and v_i,$i = 1 \ldots n$, matches P_i, (and v_0 is a root node). Determining the possible types for the v_i's can help both in determining the schema of the query result and in pruning the search.

Formally, we consider path expressions of the form $P = x_0\ R_1\ x_1\ \ldots R_n\ x_n$ where the x_i's are *distinct* variable names, and the R_i's are regular expressions over predicates in $\mathcal{P}$.[4] (The case where some of the variables may be identical will be considered later). Given a data graph G and a path $p = u_0 \rightarrow u_1 \rightarrow \ldots \rightarrow u_k$ in G, where the label of u_i is l_i, for $i = 0, \ldots, k$, we say that p **matches** R_i iff the language defined by R_i contains a word $w = p_0 \ldots p_k$ s.t. $p_i(l_i)$ holds, for all $i = 0, \ldots, k$. We say that the nodes $v_0, v_1, \ldots, v_n$ that lie on a path in a data graph G **satisfy** the generalized path expression $P = x_0\ R_1\ x_1\ \ldots R_n\ x_n$, if v_0 is a root node, and for all $i = 1 \ldots n$, there exist a path from v_{i-1} to v_i that matches R_i.

Now, we say that the vector of types $t_0, t_1, \ldots, t_n$ in a VScmDL schema S is a **possible type assignment** for $x_0, \ldots, x_n$, respectively, if there exists a data graph G that conforms to S w.r.t a type assignment h, and nodes $v_0, v_1, \ldots, v_n \in G$ satisfying P, s.t. $h(v_i) = t_i$, $i = 0 \ldots n$. Each such possible type assignment describes the types of objects in some occurrence of the path expression. Given a schema S and a generalized path expression P, our goal is to find the set of all possible type assignments for the variables in P. We note that the size of the answer may be large, compared to the schema: if the schema is very loose, it may be the case that each of the variables can be associated with most of the types in the schema, so the size of the answer can be up to $O(|P|^{|S|})$, i.e. exponential in the size of the schema. So rather than measuring the complexity of the type computation only in terms of the size of the schema and the query, it is useful to also take the size of the answer into consideration.

[4] Note that these regular expressions are different than the ones used in our schema language and which are defined over types in $\mathcal{T}$ and not over predicates.

Theorem 4. *Given a schema S and a path expression P, the possible type assignments for the variables in P can be computed in time polynomial in the size of S, P, and the number of possible type assignments $+1$.*[5]

Proof. (sketch) To prove the theorem we first show that for every two types t', t'' in S and every regular expression R_i, it is possible to define (in polynomial time) a regular language describing all the possible paths that start at a node of type t', end at a node of type t'', and match R_i. Then we use these languages to compute the possible type assignments.

Next, we consider how computing the possible types of query variables may be useful for optimizing the query evaluation, by allowing to prune the search space.

Definition 8. *Given a schema S, two types t', t'' in S, and a regular expression R over $\mathcal{P}$, we say that a type t in S is* **useful** *w.r.t t', t'', R, if there exists a data graph G that conforms to S with a type assignment h, and a path $p = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$ in G that matches R, s.t. $h(v_0) = t'$, $h(v_n) = t''$, and $h(v_i) = t$ for some $0 < i < n$.*

Theorem 5. *For every t', t'', R, the set of useful types w.r.t t', t'', P can be computed in time polynomial in the size S and R.*

At each step in the computation of a generalized path expression P, one holds some node v_i (initially the root node v_0) and searches for paths that match a path expression R_{i+1}, looking for the nodes v_{i+i} at the end of such a path. If we know the type assignment for nodes, and in particular the type t' of v_i, then we can prune the search whenever we run into a node of type t that is not useful w.r.t t', R_{i+1}, and any of the types t'' in S.

Moreover, if we pre-compute the possibly type assignments for P, then we can do even better: Given the type of a current node (and the types of the nodes assigned to previous variables in P), we can check what are the possible type assignments for the next variable, and prune all paths going through non-useful types w.r.t these possible assignments.

Remark: When variables may occur several times in a path, we can remove from the type assignments all those vectors in which different occurrences of the same variable have been assigned distinct types. Note however that this still does not guarantee that all the remaining vectors are actually possible assignments, i.e. that there exists an instance with a path where the two occurrences of the variable have been assigned the same *node*, and not just the same *type*. Selecting the appropriate vectors in this case is still an open problem.

7 Conclusion

We have presented two schema definition languages, and illustrated their expressive power. In particular, we presented motivation for augmenting the basic

[5] The "+1" is for the extreme case where there aren't any.

language with the notions of virtual nodes and types, and showed these extend the expressive power. We also investigated the complexity of significant problems regarding schemas and instances, showing that in many cases they can be solved efficiently. Hence, we believe that our schema languages, particularly ScmDL, present a good combination of expressibility and tractability, and that our approach and results can be used to enhance the usability of data translation and integration systems, as illustrated in the TranScm system.

As we mentioned, there has been recent work on schematic description of semistructured data [5, 16, 4, 7]. However, these works focus more on describing patterns of paths in the data than on precisely describing its structure. For example, they can specify that edges with a given property going out of a node may exist, but they cannot *require* their occurrence, which can easily be done in our language. Thus, our work is much closer to classical approaches to schemas and types.

A significant recent work that presents a comparable approach is YAT [11]. They have order in their data and schemas, (but there *all* the nodes are ordered, which is less natural when modeling unordered components like sets). More importantly, it can be seen that their schema language essentially uses regular expressions, and is thus similar to ScmDL, but they do not support virtual types. While their *instantiation* mechanism provides a notion matching of data to a schema, it does not provide an explicit type assignment to instance nodes as our formalism does. But it offers subtyping that we do not treat here. Their notion seems to differ from ours in some subtle points, e.g., in the treatment of the combination of $*, |$ in regular expressions. Such differences may impact some of the complexity results we have obtained for our model. While they mention matching of data to schemas, and inferring the schemas for queries (although without regular paths), they do not mention complexity results. Further study is needed to clarify the differences, and to investigate the impact on the complexity of significant problems.

References

1. Extensible markup language, 1998. Available by from `http://www.w3.org/XML/`.
2. S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proc. ICDT 97*, pages 351–363, 1997.
3. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1), 1997.
4. S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. Symp. on Principles of Database Systems – PODS 97*, 1997.
5. P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proc. Int. Conf. on Database Theory ICDT 97*, 1997.
6. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of SIGMOD '96*, pages 505–516, 1996.
7. P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *Proceedings of PODS '98*, pages 129–138, 1998.

8. M.J. Carey et al. Towards heterogeneous multimedia information systems : The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.
9. T.-P. Chang and R. Hull. Using witness generators to support bi-directional update between object-based databases. In *Proc. Symp. on Principles of Database Systems – PODS 95*, San Jose, California, May 1995.
10. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. ACM SIGMOD Symp. on the Management of Data, 94*, pages 313–324, 1994.
11. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *SIGMOD'98, to appear*, 1998.
12. M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 6(3):4–11, 1997.
13. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. In *Journal of Intelligent Information Systems*, 1997.
14. S. Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, 1966.
15. C.F. Goldfarb. *The SGML Handbook*. Calendon Press, Oxford, 1990.
16. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of Conf. on Very Large Data Bases, VLDB '97*, 1997.
17. A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of Conf. on Very Large Data Bases, VLDB '96*, 1996.
18. A. Mendelzon, G. Michaila, and T. Milo. Querying the world wide web. *Int. Journal of Digital Libraries*, 1(1), 1997.
19. T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *To appear in VLDB '98*, 1998.
20. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE International Conference on Data Engineering 95*, 1995.