VAASAN YLIOPISTO

JARI TÖYLI

AdSchema – a Schema for Semistructured Data

ACTA WASAENSIA

No. 157

Computer Science 5

UNIVERSITAS WASAENSIS 2006

Reviewers Professor Eljas Soisalon-Soininen
Helsinki University of Technology
Department of Computer Science and Engineering
P.O. Box 5400, FI-02015 HUT

Professor Martti Penttonen
University of Kuopio
Department of Computer Science
P.O.Box 1627, FI-70211 Kuopio

ACKNOWLEDGEMENTS

The development of this thesis was a long process during which I have met many extraordinary people. Although in the cover of this thesis is only one name, many are the names of people who, one way or other nurtured it, and to whom I am pleased to devote my most sincere and deep gratefulness.

At first, I would like to thank my supervisor Professor Matti Linna, for his great support and wise guidance throughout the process, steering but without imposing, suggesting but keeping freedom intact. I would also like to thank Professor Merja Wanne for the insightful discussions we had and her indispensable help at the beginning of this work. I also like to thank Professor Jarmo Alander for his support and thoughtful feedback of this work. I would also like to thank Professor Martti Penttonen and Professor Eljas Soisalon-Soininen for reviewing the thesis and their valuable comments and remarks concerning the work.

With gratefulness, I would like to acknowledge here, that the research work behind this thesis was financed by Research Institute for Technology and Vaasan yliopistoseura.

I would like to thank all my colleagues who have supported or contributed to this work by encouragement, advice, suggestions, discussions, etc. especially Anja, Pekka, Hannu K., Kimmo, Markus, Timo, Johanna and many others.

Finally, my deepest gratitude to my loving wife Tuula who has loved me and encouraged me all these years and to our children Jani, Hannakatrin, Ville, Elina-Sofia and Johannes for being there.

CONTENTS

ABSTRACT

Töyli, Jari (2005). AdSchema – a Schema for Semistructured Data. *Acta Wasaensia* No. 157, 85 p.

Data that do not have a fixed schema cannot be searched efficiently because the forming of a query is difficult without a schema or some information about the structure. The quantity of such data has increased since the middle of the 90's, especially on the Internet. There is much research conducted in this area and consequently there are several proposals of how to model such data, how to search such data and also how to present the structure of such data. Semistructured data, as it is called, is still a subject of intensive investigations.

In our investigations we have found out that schemas of such data still need enhancements. It has turned out that the schemas can be bigger than the original source database or, if the size is limited, some accuracy is lost. In this work we first propose a schema to be used to present the structure of such data and secondly an algorithm with which we can construct such a schema. Our earlier investigations showed that we can use the theory of Adjacency Relation Systems to model semistructured data and also that the same theory can be used, with some minor modifications, as a foundation for a schema for semistructured data.

In this work we apply the theory of Adjacency Relation Systems and propose a new way to present the schema of semistructured data. This approach differs in some parts substantially from those presented by other researchers. Our proposal is not based on merging of similar parts of the schema but on the interrelationships between adjacent types. By this technique we can prevent the schema from growing too large and avoid loosing of accuracy of the schema. Our proposal has also the advantage that the produced schema has the same kind of characteristics as a strong DataGuide.

Our work gives also an accurate definition of semistructured data and introduces an algorithm for calculating the degree of semistructured data. Moreover, we present a general query by which we can search data from such data structure.

Jari Töyli, Department of Computer Science, University of Vaasa, P.O. Box 700, FI-65101 Vaasa, Finland, email jari.toyli@uwasa.fi.

Keywords: Semistructured Data, DataGuide, Schema Construction, Algorithms, Interrelationship, Degree of Semistructured Data.

1. INTRODUCTION

1.1. Background of the Research

In structured database systems the schema has two purposes. First it describes the structure or type of the data and secondly it describes some constraints on the information system. This type of data can easily be represented and accessed by conventional database systems, like relational and object oriented database systems. However, today there exist more and more data that cannot be constrained by a schema because the data has variations in its structure. The variations typically consist of missing data, duplicated fields, or minor changes in representation. This kind of data arises among others on the Internet, and it is called semistructured data.

Semistructured data (Abiteboul 1997; Buneman 1997; Abiteboul, Buneman & Suciu 2000) has been investigated since the middle of the 90's. During that time the research has mainly focused on developing data models and query languages for semistructured data. There is one proposal that has gained more attention than any other. This model, based on objects, is called Object Exchange Model (OEM) (McHugh, Abiteboul, Goldman, Quass & Widom 1997). In another model, proposed by Buneman, Davidson, Hillebrand & Suciu (1996) and Buneman, Fernandez & Suciu (2000), the node labels are absent and the data is carried entirely on the edge labels. This model has not been named, although it is based on a sound theory.

In addition to these two data models there are also other proposals that have a close affinity to these models. The first one, ACeDB (Thierry-Mieg & Durbin, 1992) has been developed as a database for genetic data and the second, is the XML (World Wide Web Consortium, 2004). There is a great deal of research on

XML in the context of semistructured data, e.g. Goldman, McHugh & Widom (1999) and Broekstra, Fluit, & van Harmelen (2000).

Because of the absence of a fixed schema, conventional SQL cannot be used to access semistructured data. However, there are new query languages developed for semistructured data. Some of them are meant to be used with XML data (Broekstra et al. 2000), and others with object data, like *Lorel*. Lorel has been developed in the context with the Lore system, and it is based on the object query language (OQL). Still another language is called *UnQL*, and it is a value based query language (Buneman et al. 2000).

The main purpose of this work is to present a new proposal for a schema of semistructured data and an algorithm for its construction. Besides, this work contains a definition of semistructured data, and an algorithm for defining the degree of semistructured data. We present also a general query for semistructured data, which includes all the special queries presented in Wanne (1998).

Schema proposals for semistructured data are one of the research areas which have been widely investigated during the last decade. DataGuide (Goldman, Widom 1997; Goldman 2000) is the most promising proposal of a schema for semistructured data, but as it has some drawback, the research on that subject is still active. There are also other proposals e.g. Wang & Liu (1997) and Nestorov, Abiteboul & Motwani (1998) of how to discover the inherent structure of semistructured data, but these proposals are all in a stage of development, and so far there is no prominent solution to the problem.

The drawbacks in the proposals of a schema for semistructured data represented so far are significant to the extent that there is a need of a new proposal. The following problems are the most distinctive ones:

- There can be more than one schema for a semistructured database.
- Some accuracy of the schema can be lost when one generates an approximate schema (i.e. merge similar parts) of the source database.
- The size of the schema may be bigger than the size of the original database.

As long as these problems are unsolved the queries on semistructured data are inefficient, which in turn means higher costs. Researches have developed new query languages that use novel techniques and there are also more efficient methods to discover the embedded structure of the data from the underlying structure. However, the results are still unsatisfactory in some areas. In our work we focus on the latter technique and propose a method in which we can complete the underlying data structure with new relations that are derived from the known relations. The completion of the data structure applies the idea of relations between adjacent elements, and adjacency defining sets (Wanne 1998, Wanne & Linna 1999).

1.2. Research Objectives and Contributions

According to the research conducted on schema discovery and extraction we conclude that schema development is a vital area that is worth to be explored more. Our research hypotheses are the following:

> A schema for semistructured data can be constructed and completed according to the theories of the adjacency relation system.
>
> The size of the present schemas for semistructured data can be reduced and its accuracy can be increased with the help of the Adjacency Schema.

In order to test our hypothesis we state some research questions which will guide our work. There are four research questions that are the following:

1. Is it possible to apply the theory of the adjacency relation system to construct a schema on semistructured data?
2. Can we develop an algorithm with which the schema can be built up?
3. Does the Adjacency Schema fulfil the same requirements imposed on the present schemas for semistructured data?
4. Is the Adjacency Schema minimal and more accurate than the existing schemas?

The results of Töyli (2002) and Töyli, Wanne & Linna (2002a, 2002b) showed that semistructured data can be presented with the adjacency model. In this work we will go further ahead in our investigations and propose a schema for semistructured data. Our further research is based on the theory of the adjacency relation system (Wanne 1998), which we have somewhat modified, as well as the theory of representative objects (Nestorov, Ullman, Wiener & Chawathe 1997a).

According to the hypothesis and research questions stated above, the main contributions of this work can be stated as follows:

1. A new schema proposal for semistructured data.
2. An accurate definition of semistructured data.
3. An algorithm for calculating the degree of an incomplete data structure.
4. A general query for a given relation combination.
5. An algorithm for modelling semistructured data.
6. An algorithm for modelling relational data.
7. An algorithm for the construction of a schema for semistructured data.

1.3. Structure of Thesis

The rest of this thesis is organized as follows. Section 2 is a literature review, in which we present the most recent research findings in the area of this study. The literature review is restricted to cover the areas of schema extraction and discovery, and the most significant papers written on schemas of semistructured data until today.

In Section 3 we present the theories of representative objects and DataGuide, which set down the ground for a schema of semistructured data. We also give a short presentation of other research conducted in this area concerning structure discovering.

In Section 4 we present the theory of Adjacency Relation Systems (ARS), which has been the basis when modelling semistructured data (Töyli 2002), and also in this case when we present our proposal of a schema for semistructured data. This theory is fundamental to our work, although we have made some minor changes to some of the definitions.

In Section 5 we present our schema proposal for semistructured data. First we give the definition of semistructured data, an algorithm for determining the degree of a data structure, and define a general query. After that we present two algorithms for the modelling of relational and semistructured data and one for the construction of a schema for semistructured data. We also introduce the concept of an Adjacency Schema (AdSchema) and give some examples.

Finally, in Section 6 we give some conclusions of the results, and outline some proposals of future work.

2. DISCOVERING THE DATA STRUCTURE

Schema discovery, especially in context with semistructured data, has been under intensive investigations during the last few years. In order to get some understanding of the research conducted in this area we present some work that deal with this specific subject. There is much research that does not propose any new schema, but which is very important for the development of the schema construction.

Actually, there is a proposition called DataGuide (Goldman et al. 1997) which can be used as a schema for semistructured data. It is a directed graph, where every edge of the graph is represented with a formula. However, DataGuide has been developed since then and a more accurate version of it has been introduced. The most significant problems in context with a schema for semistructured data are the size and the accuracy of the schema. The more accurate schemas we want the larger the schema becomes and vice versa. We have to balance between these two aspects.

In addition to DataGuide there are other propositions in which the researchers try to discover (Wang et al., 1997; Wang & Liu, 1998; Cong, Yi, Liu & Wang, 2002; Wang & Liu, 2000b) or extract (Wang, Yu & Wong, 2000; Hacid, Soualmia & Toumani, 2000; Nestorov et al., 1998) schema information from the data, or more generally, they try to discover a typical or frequent structure on the data. The data will then be classified or clustered into sets of objects. The results of these investigations are promising, but more research is needed because the accuracy and conciseness of the schema is still incomplete.

All the above-mentioned papers consider object models like Object Exchange Model (OEM). The same idea and model have been considered in Nestorov et al. (1997a) in which the authors present the concept of *representative objects* that facilitates the querying and browsing of semistructured data. In Nestorov et al. (1997b) the authors consider the problem of identifying some underlying structure in large collections of semistructured data.

Buneman et al. (1997) present a proposal of a schema for semistructured data in which both data and schema are represented as edge-labeled graphs. It differs from the OEM model in that the labels are associated with edges, not nodes. The notion of conformance between a graph database and a graph schema are introduced. Also a "deterministic" subclass of schemas is investigated.

In Wang et al. (1997) the authors consider schema discovery. The basic idea is to generate a structured layer of typical object structures of semistructured data (e.g. of an OEM database). The main concept in this process is a so-called *tree-expression*, which is a partial (labelled) tree representation of an object, i.e. a generalization of an object. When generalizing an object some information may be lost. The generalization problem is considered in Wang et al., (2000b).

The concept of the tree expression is further developed in Wang et al. (1998a) and Wang et al. (2000a), where the authors represent a new concept called a *k-tree-expression*. A *k*-tree expression is a tree expression of *k* leaf nodes, and it is constructed by "cluing" a sequence of *k* paths that are not prefixes of each other. In other words, a *k*-tree expression is the "prefix tree" of the *k* paths that preserves their order in the sequence. The authors also present an algorithm for searching all *k*-tree expressions. The simple paths of *k*-tree expressions are then traversed in depth-first order (Goodrich & Tamassia 1998).

In Wang et al. (1998a) and Wang et al. (2000a) the authors noticed that the use of wildcards on paths can cause over generation of them, and also that the algorithm cannot discover typical substructures which exist in the lower part or in the middle part of semistructured objects when the upper part of objects has a different number of labels. The authors in Cong et al. (2002) propose a new and more effective method to compute the frequencies of the substructures (with a wild card or not). This approach contains a new feature in which the semistructured objects are represented with paths and corresponding *tidlists*, where the tidlist is a set of transaction objects or trees that contain a path as a tuple in the database. Representation of objects with labelled paths saves a lot of space. The authors also propose an effective method by which the over generation of paths with wild cards can be removed. Finally, they propose an adapted mining algorithm for the substructure discovery problem.

The algorithm proposed in Cong et al. (2002) is more effective than the one developed and improved in Wang et al., (1997, 1998a, 2000a). However, as long as wildcards are used we must balance between efficiency and accuracy. The number of wildcards affects restrictively to accuracy and increasingly to the search space, so we would need a method that does not use wildcards at all, or at least the number of them is minor.

The same problem of schema extraction has been considered in Wang et al. (2000b) and in Nestorov et al. (1998) but this time the authors have used a slightly different method. In Wang et al. (2000b) the method has been to use description logics and in Nestorov et al. (1998) the method is based on the semantics of monadic datalog. However, both of these methods use the same approach of merging of similar object classes. These classes proved to be too large so the authors in both papers propose approximation of the generated groups. Here we give only a short presentation of the subject, but a more detailed description of the methods can be found in Wang et al. (2000b) and in Nestorov et al. (1998). As a motivation for our work we can mention that the authors faced the same kind of problem as we have encountered before, i.e. inaccuracy when grouping or merging objects.

In Wang et al. (2000b) the authors propose a construction of an approximated graph schema by clustering objects with similar incoming and outgoing edge patterns. This approach differs from the DataGuide in that the DataGuide combines objects with similar type definitions. The main idea is to predict the sets of incoming and outgoing edges of a vertex of a data graph that also appears as a vertex of the schema graph. Another principle is to minimize the size of the schema graph by minimizing the number of appearances of each label.

The approximate graph schema is small in size, cheap to construct and maintain, and there is no predetermined user defined threshold. However, there may be duplicates and false paths in the schema graph, which means that it is not fully accurate.

In Nestorov et al. (1997a) the authors investigated the typing problem of semistructured data. This problem will be considered more detailed in Section 3 and is bypassed here.

The idea of a concise representation of the inherent schema of semistructured data has been refined in Nestorov et al. (1997b). The authors present an algorithm for constructing a type hierarchy for semistructured data source. The classification is highly dependent on the choice of the so-called *threshold* value. If the threshold is too low, the result set is too large and on the other hand if the threshold is too high, the accuracy becomes very low. Once again, there is a trade-off between the accuracy and the size. It is also problematic to choose the value of the threshold.

In Nestorov et al. (1998) the authors present an algorithm for approximate typing of semistructured data, i.e. an object does not have to fit its type definition precisely. The size of perfect typing was a problem in Nestorov et al.

(1997b), because the resulting data sets may be imperfect with respect to the typing, i.e. the data sets may contain edges that should not be present or lack some information (missing edges). Nestorov et al. (1998) presents a technique for computing an approximate typing of an appropriate size, which allows the data set to be imperfect with respect to the typing and they focus on the quality of the typing result rather than the time performance. In Nestorov et al. (1998) the authors have used the semantics of monadic datalog programs. Another approach, presented in Hacid et al. (2000), use description logics for discovering the (partial) implicit structure of semistructured data. They propose an approach for approximate typing of semistructured data. The extracted schema is expressed in the form of terminological axioms in a small description logic. A more detailed description of the approach is given in Hacid et al. (2000).

Constraints on semistructured data are not easy to express. In Buneman et al. (1997) there is a proposal of how to present schema for such data. In that proposal the database is presented as an edge-labeled graph, with a corresponding schema in which the edges are labeled with unary formulas. A database conforms to a schema if there is a correspondence between the edges in the source database and the schema. A graph database may conform to several graph schemas. So there exists a schema to which all databases conform. It is also possible to store multiple schemas for the same data, in order to help optimize queries. Since there is a natural ordering on graphs schemas, it is possible to take the *least upper bound* of a set of schemas and combine all their constraints into a single schema. This approach is one of the first proposals for a schema for semistructured data, and has had a very strong implication in the research of schemas for semistructured data.

3. FOUNDATIONAL THEORIES

In this section we present the theories behind representative objects (Nestorov et al. 1997a) and DataGuides (Goldman et al. 1997, Goldman 2000). We consider here the theory of representative objects because it lays the foundation for the DataGuide. The DataGuide, in turn, is considered because it is the de facto standard of schemas for semistructured data and as far as we know, the only implementation of a schema for semistructured data. In Figure 3.1 we have a part of an OEM database, representing a premiership document. It serves as a basis for multiple examples through this section.

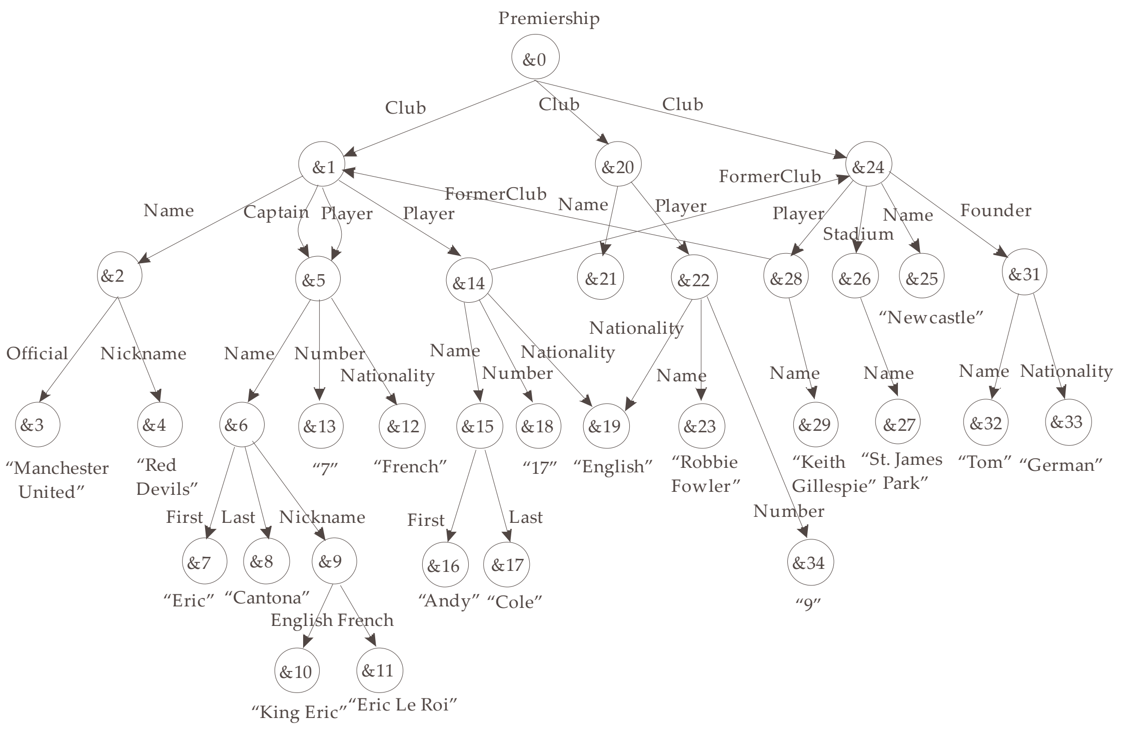


Figure 3.1. The premiership object (Nestorov et al. 1997a).

3.1. Representative Objects

The theoretical foundation of dynamically generated structural summaries has been given in (Nestorov et al. 1997a). The authors present the *representative object* concept, which is also called as a *DataGuide*. The motivation behind a representative object is not only to create a concise description of the structure of the data, but also to facilitate querying and browsing of semistructured data.

Representative objects are used with an object model called *Object Exchange Model* (OEM), initially introduced in Papakonstantinou et al. (1995). In an OEM model every node represents an object identification, and the edges with their labels represent object references. One important concept used with the OEM model is a *simple path expression* that is a dot separated sequence of labels of the graph (Nestorov et al. 1997a).

Definition 3.1. Let l_i be a label (of object references) for $i = 1, \ldots, n$, $n \geq 0$. Then $pe = l_1 \cdot l_2 \cdots l_n$ is a *simple path expression* of length n.

Another concept is a data path that is a sequence of alternating objects and labels, separated by commas. A data path starts and ends with an object. For every two consecutive objects the first object has an object reference to the second object and there is a label between the objects (Nestorov et al. 1997a).

Definition 3.2. Let o_i be an object for $i = 0, \ldots, n$, l_i be a label for $i = 1, \ldots, n$, and $\langle l_i, identifier(o_i) \rangle \in value(o_{i-1})$ for $i = 1, \ldots, n$, $n \geq 0$. Then $p = o_0, l_1, o_1, l_2, \cdots, l_n, o_n$ is a *data path*, of length n.

Here are some additional terms in order to help to understand the concepts of simple path expressions and data paths:

- A data path $p = o_0, l_1, \cdots, l_n, o_n$ *originates from* o_0 or is *rooted* at the object o_0.
- An object o_i is *within* an object o if there exists a data path originating from o and ending with o_i.
- A data path p is *within* an object o if p originates from an object within o.
- A data path $p = o_0, l_1, \cdots, l_n, o_n$ is an *instance of* the simple path expression $pe = l_1.l_2.\cdots.l_n$.

Example 3.1. To illustrate the above terms consider the *premiership* object of Figure 3.1.

- The simple path expression *Player.Number* has two instance data paths within the premiership object, namely *obj(&1),Player,obj(&5), Number.obj(&13)* and *obj(&1),Player,obj(&14),Number,obj(&18)*.
- Consider the following two data paths *obj(&1),Player,obj(&14). FormerClub,obj(&24)* and *obj(&24),Player.obj(&28),FormerClub,obj(&1)*. Thus *obj(&1)* is within *obj(&24)* and *obj(&24)* is within *obj(&1)*. i.e. there is a cycle within the premiership object.

With a continuation function represented in Nestorov et al. (1997a) we can find out the labels of the object references on a given path, and we can also find out an answer if we can continue our navigation from the current node, i.e. the node is not an atomic node. The continuation function is substantial for representative objects.

Definition 3.3. Let o be an object in OEM and $pe = l_1.l_2.\cdots.l_n$ a simple path expression, $n \geq 0$. Then we define the concept *continuation(o,pe)* as follows.

- *continuation(o,pe)* $\supseteq \{ l \mid \exists$ a data path $p = o, l_1, o_1, \cdots, l_n, o_n, l, o_{n+1}$ that is an instance of *pe.l*$\}$.
- *continuation(o,pe)* $\supseteq \{ \perp \mid \exists$ a data path $p = o, l_1, o_1, \cdots, l_n, o_n$ that is an instance of *pe* and o_n is an atomic object$\}$

If the simple path expression is of length 0, then the continuation of o is the set of the labels of all outgoing edges from o. However, if the simple path expression is pe, then the continuation of o is the set of all the labels of the outgoing edges of the last node of the path of length $n+1$ starting at o plus $\perp$ if any of the last nodes is atomic.

Example 3.2. Consider the *premiership* object from Figure 3.1. The following examples illustrate Definition 3.3.

- *continuation(premiership,* ε *) = {Club}*
- *continuation(premiership, Club) = {Name, Player, Stadium, Captain}*
- *continuation(premiership, Club.Player.Name) = {First, Last, Nickname,* $\perp$ *}*

In Definition 3.3 we consider only data paths originating from an object that is the first argument of the continuation function. There are also situations when it is convenient to relax this restriction by e.g. limiting the length of the simple path expression or by allowing the data path to be within a given object.

Definition 3.4. Let o be an object, $k \geq 1$, and let pe be a simple path expression of length n, $0 \leq n \leq k$. Then we define *continuation*^{k}*(o,pe)* as follows

- If $n = k$ then
  - *continuation^k(o,pe)* $\supseteq \{ l \mid \exists$ a data path p within o, not necessarily rooted at o, that is an instance of *pe.l*$\}$
  - *continuation^k(o,pe)* $\supseteq \{ \perp \mid \exists$ a data path $p = o_0, l_1, o_1, \cdots, l_n, o_n$ within o, that is an instance of *pe and* o_n is an atomic object$\}$.
- Otherwise (*if* $n < k$) *continuation^k(o,pe)* = *continuation(o,pe).*

Example 3.3. Consider the *premiership* object in Figure 3.1. The following examples illustrate Definition 3.4.

- *Continuation^1(premiership, Name) = {Official, Nickname, First, Last,* $\perp$ *}*
- *Continuation^2(premiership, Club) = {Name, Player, Stadium, Captain}*
- *Continuation^2(premiership, Player.Name) = {First, Last, Nickname,* $\perp$ *}*

There are two types of representative objects. First, we have a *full representative object* (FRO), which is restricted to a particular object, and secondly, if the length of the simple path expression is given, then we have a *degree-k representative object* (*k*-RO). The *k*-ROs are less complex than FROs and can be used to approximate FROs. Formally, the definition is as follows.

Definition 3.5. Let o be an object. Then the function *continuation$_o$(o,pe)* = *continuation(o,pe)*, where pe is a simple path expression, is a full representative object (FRO) for o.

When the *continuation* function of Definition 3.5 is replaced by the *continuation^k* function we get the following definition for a degree-*k* representative object:

Definition 3.6. Let o be an object and $k > 1$. Then $continuation_o^k(o,pe)$ = $continuation^k(o,pe)$, where pe is a simple path expression, is a degree-*k* representative object (*k*-RO) for o.

k-ROs take usually less space than FROs and may be constructed faster. However, *k*-ROs only approximately support the schema discovery. If the length of the simple path expression pe is less than *k*, where *k* is the degree of the *k*-RO, the exact value of $continuation_o(pe)$ can be computed, and the *k*-RO provides the same support as a FRO. However, if the length of pe is at least *k* then the approximation of $continuation_o(pe)$ provided by the *k*-RO must be used. A description of the method of computing an approximation of $continuation_o(pe)$ from *k*-RO is given in Nestorov et al. (1997a).

When a FRO is implemented in an OEM graph, the $continuation(o,pe)$ is computed. The FRO consist of an object and an algorithm for computing the function $continuation_o$. The object is explored in a breadth first manner in order to find the instance data paths of pe. If every such data path only the last object in the data path is considered. Finally, we get the continuation of pe, which is the set of all the different labels of object references of those objects and possibly $\perp$ if any of those objects is atomic. The algorithm is presented in Nestorov et al. (1997a). Formally, the implementation of FROs in OEM graph is as follows:

Definition 3.7. Let o_1 and o_2 be objects in OEM. Then o_1 is a full representative object in OEM for o_2 if for any simple path expression pe we have $continuation_{o_1}(pe) = continuation_{o_2}(pe)$.

Usually there are many FROs for a given o, including the object itself. One of those FROs is minimal, i.e. there is one FRO for which the computation of the continuation function takes minimal time in proportion to the size of the object set.

Definition 3.8. Let R_o be a FRO (in OEM) for o. Then R_o is a *minimal* FRO if any simple path expression $pe = l_1.l_2.\cdots.l_n$, $n \geq 0$, has at most one instance data path originating from R_o and ending with a complex object, and at most one instance data path originating from R_o and ending with an atomic object.

In brief, the construction of a minimal FRO can be done by first creating a nondeterministic finite automaton (NFA) of a given object o, then determinizing and minimizing that NFA, and finally constructing a minimal FRO from that deterministic finite automaton (DFA).

The construction of a NFA from an object in OEM is the following: the objects of the OEM graph correspond to the states and the object references and their labels correspond to the transitions and their respective letters. There is also a function that maps all the objects within o to the corresponding unique automaton states. All the objects of the OEM are either *complex* or *atomic*. There are also two sets of which the first one contains all the objects within o, and the other one contains all the different labels of the object references within o. The determinization and minimization of the NFA is a well studied problem, so the readers are suggested to turn to Hopcroft & Ullman (1979) for a more detailed description.

After the determinization and minimization of the NFA, the next phase is to construct the minimal FRO from the DFA. In the transformation process from a DFA to an OEM object there is one major concern which is the consideration of the states that have two different associations of objects, i.e. complex and atomic objects.

The construction of a minimal FRO from the given DFA is associated with two functions, one called *atomic_obj* for mapping of a state to its corresponding atomic object, and another called *complex_obj* for mapping of a state to its corresponding complex object. The rest of the construction work can be found in Nestorow et al. (1997a). A proof of the correctness can also be found at the same source.

3.2. DataGuide

A first example of the functionality of the theory of full representative objects has been presented in a system called Lore (McHugh et al. 1997). In that system an overall description of the incomplete structure of an OEM database is presented with the help of a so-called DataGuide.

A DataGuide has the same meaning as metadata in conventional database management systems, like relational and object-oriented systems. DataGuides are stored as objects and they can be accessed by applications and query processors. Because there is no structure in the data, the DataGuide conforms to data rather than forcing data to conform to the DataGuide. The DataGuide is specially developed to be used with the Lore system. However, DataGuides can be used with any graph-based data models. Graph-structured databases and other related theoretical research is presented in Buneman et al. (1997).

The basic theory of a DataGuide is presented in Goldman et al. (1997) and Goldman (2000). There are also two other concepts, which are extensions of the primary DataGuide. These are *strong DataGuide* (Goldman 2000) and *approximate DataGuide* Goldman et al. (1999). A strong DataGuide supports annotations, i.e. a statement about the set of objects in the database reachable by a particular path. An approximate DataGuide, in turn, relaxes some aspects of the definition of a DataGuide. This inaccuracy is possible because all DataGuide paths do not need to exist in the original database.

We have already given some definitions that are crucial for a schema for an OEM database, like simple path expression and data path. In Figure 3.2 we have a sample of the OEM database of Figure 3.1. We can observe, for example, path expressions *Club.Name* and *Club.Player.Name* both starting at object *&0*.

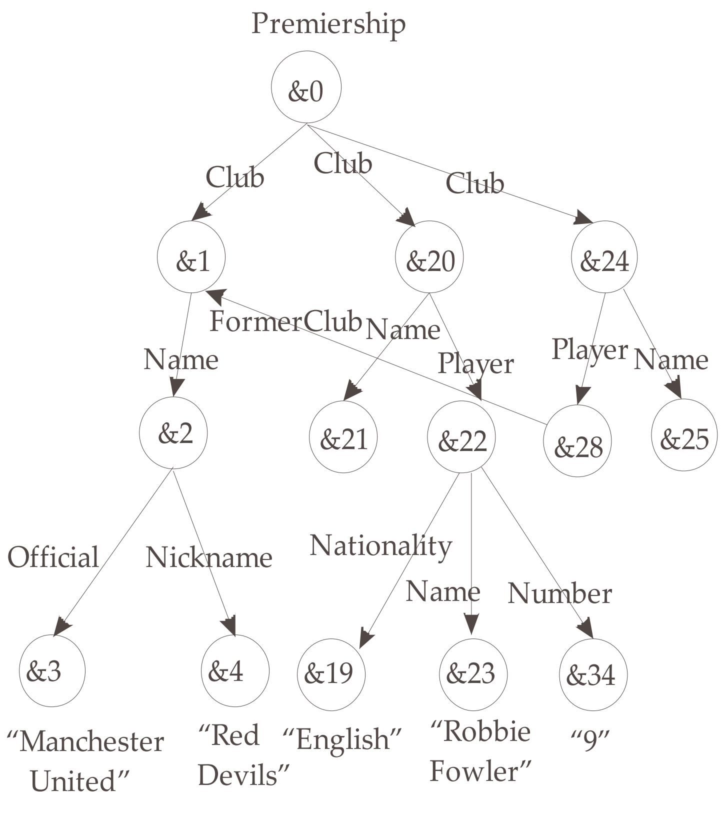


Figure 3.2. Sample of the premiership object of Figure 3.1.

In Figure 3.2 *Club.&1.Name.&2* and *Club.&20.Player.&22.Name.&23* are examples of data paths of the object &0. We can also see an instance of the data path *Club.Name*, i.e. *Club.&1.Name.&2*. For a DataGuide we can define the so-called target set:

Definition 3.9. In an OEM object s, a *target set* of a label path l is a set t of oids such that $t = \{o \mid l_1.o_1.l_2.o_2.\cdots.l_n.o \text{ is a data path instance of } l\}$. That is, a target set t is the set of all objects that can be reached by traversing a given label path l of s. We write $t = T_s(l)$. We say that l *reaches* any element of t, and likewise each element of t is *reachable* via l (Goldman 2000).

For example, the target set of *Club.Name* in Figure 3.2 is {&2, &21, &25}. Two different label paths may share the same target set. In Figure 3.2 the label paths *Club.Player.FormerClub* and *Club* have the same target set i.e. {&1}.

There are two properties to which a DataGuide must conform. The first one is *conciseness* and the second is *accuracy*. The third property of a DataGuide is that it should be *convenient*, i.e. a DataGuide must itself be an OEM object. A DataGuide is concise if every unique label path of a source database is described exactly once, regardless of the number of them in the source database. Accuracy means that there is no label path in the DataGuide that does not exist in the original source database. A database that will be summarized is called as the *source database*, and the given source database is assumed to be identified by its root object. So, the formal definition of a DataGuide is the following:

Definition 3.10. A *DataGuide* for an OEM source object s is an OEM object d such that every label path of s has exactly one data path instance in d, and every label path of d is a label path of s (Goldman 2000).

Figure 3.3 shows a minimal FRO, i.e. a DataGuide of the source database of Figure 3.1. Using this DataGuide we can check whether a given data path of length *n* exists in the original source database by examining only *n* objects from the root object in the DataGuide. In our example we need only to examine the outgoing edges of objects &35, &36 to find out that the label path *Club.Player* exists in the database. Similarly, we can follow any single instance of a label path l in a DataGuide and reach some object o, so that the labels of the outgoing edges of that particular object o represents all possible labels that could ever follow l in the source database. So the three different labels of the edges emanating from the object $\&40$, represent all possible labels that ever follow *Captain* in the source database. In Figure 3.3 there are no atomic values, because a DataGuide is a structural summary of the source database (Nestorov et al. 1997a).

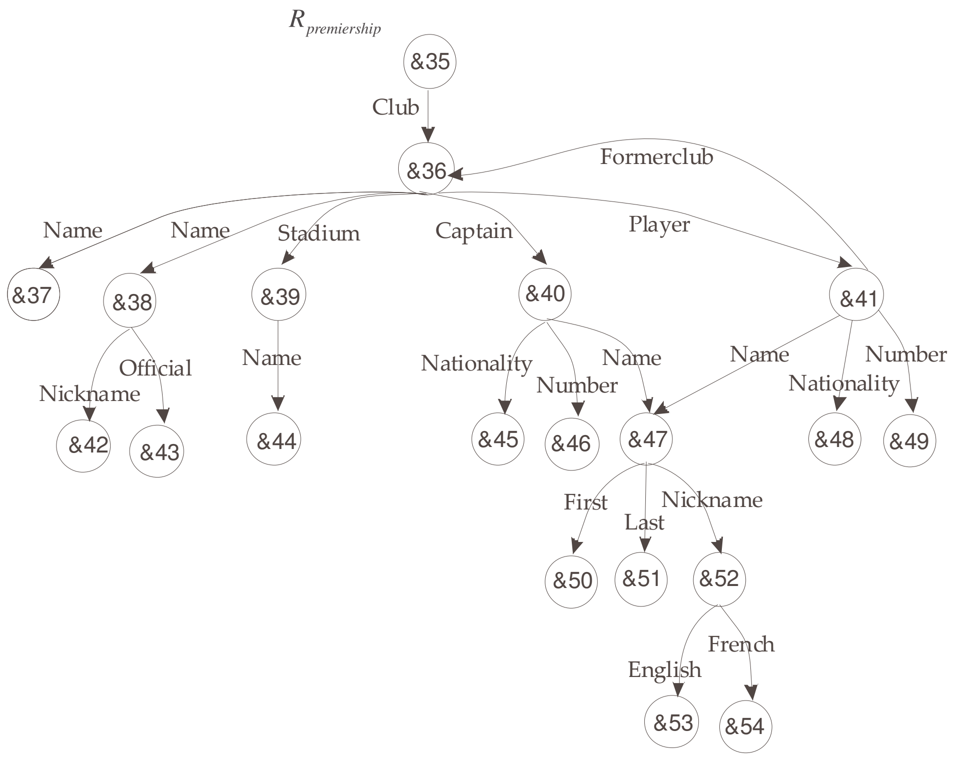


Figure 3.3. The minimal FRO (DataGuide) for the premiership object. (Nestorov et al. 1997a).

Goldman (2000) points out two main drawbacks of DataGuides. The first one is related to the annotations, i.e. a property of a set of objects that comprise the target set. The problem with annotations is that, when multiple label paths reach the same object, we cannot say to which label path the annotation is applied. This problem has been solved by *strong DataGuide* which is also introduced in Goldman (2000).

The second problem is related to the computation of a DataGuide. The computation of a DataGuide can be very expensive if there exists cycles in the structure of the source database. In Goldman et al. (1999) the authors propose a solution to this problem in a form of an *Approximate DataGuide* (ADG). An approximate DataGuide is like any other DataGuides, but it allows some degree of inaccuracy.

The idea behind an approximate DataGuide is to identify "similar" parts of the DataGuide and merge them. There are two main approaches to the approximation: *Object Matching* and *Role Matching*.

According to the original definition of a DataGuide all the data paths of the source database must be found in the corresponding schema. The approximate DataGuide relaxes this rule, i.e. there may be some paths in the schema that do not exist in the source database.

When an approximate DataGuide is created with the object matching method, (Goldman et al. 1999) the target set of the objects is considered. Every path of the DataGuide points to some target set, but in object matching we allow DataGuide paths to point to the same object when their target sets are "almost" similar. Two target sets X and Y are defined to be similar when $|X \cap Y| / \max(|X|,|Y|)$ is above a threshold θ.

Role matching is the second method for the creation of an approximate DataGuide. In this method DataGuide objects are merged based on the label paths i.e. the roles. If the function $M(p_1, p_2)$ returns *True* for label paths p_1 and

p_2, then the paths p_1 and p_2 point to the same ADG object. This merging process, in turn, can be done with two different ways. In *suffix matching* the function $M(p_1, p_2)$ returns *True* if and only if the last labels of p_1 and p_2 are equal.

The other method, called *Path-Cycle Matching*, addresses the problem caused by cyclic databases. Cycles in a database cause the paths to grow too large before reaching an identical target set. The solution for this kind of problem is to add a heuristic into the algorithm. If a specific label can be seen more than once on the path starting from the root, we can assume that this is a "semantic" cycle and the paths can be merged. The path-cycle matching function $M(p_1, p_2)$ returns *True* if and only if p_1 is a prefix of p_2 (or p_2 is a prefix of p_1) and the last labels of p_1 and p_2 are the same (Goldman et al. 1999).

The choice between a DataGuide and an approximate DataGuide is the same as if we have to choose between accuracy and efficiency. If we want to have an accurate but maybe inefficient schema we use a DataGuide, but if some inaccuracy is allowed then we choose an approximate DataGuide. It is also possible to combine some of the techniques.

4. ADAJACENCY RELATION SYSTEM

In traditional database management systems, the data is formally separated from the specification of its schema, or structure. The main purpose of the schema is to serve the query processor, but it also helps the user to understand the structure of the database. As we already know, a semistructured database does not have a schema, or if there is a schema, it is incomplete or irregular. Since a schema can prominently facilitate the query performance and help to avoid exhaustive searches, it is very important that all the information about the structure of the semistructured data can be discovered.

Schema discovery from semistructured data has been under intensive investigation since the middle of the 90's. In Section 2 we presented some examples and results of such investigations. We also noticed that there were some shortcomings, like inaccuracy after merging parts of the schema and large schemas caused by cyclic structures.

An early work of applying adjacencies has presented in Ni & Bloor (1994). Their work considers *boundary data structures*, which normally are built on relations between faces, edges and vertices. The novel idea of adjacencies between these entities is furthermore developed in Wanne (1998) and Wanne et al. (1999). In the following chapters we will briefly present those concepts of the Adjacency Relation System (ARS) that are necessary to understand our proposal of a schema for semistructured data. The same concepts are somewhat modified later in this work in order to make them suit better to our purposes.

Our work on a schema for semistructured data is based on adjacencies between edges. In our proposal we consider an adjacency as an interrelationship between two consecutive edges, and it is defined with the help of the node between them, whereas in Wanne et al. (1999) the adjacency is defined between elements of different types. Formally an Adjacency Relation System is (Wanne et al. 1999) as follows:

Definition 4.1. The *adjacency relation system* (ARS) is a pair $(\mathcal{A},\mathcal{R})$ where $\mathcal{A}=\{A_1,A_2,\cdots,A_n\}$, $n\geq 1$, is a set containing pairwise disjoint finite nonempty sets and $\mathcal{R}=\{R_{ij} \mid i,j\in\{1,2,\cdots,n\}\}$ is a set of relations, where each R_{ij} is a relation on $A_i\times A_j$.

If $(x,y_1),(x,y_2),\ldots,(x,y_m)\in R_{ij}$ are all the pairs of relation R_{ij} having x as the first component, then each element y_k $(k=1,2,\ldots,m)$ is said to be adjacent with the element x. Furthermore, denote by $Ad_j(x)$ the set $\{y_1,y_2,\ldots,y_m\}$ (Wanne et al. 1999).

We assume that the elements of each set A_i, $i=1,2,\cdots,n$, represent entities of a certain type T_i. The adjacency between elements can also be defined with the help of the so-called adjacency defining sets (τ) as follows. Associate with each index pair $i,j\in\{1,2,\cdots,n\}$ a set $K\subseteq\{1,2,\cdots,n\}-\{i,j\}$ of indices and a set of entity types $\tilde{T}_{ij}=\{T_k \mid k\in K\}$.

Elements $x\in A_i$, $y\in A_j$, where $i,j\in\{1,2,\ldots,n\}$ and $x\neq y$, are said to be *adjacent with respect to a set of entity types* $\tilde{T}_{ij}=\{T_k \mid k\in K\}\neq\phi$ if for each $k\in K$ there is an element $z\in A_k$ such that $x\in Ad_i(z)$ and $y\in Ad_j(z)$. The set $\tilde{T}_{ij}$ is called the *adjacency defining set* between the elements of A_i and A_j. For an *adjacency relation system associated with adjacency defining sets* (ARST), we use the notion $(\mathcal{A},\mathcal{R},\tau)$, where τ refers to the set of adjacency defining sets.

Definition 4.2. An ARST $(\mathcal{A},\mathcal{R},\tau)$ is said to be *unique* if for each pair $i, j \in \{1,2,\ldots,n\}$ of integers such that the corresponding adjacency defining set $\tilde{T}_{ij}$ is nonempty and for all elements $x \in A_i$, $y \in A_j$, x and y are adjacent if and only if they are adjacent with respect to $\tilde{T}_{ij}$.

Consider (Figure 4.1) a set of relations $\mathcal{R} = \{R_{11}, R_{12}, R_{13}, R_{21}, R_{22}, R_{23}, R_{13}, R_{23}, R_{33}\}$, where

$$R_{11} = \{(x_1, x_2), (x_2, x_1)\},$$
$$R_{12} = \{(x_1, y_2), (y_2, x_1), (x_2, y_2), (y_2, x_2)\},$$
$$R_{13} = \{(x_2, z_1), (z_1, x_2), (x_2, z_2), (z_2, x_2)\},$$
$$R_{33} = \{(z_1, z_2), (z_2, z_1)\},$$
$$R_{21} = R_{22} = R_{23} = R_{31} = R_{32} = \phi.$$

and the adjacency defining sets

$$\tilde{T}_{11} = \{T_2\},$$
$$\tilde{T}_{33} = \{T_1\},$$
$$T_{22} = T_{12} = T_{21} = T_{13} = T_{31} = T_{23} = T_{32} = \phi.$$

We can see that elements x_1 and x_2, $x_1, x_2 \in A_1$, are adjacent with each other as well as z_1 and z_2, $z_1, z_2 \in A_3$. The adjacency is defined via the relation R_{11} for x_1 and x_2, and via the relation R_{33} for z_1 and z_2, and on the other hand, also with respect to the adjacency defining sets $\tilde{T}_{11}$ and $\tilde{T}_{33}$. If there are no other non-empty adjacency defining sets, the ARST considered is unique.

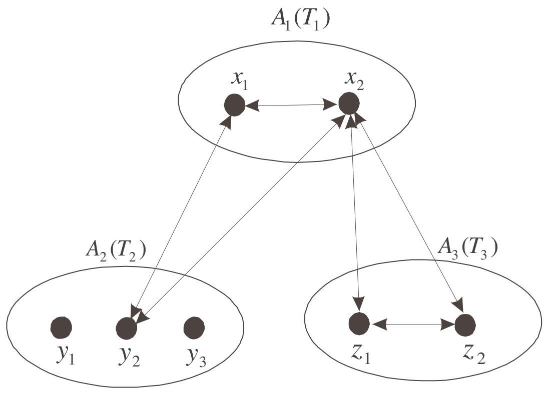


Figure 4.1. Directed graph illustrating the relations presented above.

We can use the notation $T_i \to T_j$ for a relation type to indicate that the relations R_{ij} are defined on $A_i \times A_j$. Furthermore, we can consider sets of relation types

$$\{T_i \to T_j \mid (i, j) \in S\},$$

where $S \subseteq \{1,2,\ldots,n\} \times \{1,2,\ldots,n\}$. For an ARST $(\mathcal{A}, \mathcal{R}, \tau)$, denote by $R \mid S$ the restriction of $\mathcal{R}$ on S, i.e. $\mathcal{R} \mid S = \{R_{ij} \in \mathcal{R} \mid (i, j) \in S\}$ (Linna et al. 2003).

Definition 4.3. A relation $T_r \to T_s$ is *determined uniquely* by the relation combination $\{T_i \to T_j | (i,j) \in S\}$ if for any unique ARST $(\mathcal{A}, \mathcal{R}, \tau)$ there is no other unique ARST ($\mathcal{A}, \mathcal{R}', \tau$) such that $\mathcal{R}|S = \mathcal{R}'|S$ but $R_{rs} \neq R'_{rs}$ (Wanne et al. 1999).

One of the problems with DataGuides is their size. In this work we consider also that problem, because even if the relation combination is valid, it should be as small as possible. A relation combination is valid and minimal when the number of relations in the combination is as small as possible.

Definition 4.4. Given the entity types $T_1, \ldots, T_n$, a set $S \subseteq \{1,2,\ldots,n\} \times \{1,2,\ldots,n\}$, and adjacency defining sets τ. A relation combination $\{T_i \to T_j \mid (i, j) \in S\}$ is said to be *valid* if for any unique ARST $(\mathcal{A},\mathcal{R},\tau)$ there is no other unique ARST $(\mathcal{A},\mathcal{R}',\tau)$ such that $\mathcal{R}|S = \mathcal{R}'|S$. Otherwise the relation combination is said to be *non-valid*.

Let us consider an ARST associated with planar graphs. We have the entities V (vertices), E (edges) and F (faces). Denote e.g. $T_1 = V$, $T_2 = E$ and $T_3 = F$. Relation combination of two relations, where the entity E occurs twice, i.e. the combination $\{V \to E, E \to F\}$ and its symmetric variants $\{V \to E, F \to E\}$ $\{E \to V, E \to F\}$ $\{E \to V, F \to E\}$ are minimal valid combinations (Wanne et al. 1999).

Next we will give a theorems of Wanne et al. (1999) that will be referenced later on in this work.

Theorem 4.1. Let $(\mathcal{A},\mathcal{R},\tau)$ be a unique ARST with n types $T_1,\ldots,T_n$, such that $\tilde{T}_{ii} \neq 0$ for $i = 1,2,\ldots,n$. A relation combination

$$\{T_i \to T_{k_1}, T_j \to T_{k_1}, T_i \to T_{k_2}, T_j \to T_{k_2}, \ldots, T_i \to T_{k_r}, T_j \to T_{k_r}\}$$

$i \neq j$, $i, j \neq k_1,\ldots,k_r$, determines uniquely the relations $T_i \to T_j$ and $T_j \to T_i$ if and only if $\tilde{T}_{ij} \neq \phi$ and $\tilde{T}_{ij} \subseteq \{T_{k_1},\ldots,T_{k_r}\}$ (Wanne et al. 1999).

5. SCHEMA PROPOSAL FOR SEMISTRUCTURED DATA

In structured database systems the main purpose of the schema is to describe the structure of the data. This in turn is a prerequisite that the query languages can access data efficiently. In this Section we will give our definition of semistructured data, show how to determine the degree of given data structure, and finally present our proposal of a schema for semistructured data.

5.1. Definition of Semistructured Data

According to Wanne et al. (1999) some relations of a given data structure are derivable from a set of stored relations. Each relation determined uniquely by a given relation combination is said to be *derivable* from the given relation combination. If a relation combination determines uniquely all other relations then it is said to be *valid*. This means that we have all relations of a data structure and the degree of the structure is one (1).

Example 5.1. Consider an ARST $(\mathcal{A},\mathcal{R},\tau)$, where the relations are as follows (Figure 5.1):

$$R_{11} = \{(x_1, x_2), (x_2, x_1)\},$$

$$R_{22} = \phi,$$

$$R_{33} = \{(z_1, z_2), (z_2, z_1)\},$$

$$R_{12} = \{(x_1, y_1), (x_1, y_2), (x_2, y_2), (x_2, y_3)\},$$

$$R_{21} = \{(y_1, x_1), (y_2, x_1), (y_2, x_2), (y_3, x_2)\},$$

$$R_{13} = \{(x_2, z_1), (x_2, z_2)\},$$

$$R_{31} = \{(z_1, x_2), (z_2, x_2)\},$$

$$R_{23} = \{(y_1, z_2), (y_2, z_1), (y_3, z_1)\},$$

$$R_{32} = \{(z_1, y_2), (z_1, y_3), (z_2, y_1)\},$$

and the adjacency defining sets are

$$\tilde{T}_{11} = \{T_2\}, \tilde{T}_{33} = \{T_1\},$$

$$\tilde{T}_{22} = \tilde{T}_{12} = \tilde{T}_{21} = \tilde{T}_{13} = \tilde{T}_{31} = \tilde{T}_{23} = \tilde{T}_{32} = \phi .$$

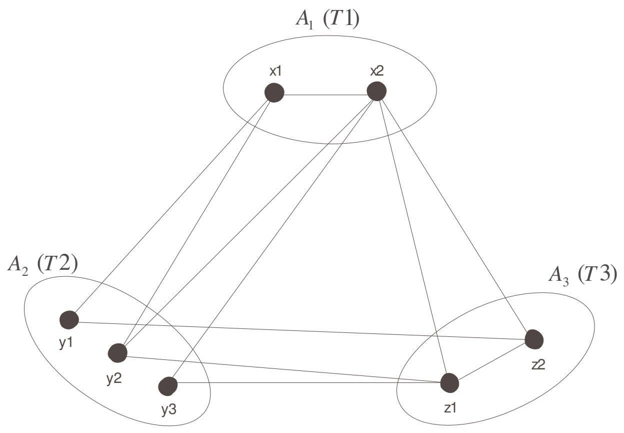


Figure 5.1. Undirected graph illustrating the relations of Example 5.1.

Let $S = \{(1,2),(2,3)\}$, then $\mathcal{R} \mid S = \{R_{ij} \in \mathcal{R} \mid (i, j) \in S\} = \{R_{12}, R_{23}\}$.

Let $(\mathcal{A}, \mathcal{R}', \tau)$ be another ARST where $\mathcal{R}'$ contains the same relations as $\mathcal{R}$ except the relation on $A_2 \times A_2$ which is defined by

$$R'_{22} = \{(y_1, y_2), (y_2, y_1)\}.$$

As in Example 5.1 it can be deduced that the ARST $(\mathcal{A}, \mathcal{R}', \tau)$ is unique. Since $\mathcal{R} \mid S = \mathcal{R}' \mid S$, the relation combination $\{T_1 \to T_2, T_2 \to T_3\}$ is not valid.

Now, e.g. the relation $T_1 \rightarrow T_3$ is not uniquely determined. Namely, if $(\mathcal{A}, \mathcal{R}^{'}, \tau)$ is another ARST, where $\mathcal{R}^{'}$ contains the same relations as $\mathcal{R}$ except the relations on $A_1 \times A_3$ and on $A_3 \times A_1$, which are defined by

$$R_{13}^{'} = \{(x_1, z_1), (x_1, z_2)\} \text{ and } R_{31}^{'} = \{(z_1, x_1), (z_2, x_1)\}.$$

Now we will define the notion of the degree of the data structure (Linna et al. 2003).

Let C be a subset of all possible relations of the form $T \rightarrow T^{'}$, where T and $T^{'}$ are types. Denote by $|C|$ the number of relations in C. Let $C^{'}$ be a relation combination such that $C \cup C^{'}$ is valid and $|C^{'}| = \min\{|C_i| \mid C \cup C_i \text{ is valid}\}$. Denote $\min(C) = |C^{'}|$. In other words, $\min(C)$ is the minimum number of relations, which is needed to complete C to a valid relation combination.

Denote by $D(C)$ the set of all relations, which can be derived uniquely from C. Obviously, $C \subseteq D(C)$. The *degree of a relation combination* C is defined as the ratio

$$Deg(C) = \frac{|D(C)|}{\text{the number of all relations}}.$$

According to the definition, it holds that $0 \leq Deg(C) \leq 1$ for each relation combination C.

Definition 5.1. Let C be the relation combination describing the given data structure. The data structure is said to be

(i) structured, if $Deg(C) = 1$;

(ii) unstructured, if $Deg(C) = 0$;

(iii) semistructured, if $0 < Deg(C) < 1$.

Note that obviously, $Deg(C) = 1$ if and only if C is valid.

In the following example (Wanne 1998) it is shown (Figure 5.2) that if we know the relations $F \to V$ and $V \to E$, i.e. $C = \{F \to V, V \to E\}$, then we can always derive the relations $V \to F$, $E \to V$ and $V \to V$. In Wanne (1998) it is also shown that relations $F \to F, E \to E, E \to F, F \to E$ can not be derived. This means that $D(C) = \{F \to V, V \to E, V \to F, E \to V,\ V \to V\}$. Thus the degree of C is $Deg(C) = 5/9 = 0.56$.

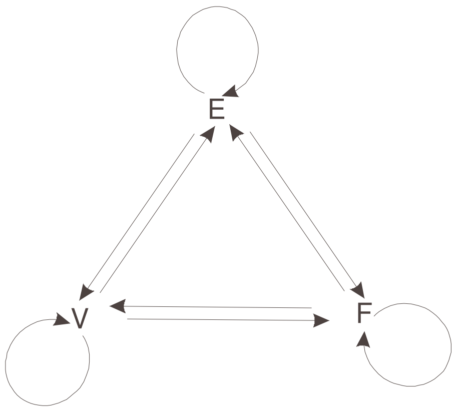


Figure 5.2. Three entities (E, V, F) and nine adjacent relations.

If e.g. $C = \{V \to E, E \to F\}$, it can be shown that $D(C)$ contains all the nine relations and thus $Deg(C) = 9/9 = 1$, and the relation combination C is valid. If on the other hand $C = \{V \to E, F \to V, E \to E\}$, it can be shown that

$$C(D) = \{V \to E, F \to V, E \to E, E \to V, V \to F, V \to V\}.$$

Thus the degree of C in this case is $Deg(C) = 6/9 = 0.67$.

In Figure 5.3 (Goldman 2000) we have a typical situation in which an OEM source database is depicted with two DataGuides. As we have found out in the previous sections it is fully possible that there are one, two or more DataGuides of one and the same source database. When we merge the lower parts of the DataGuides of Figure 5.3a and Figure 5.3b, i.e. the paths *C.D*, we loose some accuracy.

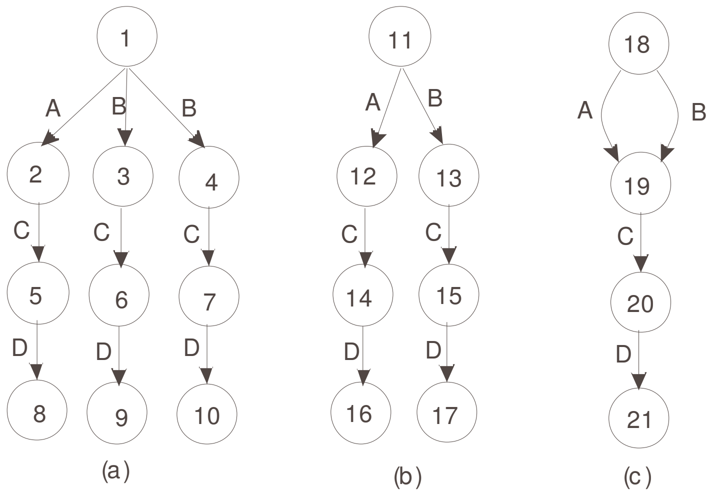


Figure 5.3. A source and two DataGuides (Goldman 2000).

If we wish to keep additional information in a DataGuide, like annotations e.g. statistical odds that an object reachable via *l* has any outgoing edges with a specific label (Goldman 2000), then we cannot say to which label path the annotation is applied. Referring to Figure 5.3c, we see that label paths *A.C* and *B.C* both reach the same object. Thus, if we store an annotation on object 20, we cannot know if the annotation applies to label path *A.C*, label path *B.C* or both. The DataGuide of Figure 5.3c is minimal. In the DataGuide in Figure 5.3b, however, we have two distinct objects for the two label paths, so we can correctly separate the annotation, but the DataGuide in Figure 5.3b is not any more minimal.

This is a situation in which the use of AdSchema, defined in detail later on, is a clear advantage. When the source database of Figure 5.3a is depicted with the AdSchema we will have a schema that is minimal, and the problem concerning annotations can be avoided. Besides, the AdSchema conforms initially to the same requirements as a strong DataGuide. By using AdSchema we can increase efficiency because the schema need not to be rebuilt in order to make it minimal. An AdSchema of the database of Figure 5.3 is depicted in Figure 5.4. The interrelationship between the types *B* and *D* is left out because the type *C* is an interrelationship defining type.

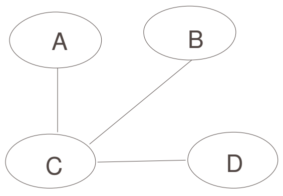


Figure 5.4. An AdSchema of the source database of Figure 5.3.

The graphs of Figure 5.3 can also be seen from another point of view. We can consider the number of the relations, i.e. the degree of the structure, so the three graphs of Figure 5.3 can be interpreted as follows. In Figure 5.3a we have 12 relations altogether. The relations are $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow C$, $C \rightarrow D$, $A \rightarrow D$ and $B \rightarrow D$, and their symmetric relations. Of those relations $A \rightarrow B$ is an empty relation (as well as its symmetric version). The relations $A \rightarrow D$ and $B \rightarrow D$ have an interrelationship defining type, which in this case is the type C. If the self-relations, i.e. $A \rightarrow A$, $B \rightarrow B$, $C \rightarrow C$ and $D \rightarrow D$ had been included then we should have had 16 relations altogether, but because the self-relations are insignificant to the result they are left out.

In Figure 5.3b we have the same 12 relations. The number of relations is the same because each relation is depicted only once, and in Figure 5.3a there are two paths that have the same edge labels. Also here the self-relations are omitted. In both of these cases (Figures 5.3a and 5.3b) the degree of the data structure is $Deg(C) = 12/12 = 1.0$, which means that the structure of the data is quite high.

In Figure 5.3c the number of relations is smaller and so is the degree of C. The known relations in Figure 5.3c are $A \to B$, $A \to C$, $B \to C$, $C \to D$, $A \to D$ and $B \to D$. Here again, we consider also the symmetric variants of each of the relations. So the total number of relations here is 12. However, in this case there are a few unknown relations. The relations $A \to D$ and $B \to D$ are not known as well as the relations $A \to C$, $B \to C$. The degree of the data structure is $Deg(C) = 4/12 = 0{,}33$.

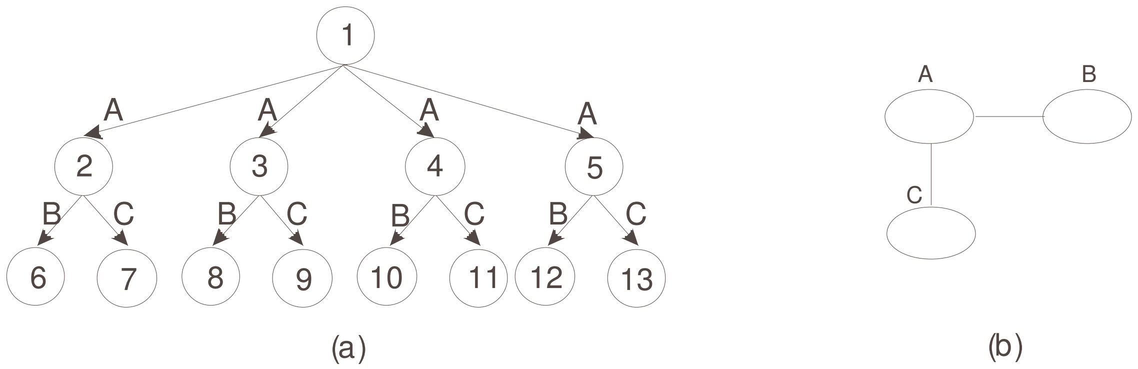


Figure 5.5. An example of the sizes of the source database (a) and the corresponding AdSchema (b).

In Figure 5.5a there is a sample OEM database. We can see that there are four edges emanating from the root node labelled with a letter *A*. From each of the nodes *2*, *3*, *4* and *5* there are two outgoing edges labelled with letters *B* and *C*. The relations in that graph are $A \to B$, $A \to C$, $B \to C$, $B \to A$, $C \to A$, $C \to B$, $A \to A$, $B \to B$ and $C \to C$, so we have 9 relations altogether. Of these relations we omit the self-relations. The degree of the data structure is $Deg(C) = 6/6 = 1.0$.

In Figure 5.5b we have the graph of Figure 5.5a presented with an AdSchema. The schema is simple and there are only three types with two interrelationships between them. The total number of relations is the same as in Figure 5.5a, but only two of them are needed because the rest of them can be derived. The schema is minimal and there is no confusion about interpretation of it.

As a conclusion of these examples we can say that, in most cases, if a semistructured database is presented with an AdSchema the size of the new schema is smaller than the original database, and it is also minimal. The AdSchema also fulfils the requirements of a strong DataGuide. As we can derive some relations from the known ones, we do not need to store all relations we know, and still we can keep the same level of accuracy as if all the relations were stored.

5.2. Determining of the Degree of a Data Structure

First we give an algorithm for the determining of the degree of a given data structure C, given as a set of relations.

Let entity types $T_1, \ldots, T_n$, a set $S \subseteq \{1,2,\ldots,n\} \times \{1,2,\ldots,n\}$ and adjacency defining sets τ be given. Consider the relation combination $C = \{T_i \to T_j \mid (i, j) \in S\}$. By the symmetry, we may assume that for each pair $(i, j) \in S$ also $(j, i) \in S$. The *Degree Testing Algorithm* is the following:

Step 0. Let the pairs $(i, j) \in S$ be marked pairs and all the other pairs unmarked pairs.

Step 1. Search for each unmarked pair $(i, j) \notin S$, such that there exists an integer $k \neq i, j$ for which $(i,k) \in S$ and $(j,k) \in S$. If such a pair and an integer are found, goto Step 2. Otherwise goto Step 3.

Step 2. Find all integers $k_1, \dots k_r$, $r \geq 1$, such that $(i,k_s),(j,k_s) \in S$ for $1 \leq s \leq r$. If for at least one s, $1 \leq s \leq r$, we have $T_{k_s} \in \tilde{T}_{ij}$ and $\tilde{T}_{ij} \subseteq \{T_{k_1}, T_{k_2}, \dots, T_{k_r}\}$, then include relations $T_i \to T_j$ and $T_j \to T_i$ in the relation combination, i.e. add the pairs (i, j) and (j,i) to the set S and goto Step 1. If there is no such integer s, then goto Step 1.

Step 3. Let C_1 be the set of all new relations derived by Step 2. The degree of the given data structure C is

$$Deg(C) = \frac{|C| + |C_1|}{n^2}$$

and the set of all obtained relations is $C \cup C_1$.

Theorem 5.1. The determining of the degree of semistructured data is a decidable question.

Proof:
We will show that the above algorithm decides the question.

The purpose of Steps 1 and 2 is to decide if the present relation combination determines uniquely at least one new relation. If in Step 1 no pair $(i, j) \in S$, $i \neq j$, such that there exists an integer $k \neq i, j$ for which $(i,k) \in S$ and $(j,k) \in S$ is found, then, by Theorem 4.1, no further relation can be derived from the obtained relations. Thus we know the set C_1 of all new relations derivable from the set C and can go to Step 3 and calculate the degree of C.

If in Step 1 at least one pair $(i, j) \notin S$, $i \neq j$, satisfying the given property was found and the assumptions of Step 2 are satisfied then the new relations $T_i \to T_j$ and $T_j \to T_i$ are derivable by Theorem 4.1 and they are added to the set of previous relations S. The procedure, i.e. Step 1, is then repeated for finding all possible new relations. If, conversely, the assumptions of Theorem 4.1 are not satisfied in Step 2, make the pair (i, j) marked and go to Step 1. This proves that the algorithm is working correctly.

Illustrate the Degree Testing Algorithm by the following example. Let $T_1 = V$, $T_2 = E$ and $T_3 = F$ be the types of vertices, edges and faces in the case of planar graphs. In the usual interpretation of planar graphs

$$\tilde{T}_{11} = \tilde{T}_{VV} = \{E\}$$

$$\tilde{T}_{22} = \tilde{T}_{EE} = \{V, F\} \quad \text{(winged-edge interpretation)}$$

$$\tilde{T}_{33} = \tilde{T}_{FF} = \{E\}$$

$$\tilde{T}_{12} = \tilde{T}_{VE} = \tilde{T}_{EV} = \tilde{T}_{21} = \phi$$

$$\tilde{T}_{23} = \tilde{T}_{EF} = \tilde{T}_{FE} = \tilde{T}_{32} = \phi$$

$$\tilde{T}_{13} = \tilde{T}_{VF} = \tilde{T}_{FV} = \tilde{T}_{31} = \{E\}$$

Let now the relation combination be $C = \{V \to E, F \to V\}$. By the symmetry, the relations $E \to V$ and $V \to F$ belong also to $D(C)$. Thus we may assume at this stage that $S = \{(1,2), (2,1), (1,3), (3,1)\}$.

Consider in Step 1 the pair $(1,1) \in S$, i.e. the relation $V \to V$. Since $(1,2) \in S$, i.e. $V \to E \in C$, and $(2,1) \in S$, i.e. $E \to V \in C$ go to Step 2 with $k = 2$. There are two integers k_s such that $(1, k_s)$, $(1, k_s) \in S$, namely $k_1 = 2$ corresponding to E and $k_1 = 3$ corresponding to F. Since $E \in T_{VV} \subset \{E, V\}$, i.e. $T_{k_1} \in \tilde{T}_{11} \subset \{T_{k_1}, T_{k_2}\}$, we can include the relation$V \to V$ in the relation combination and go back to Step 1. For the unmarked pair $(2,2)$ there is an integer $k = 1$ such that $(2, k) \in S$. There is only one integer k_s, namely $k_s = 1$, such that that $(2, k_s) \in S$. Now we have $V \in \tilde{T}_{EE} = \{V, F\}$, i.e. $T_{k_s} \in \tilde{T}_{22}$ but $\tilde{T}_{22} = \{V, F\} \subseteq \{T_{k_s}\}$. Similar deduction holds also for the pair $(3,3)$. Thus the relations $E \to E$ and $F \to F$ do not belong to $D(C)$. Consider finally the unmarked pair $(2,3)$, i.e. the pair (E, F). In this case for $k = 1$ we have $(2,1), (3,1) \in S$, i.e. $E \to V, F \to V \in S$, and we can go to Step 2. Since $\tilde{T}_{ij} = \tilde{T}_{23} = \tilde{T}_{EF} = \phi$, the pair $(2,3)$, i.e. the relation $E \to F$ (and $F \to E$), does not belong to $D(C)$. Since there are no other candidates to $D(C)$ left, we can deduce that $C_1 = \{E \to V, V \to F, V \to V\}$ and so

$$Deg(C) = \frac{|C| + |C_1|}{n^2} = \frac{2+3}{3^2} = {}^5\!/_9 = 0{,}56 .$$

The Degree Testing Algorithm enables the definition of a general query based on the given data structure, i.e. on the given relation combination C. We now give a definition for the general query.

5.3. General Query

Given an ARST $(\mathcal{A}, \mathcal{R}, \tau)$, a relation combination C and an element x of type X, find the set Y_x of all elements of type Y such that

(i) the relation $X \to Y \in D(C)$ and

(ii) $(x, y) \in R_{AB}$ for all $y \in Y_x$, where $A, B \in \mathcal{A}$ and elements of A are of type X and elements of B are of type Y.

Note that in (i) the set $D(C)$ of relations can be determined by the preceding algorithm.

It can be shown that this general definition of a query includes e.g. all the special queries introduced in Ni et al. (1994), namely the queries such as direct query, inverse query, self queries, direct transitive query, inverse transitive query, and indirect transitive queries.

5.4. Algorithms

In the following sections we will present three algorithms. The first two algorithms are based on the algorithms of Töyli (2002), whereas the third algorithm is developed in this work. The aim of the first two algorithms is to show that it is possible to model data of different structures with the adjacency model. The third algorithm creates a new schema of a source database, which is a semistructured database.

5.4.1. Modeling of Relational Data

In Buneman et al. (1996) the authors show that relational databases are easily encoded as trees. In this subsection we show that relational databases can also be encoded with the adjacency model. We will go through the process in which we convert a relational model into the adjacency model. In order to succeed with this process we expect that the data in the relational model is normalized, at least to the 3rd normal form. The process during which we proceed from the relational model into the adjacency model is quite straightforward. It consists of nine steps, which are given here:

Step 1. Select an attribute (column) from a table.
Step 2. Create a corresponding type.
Step 3. Repeat steps 1 and 2 until all the distinct tables and attributes are handled.
Step 4. Remove those types which have been created of the secondary keys, if any.
Step 5. Take a primary key and define it as an adjacency defining type.
Step 6. Repeat step 5 until all the distinct primary keys are defined as adjacency defining types.
Step 7. Create an adjacency between an element of a type and an element of its corresponding adjacency defining type (i.e. between an attribute and the primary key of a table).
Step 8. Repeat step 7 until all the adjacencies are created between the elements and their adjacency defining sets.
Step 9. Create all other wanted adjacencies between the types.

In Step 1, if there are multiple tables in our relational database, we select first one table from which we start the conversion process. Actually, it does not matter which table we select, but it could be a good advice to choose the one which have the most attributes. Next we will choose one attribute of that table and proceed to the next step.

In Step 2 we create a new type of the attribute we have just selected in step one. When we name a new type, it is advisable to use the column name of that attribute as a type name. Often the column name is representational of the content of that particular attribute, and on the other side we can avoid conflicting names between the types. The type and the length of the attributes are insignificant, because the Adjacency Model treats all data equally, i.e. the type names and the adjacencies between them are the only significant matters.

In Step 3 we repeat the two previous steps. However, as soon as all the attributes of one table are treated, we take another table under consideration. As long as we repeat steps one and two we do not need to be concerned if there are more than one type with the same name, because these extra types are removed in the next step.

Now the primary phase of the conversion process is completed. After all the tables and all the attributes are considered we enter the next step. We have as many types as we had columns in the original database. However, some of them describe one and the same attribute. In Step 4 we remove all those duplicate attributes that have been created of the secondary keys of the tables, because they all have a corresponding type created of the primary key. Finally, there should not be duplicate types supposing the source database is at least in 3rd normal form.

In the theory of the Adjacency Relation Systems (Wanne 1998) there is one concept called an adjacency defining set. In this fifth Step we consider these sets. Every distinct attribute of a primary key, sometimes the primary key consists of two or more attributes, can be defined as an adjacency defining type. So we will go through all types and check if it is a part of a primary key or not. If it is, then that particular type can be used to define adjacency between two or more types of the same table. If the same type is used as a secondary key in another table of the source database, then it can define adjacencies also between types created of attributes of two or more tables.

In Step 6 we repeat the procedure of step five and select another type which is a part of a primary key of any table of the source database. This new type will also be defined as an adjacency defining type. This step will be repeated until all adjacency defining types are found and declared.

In the previous steps we have concentrated on the types of the adjacency model. Next we will consider the elements of the types. In the relational model adjacencies are defined with relationships between the tables of the database. In the adjacency model the adjacency is defined with the help of the adjacency defining types. Moreover, the adjacency is created on the element level, not on the type level.

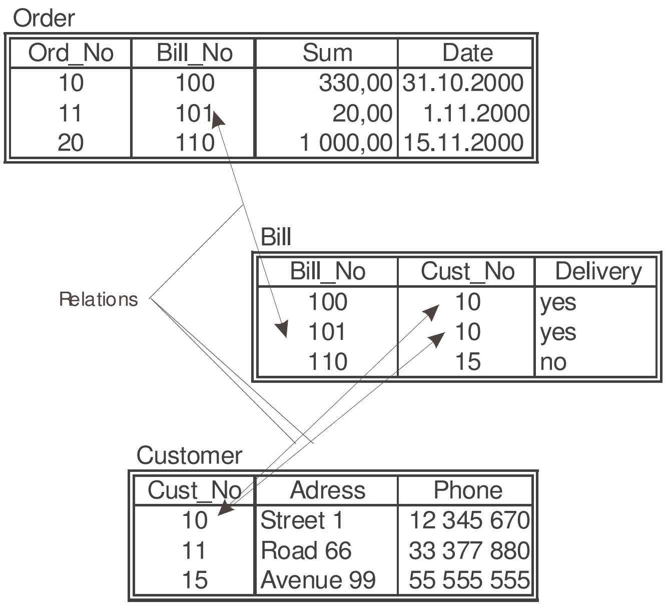
In Step 7 we draw a line, i.e. a relation between an element of a type and an element of its corresponding adjacency defining type. We must remember here that we are not connecting elements based on their values, but according to the adjacency they have as attributes in the same table. The attributes, which have been declared as secondary keys in the source database, connect the types taken from different tables.

In Step 8 we repeat the procedure of Step 7 until all the adjacencies between the elements of the different types are created. When we are finished with our work all elements of every distinct type are in relation with some other element. Some elements are adjacent with two or more elements, others has only one adjacent element. In the first case we consider elements of adjacency defining types and in the other case elements of the ordinary types, i.e. adjacencies between the attributes within a table.

In Step 9 we considered all those adjacencies that are for some reason not considered yet.

Before we go on with another algorithm for semistructured data we will give an example of how the algorithm presented above works. The example database contains three tables with ten attributes, so our adjacency model will contain only eight types, because the secondary keys are removed.

In Figure 5.6 the three tables are *Order*, *Bill* and *Customer*. Each of the tables have records (rows) with a number of attributes (columns) and relationships which are composed with the primary keys (*Ord_No*, *Bill_No* and *Cust_No*) and the secondary keys (*Bill_No* and *Cust_No*). None of the attributes have null values.



Order

Ord_No	Bill_No	Sum	Date
10	100	330,00	31.10.2000
11	101	20,00	1.11.2000
20	110	1 000,00	15.11.2000

Bill

Bill_No	Cust_No	Delivery
100	10	yes
101	10	yes
110	15	no

Customer

Cust_No	Adress	Phone
10	Street 1	12 345 670
11	Road 66	33 377 880
15	Avenue 99	55 555 555

Figure 5.6. Customer Order System – an example of a relational database.

As mentioned above, we start by picking up attributes from the tables of the relational model and create types of them one at a time. In our example we have the attributes *Ord_No*, *Bill_No*, *Sum* and *Date* in the *Order* table and *Bill_No*, *Cust_No* etc. in the tables *Bill* and *Customer*. After the first three steps we have ten types, some of which are duplicates.

In step four we eliminate those duplicate types which are declared as secondary keys, in this case the types *Bill_No* and *Cust_No*. So, at the end, we have only eight types left, *Ord_No*, *Bill_No*, *Sum*, *Date*, *Cust_No*, *Delivery*, *Address* and *Phone*. The next steps, steps five and six, check the primary keys and define them as adjacency defining types. There are three of them, i.e. *Ord_No*, *Bill_No* and *Cust_No*.

After that it is quite simple to create the adjacencies between the elements of the adjacency defining types and the elements of those types which are adjacent with them, like the elements of *Ord_No*/*Bill_No*, *Ord_No*/*Sum* and *Ord_No/Date*. The same thing will be done with the other adjacency defining sets and their corresponding types.

Now we have managed all the steps except the last one. In the last step, step nine, we create the rest of the adjacencies, if there are any. In most of the cases we have succeeded to create all the adjacencies, because the relations between the tables are mostly implemented with the help of the primary and secondary keys. When we eliminated the secondary keys we had only the primary keys left, and with them we, in the adjacency model, could create the adjacencies necessary for our purposes. If there are any non-key relations in the relational model, we will add them into our representation.

In Figure 5.7 we have presented the relational database of Figure 5.6 with the adjacency model. There are only eight types left: *Ord_No, Bill_No, Sum, Delivery* etc., which represent the attributes of the tables. Each type contains elements, which in turn, correspond the values of those attributes, like *Order_No*: *10, 11, 20*; *Delivery*: *yes, no* etc.. The relationships between the elements of different types are represented with directed edges; for example the upper part of Figure 5.7 represents the relationships (R_{Date,Ord_No}, R_{Bill_No,Ord_No} and R_{Sum,Ord_No}) of the attributes of the table *Order* of Figure 5.6.

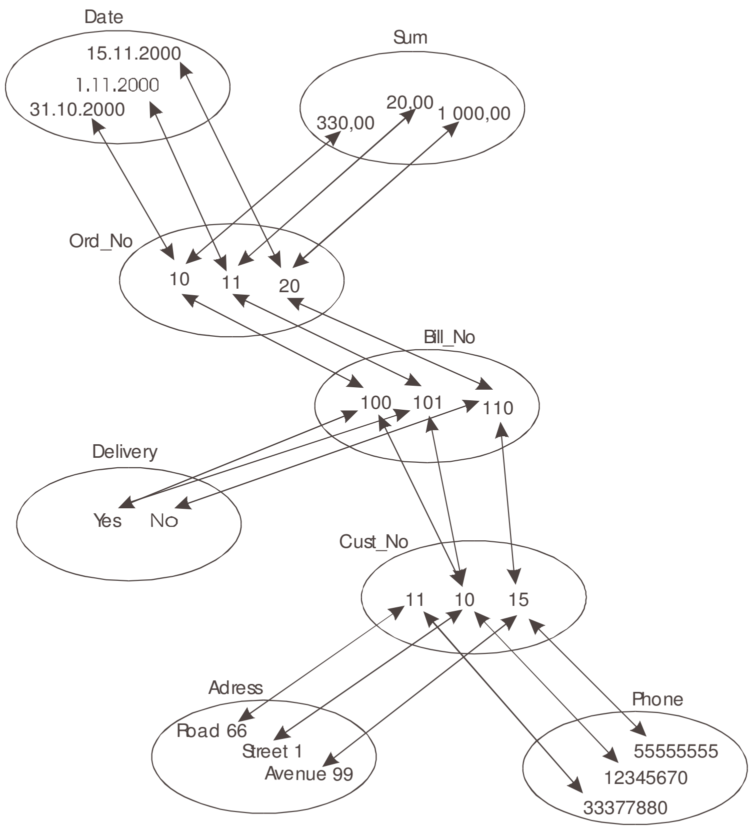


Figure 5.7. The relational database of Figure 5.15 represented with the Adjacency Model.

The types *Bill_No*, *Sum* and *Date* are adjacent with each other with respect to the adjacency defining type, which in this case, is the *Ord_No*. So we can denote the adjacency defining sets with $\tilde{T}_{Sum,Bill\_No} = \{T_{Ord\_No}\}$, $\tilde{T}_{Date,Bill\_No} = \{\tilde{T}_{Ord\_No}\}$ and $\tilde{T}_{Sum,Date} = \{T_{Ord\_No}\}$. Additionally, we can see that the elements of the type *Bill_No* are adjacent with the elements of the type *Ord_No* (e.g. $100 \in Ad_{Bill\_No}(10)$) and so do the elements of the types *Sum* and *Date* (e.g. $330.00 \in Ad_{Sum}(10)$, $31.10.2001 \in Ad_{Date}(10)$). These adjacencies are defined by the adjacency defining sets, not with respect to the values of the corresponding types.

The adjacencies, which we have shown above, were implemented between the types taken from one and the same table. It is equally possible to define adjacencies between types taken from two different tables. In our example the type *Bill_No* is an attribute of two tables, *Order* and *Bill*, so we can define the adjacency between the tables in the adjacency model, e.g. $10 \in Ad_{Cust\_No}(100)$ and $100 \in Ad_{Bill\_No}(10)$. The rest of Figure 5.7 can be represented equally applying the definitions above.

The algorithm we have presented above is quite straightforward, because the underlying data model is based on a fixed schema. However, the amount of data that do not have a fixed schema increases rapidly. Our next algorithm is developed to manage such data. The algorithm reads a tree or a graph-like structure and creates an adjacency model of it. With this algorithm we want to show that semistructured data can be modelled with the adjacency model.

5.4.2. Modeling of an Object Exchange Model

In this Section we will present our algorithm that converts an Object Change Model (McHugh 1997) to an adjacency model. We have chosen this model because it is most widely used, and the DataGuide schema is also based on this same model. The conversion process from the OEM model into the adjacency model differs in some aspects from the process of the relational model. The main principle is that every edge of the source database corresponds to one type of the adjacency model, and every node of the source database corresponds to a relation between two elements of two different types. The conversion between the models proceeds as follows:

Step 1. If there is an incoming edge into the root node of the graph, create a new type of it.

Step 2. Select an edge from the graph.

Step 3. Create a new type of the edge.

Step 4. Repeat steps 2 and 3 until all distinct edges are handled.

Step 5. Select a type, which has two or more distinct types that are adjacent with it, and define it as an adjacency defining type.

Step 6. Repeat step 5 until all such types are handled.

Step 7. Create an adjacency between an element of an atomic type (a type which does not have other types that are adjacent with it) and an element of its corresponding adjacency defining type.

Step 8. Repeat step 7 until all the elements of the atomic types are handled.

Step 9. Draw the adjacencies between the elements of the adjacency defining types according to the original graph.

Very often the root node has an incoming edge. This edge is considered first in Step 1, where we create a new type of it. Most often that particular edge has a name already and we can use it, but if it is not named then we can use what ever name we want. However, we must think out the name well so that there would not be any conflicts between the names of the other edge of the semistructured database. If there is no incoming edge, we can directly move on to the next step.

In Step 2 we select one of the edges of the semistructured database. If we make the selection for the first time we take an edge emanating from the root node, otherwise we can select any of the remaining edges we have not visited yet. Our algorithm uses the depth-first search, so we know exactly those edges that are already picked up.

In Step 3 we create a new type of the label of the last selected edge. In the conversion process from the relational model to the adjacency model duplicate types were allowed and removed later, but because our algorithm keeps track of the edges handled, we will not face the same problem here.

Step 4 repeats the three previous steps and uses the DFS to manage all the edges of the source database. Because we use a commonly proven search method in our algorithm even the cycles will not cause any problem.

In Steps 5 and 6 we define the adjacency defining types. As a rule of thumb we can say that all those edges of the semistructured database that end to a node that have outgoing edges can be defined as such. These two steps will be repeated until all such types are defined.

Here we end the first phase of our algorithm. All types are created and all adjacency defining types are defined. In the next three steps we finish the conversion process by drawing the relations, i.e. the adjacencies between the elements of the types. When we consider the Object Exchange Model, the elements of the types consist of the object identifiers of the nodes to which the incoming edges are connected. Basically, if we have three edges with the same name, then we have only one type with that name and three object identifiers as elements within that type.

In Steps 7 and 8 we create the relations, i.e. an adjacencies, between elements of the different types. In this and the next step we consider only adjacencies between atomic types and adjacency defining types. The elements, between which the relation is created, can be determined according to the inner nodes of the source database. The incoming edge to the inner node is defined as an adjacency defining type, and the object identifier of that node is the starting element. The outgoing edge, which is an atomic type in the adjacency model, points to a node, and the object identifier of that node is the ending element of the relation. When we have concluded these two steps, we have created all the adjacencies between the adjacency defining types and the atomic types.

In Step 9 we create the adjacencies between the adjacency defining types. These types are inner nodes in the source database. The relation in the adjacency model is drawn from a type which is created of the label of the incoming edge of the inner node, and the relation ends to a type which is created from the label of the outgoing edge of the same inner node. The relation is, of course, an adjacency between the elements of the named types.

Finally, when all the types are considered the semistructured model is converted to an adjacency model. The number of types in our model is less (in general) than the number of edge labels in the source database, because the edges with the same label are considered as one type.

Next we will take some examples of the conversion process. We will consider here two different examples and give a brief explanation of the process. We will start with an OEM database (Figure 5.8) presented by McHugh, Widom, Abiteboul, Luo & Rajaraman (1998). In Figure 5.9 we have the same database represented with the adjacency model.

In Figure 5.8 we can see that there is a named input edge, which is called *DB*, so the first type is created of it. After that we go through the whole graph with the depth first search and select all distinct edges, and create corresponding types of their labels, like *Movie*, *Actor*, *Title* etc. Next we define the adjacency defining types. In this first example the adjacency defining types include e.g. the types *Movie*, *Actor* and *Price*. Next we create the adjacencies from the elements of the adjacency defining types to the elements of their corresponding adjacent types like the relations between *Name* - *Actor*, *Currency* - *Price*, etc. Finally we create the adjacencies between the adjacency defining sets (*Movie – Actor, Movie – Title* etc.). All special cases, if any, are handled separately after we have finished the standard procedure. In Figure 5.8 the edge with the label *References* between the two *Movie* types is a special case which will be considered later in this example.

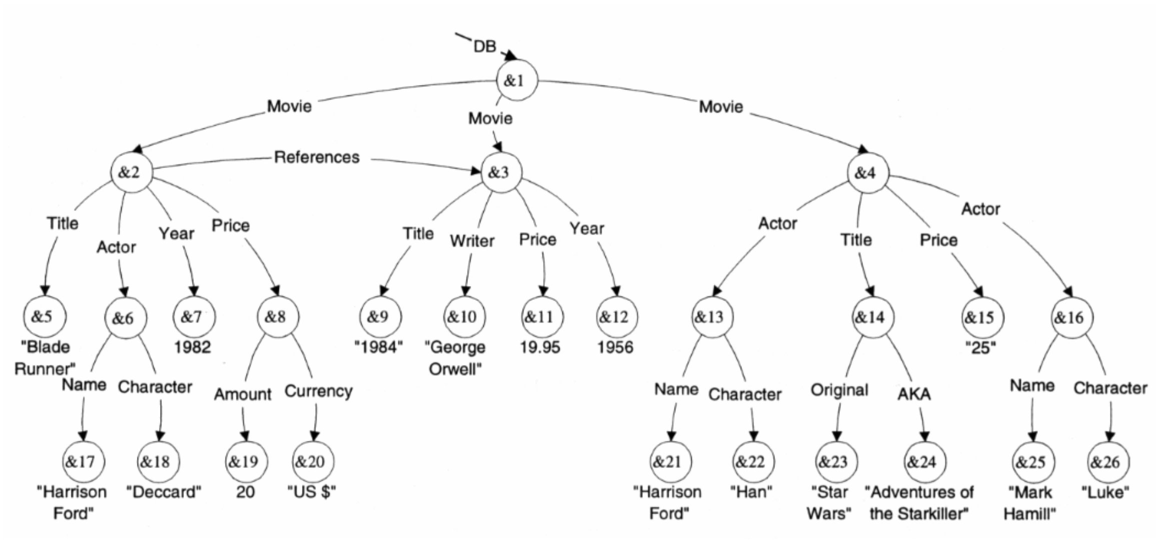


Figure 5.8. An OEM database (McHugh et al. 1998).

In Figure 5.9 there are thirteen distinct types with adjacency defining types and ordinary types. If we take the type *Actor*, we can see that some of the elements of the types *Name* and *Character* are adjacent with each other with respect to the type *Actor* ($\tilde{T}_{Name,Character} = \{T_{Actor}\}$). The elements of the types *Year*, *Title*, *Writer* and *Price* are adjacent with each other with respect to the type *Movie*, so we can write *Movie* ($\tilde{T}_{Actor,Year} = \{T_{Movie}\}, \tilde{T}_{Actor,Title} = \{T_{Movie}\}$ etc.). For example Harrison Ford is the name (&17) of the *Actor* (&6) who plays in *Movie* (&2). We can also see that the *Character* (&18) Deccard is played by an *Actor* (&6) whose *Name* is Harrison Ford (&17), because the adjacency of these two types is defined by the type *Actor*. The adjacency between the elements of the types *Name* and *Actor*, e.g. $\&17 \in Ad_{Name}(\&6)$ is defined directly by the relation without any adjacency defining type.

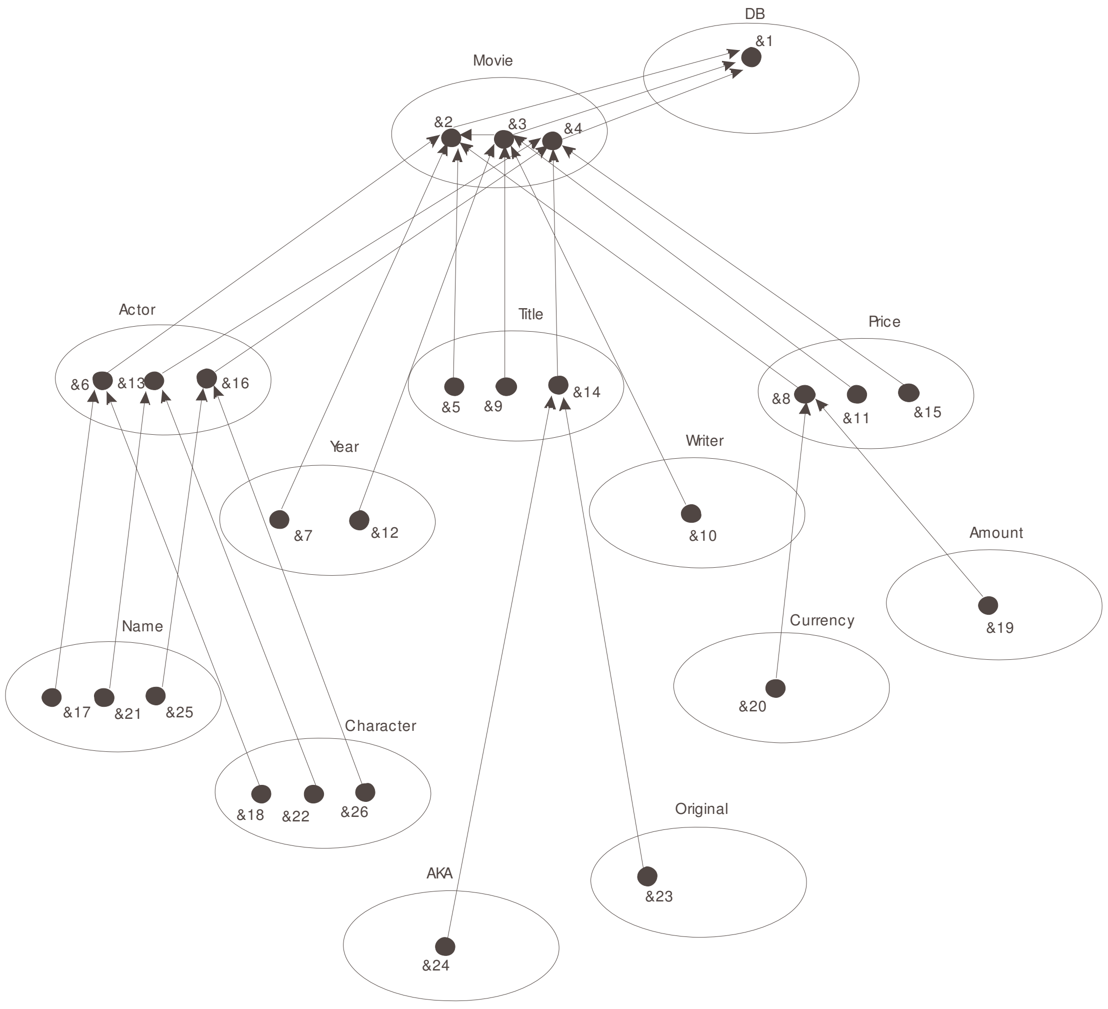


Figure 5.9. An OEM database of Figure 5.8 modelled by the Adjacency Model.

In Figure 5.9 the types *Title* and *Price* have elements with two different expressions. First, they both have elements that do not have adjacencies with other elements and secondly, they have elements with adjacencies with other elements. For example, the element &5 of the type *Title* do not have adjacent elements, whereas the element &14 has two adjacent elements (i.e. $\&24 \in Ad_{AKA}(\&14)$ and $\&23 \in Ad_{Original}(\&14)$). This kind of structures make semistructured data difficult to model with the common data models, but as we can see we can present structures like this with the adjacency model.

The edge labelled *References* in Figure 5.8 is not depicted as a distinct type in the adjacency model, because there is a more natural way to represent it. It can be represented with only one type by defining a relation between the two elements &2 and &3 of the type *Movie*. With this method we can reduce the number of distinct types.

Our next example deals with the semistructured data graph of Figure 5.10. It represents an OEM database, which contains information about the Stanford Database Group (McHugh et. al 1997). In principle it has the same structure as the OEM database of Figure 5.8 but there is one significant difference. The graph contains a cycle between the two nodes *&4* and *&5* with edges labelled *Member* and *Project*.

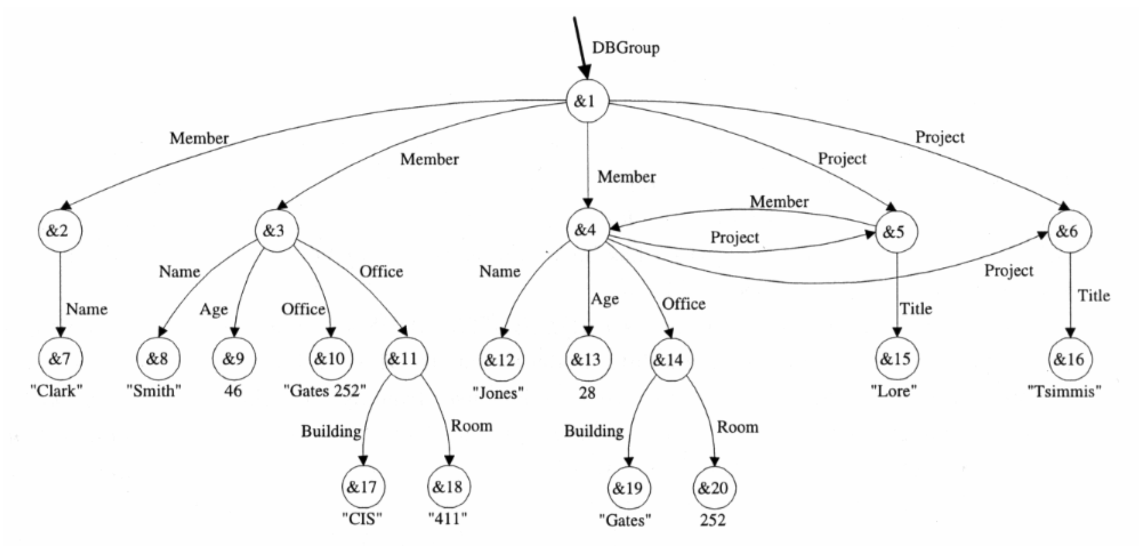


Figure 5.10. An OEM database with a cycle between the nodes &4 and &5 (McHugh et al. 1997).

In the adjacency model of Figure 5.11 the adjacency is defined with relations between the elements $R_{Member,\Pr oject}$ and $R_{\Pr oject,Member}$. In this case we can use also an adjacency defining type, i.e. the *DBGroup*. Both of the types *Member* and

Project have an element (&4 and &5) that is adjacent with an element (&1) of the adjacency defining type *DBGroup*, i.e. $\tilde{T}_{Member,\Pr oject} = \{T_{DBGroup}\}$.

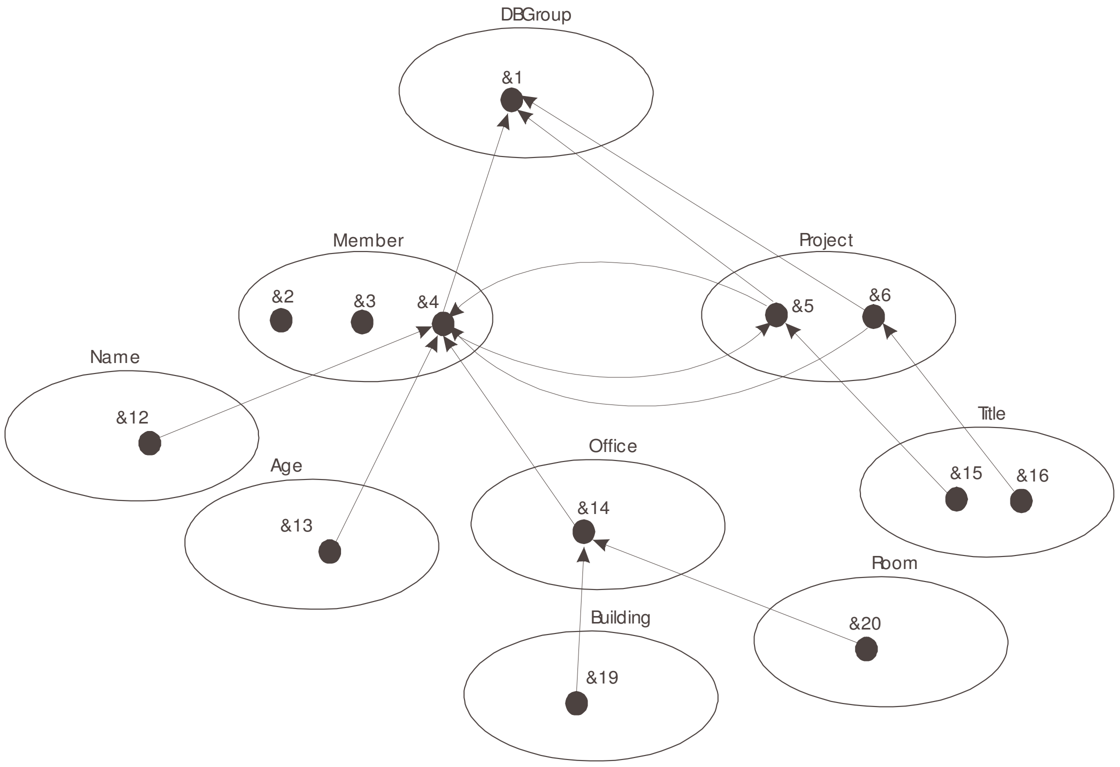


Figure 5.11. The cycles of Figure 5.10 presented with the Adjacency Model.

Since $\&5,\&6 \in Ad_{\Pr oject}(\&1)$ in Figure 5.11, the elements &5 and &6 are adjacent with each other with respect to $\tilde{T}_{\Pr oject,\Pr oject}$. This kind of adjacency is typical in the context of semistructured data when we have a distinct root node (here *DBGroup*). The same thing can be found between the types *Project* and *Member*, where $\&5,\&6 \in Ad_{\Pr oject}(\&4)$.

5.5. Definition of an Adjacency Schema

In this Section we present our proposal of a schema for semistructured data. Although our work follows the theory of Wanne (1998) and Wanne et al. (1999), there is one exception in that our proposal concerns the adjacency between the types and not between the individual elements of the types, which is the case in Wanne (1998). This means new formulations of some of the definitions introduced in Wanne (1998) and Wanne et al. (1999). These will be presented in this section.

As a consequence of the fact that we consider a schema for a database we do not need to show the atomic objects, i.e. the values of the source database. This means that the schema becomes more simplified. The values of the atomic objects can be stored in a common data store, like a relational or object database, and hence can be accessed by e.g. with SQL.

In Figure 5.12a there is a part of a semistructured database. The database graph consists of nodes and labeled edges. The nodes do not have any object identifications. In Figure 5.12b the same graph is depicted with an Adjacency Schema, which consists of types and interrelationships between them. Intuitively, in Figure 5.12a, the meaning of a relationship between two consecutive objects is defined with the help of a labelled edge, whereas in an Adjacency Schema the relationship between two interrelated types is defined with respect to a node of a database graph.

Definition 5.1. An *Adjacency Schema* (AdSchema) is a pair ($\mathcal{T}_{\mathcal{A}},\mathcal{R}$), where $\mathcal{T}_A = \{T_1, T_2, \ldots, T_n\}$, $n \geq 1$, is a set of types, and $\mathcal{R} = \{R_i \mid i \in \{1,2,\ldots,n\}\}$, where each $R_i = (T_i, Ir(T_i))$, and $Ir(T_i)$ denotes a subset of $\mathcal{T}_A$.

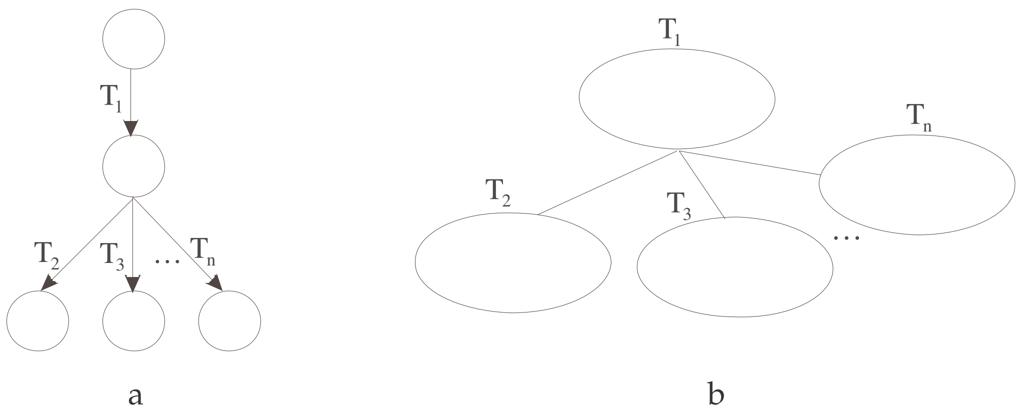


Figure 5.12. A part of a data graph depicted with an Adjacency Schema.

Definition 5.2. If $R_i = (T_i, \{T_{i_1}, T_{i_2}, \ldots, T_{i_m}\}) \in \mathcal{R}$, then each type T_k $(k = 1,2,\ldots,m)$ is said to be *interrelated* to the type T_i. Furthermore, denote by $Ir(T_i)$ the set $\{T_{i_1}, T_{i_2}, \ldots, T_{i_m}\}$. Since it is possible that $i = i_j$ for some $j = 1,\ldots,m$, the relation R_i means that a type T_i is interrelated to itself. This is a special case in a data graph where the start and end node of an edge are the same.

We use the notion $T_i \rightarrow T_j$ to show the interrelationship between types, like T_i and T_j. If we have semistructured data that contains elements of type T_i and T_j and if the elements of these two types are adjacent with each other, then we can denote that by $T_i \rightarrow T_j$. This means that $T_j \in Ir(T_i)$ and the interrelationship of the elements can be represented also as a pair of elements (Wanne 1998).

All known approaches, including approximate DataGuides (Goldman 2000), consider directed graphs as schemas for semistructured databases. In our approach we consider symmetric relations between the types and so there is no predefined order between the types of the schema. This makes it possible that we can traverse the data structure in both directions.

Definition 5.3. An Adjacency Schema ($\mathcal{T}_{\mathcal{A}}, \mathcal{R}$) is said to be *symmetric* if for each pair of types it holds that $T_i \in Ir(T_j)$ if and only if $T_j \in Ir(T_i)$.

The number of the nodes in a semistructured database graph can vary from only a few nodes to a large number of them. The number of nodes is not a problem, but it would be more convenient if some of the relations could be left out of the schema, i.e. they could be derived from the known ones. We can use the interrelationship between some particular types when the relation is defined with respect to a third type.

Definition 5.4. In the case of a symmetric Adjacency Schema, the types $T_i, T_j \in \mathcal{T}_A$ are said to be *interrelated with respect to a type* T_k, $T_k \in \mathcal{T}_A$, $k \in \{1,2,\ldots,n\} - \{i, j\}$, if $T_k \in Ir(T_i)$ and $T_k \in Ir(T_j)$. The type T_k is called an *interrelationship defining type*.

In Figure 5.13 the types T_1 and T_3 are interrelated to each other with respect to the type T_2, in notation $\tilde{T}_{13} = \{T_2\}$. If $i = j$ the type is adjacent with itself.

In Figure 5.13 we can see a schema with j types and the interrelationships between them. As we can see there are many interrelationships drawn between the types, but many of them are unnecessary because they can be derived from the already known interrelationships, i.e. there is a minimal set of relations that need to be stored.

In Goldman (2000) there is a concept called a data path, which is a sequence of one or more dot-separated labels $l_1.l_2.,\ldots,.l_n$ starting from an object o. In our work the corresponding concept is realized with transitively interrelated types.

Transitively interrelated types are types that are adjacent with each other with respect to a set of consecutive intermediate types.

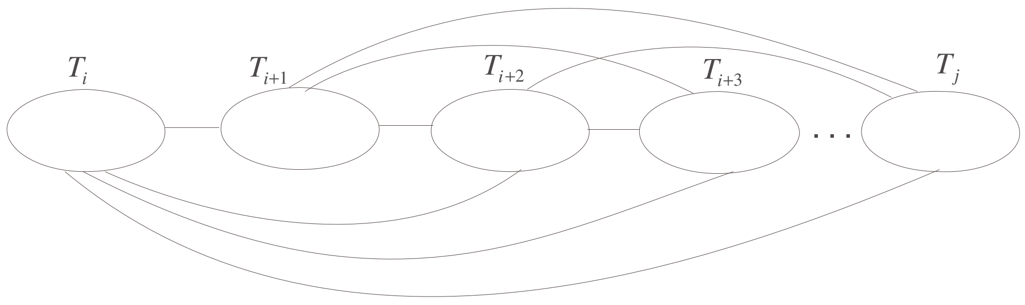


Figure 5.13. A valid structure containing interrelationships between a set of types.

Definition 5.5. Two types T_i and T_j, $i \neq j$ are *transitively interrelated* if there are one or more consecutive intermediate types between them, such that T_i and T_k are interrelated with respect to a type T_{k-1} for $k = i+2, i+3, \ldots, j$.

Consider the case of Figure 5.14. It has the same set of types as is shown in Figure 5.13 but the number of interrelationships is lesser. The interrelationship types are as in the previous definition.

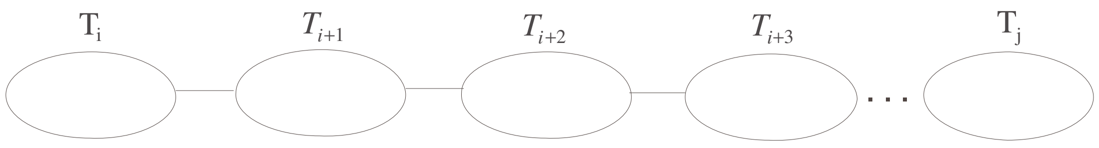


Figure 5.14. A minimal set of interrelationships of the types of Figure 5.13.

The fact that some of the interrelationships can be derived tends to decrease the size of AdSchema. So, we need to find the optimal combination of the interrelationships to form the base set from which the remaining relations can be derived.

If we can find a relation combination that is minimal and valid, then the rest of the interrelationships can be derived from the known interrelationships and the database schema can be reduced without loosing its accuracy. In Wanne (1998) the validity of a relation combination is defined between the elements of adjacent types but here we just notice that if a relation combination is valid, then it uniquely defines the interrelationships of all types.

In Figure 5.15 we have a schema with six relations, $\mathcal{R} = \{R_1, R_2, R_3, R_4, R_5, R_6\}$, i.e. $R_1 = (T_1, \{T_2, T_4\})$, $R_2 = (T_2, \{T_1, T_3, T_4\})$, $R_3 = (T_3, \{T_2\})$, $R_4 = (T_4, \{T_1, T_2, T_5, T_6\})$, $R_5 = (T_5, \{T_4\})$ and $R_6 = (T_6, \{T_4\})$. The combination of three relations R_1, R_2 and R_4 form a valid combination. This means that the rest of the relations can be derived with the help of these relations.

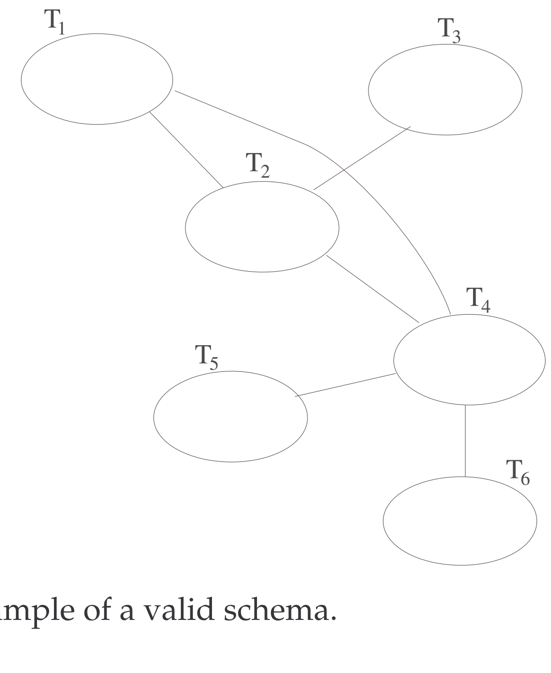


Figure 5.15. An example of a valid schema.

When we think of the relation combinations that should be stored we sometimes have to select between two or more alternatives. In Figure 5.16 we have such a situation. There are three different relation combinations from which we can choose. First we have the relations $R = \{R_1, R_2, R_4\}$, where $R_1 = (T_2, \{T_1\})$, $R_2 = (T_2, \{T_3\})$ and $R_4 = (T_2, \{T_4\})$, the second relation combination is $R' = \{R_2', R_3', R_4'\}$ where the relations are $R_2' = (T_1, \{T_3\})$, $R_3' = (T_3, \{T_2\})$ and $R_4' = (T_2, \{T_4\})$, and the third $R'' = \{R_1'', R_3'', R_4''\}$, where the relations are $R_1'' = (T_1, \{T_2\})$, $R_3'' = (T_1, \{T_3\})$ and $R_4'' = (T_2, \{T_4\})$. The choice between these relation combinations must be done with respect to the efficiency aspect. The most efficient is the relation combination R i.e. $R = \{R_1, R_2, R_4\}$. With this solution the interrelationship between the types T_1 and T_3 can be left out of the schema, because that particular interrelationship can be defined with respect to the type T_2, which is an interrelationship defining type.

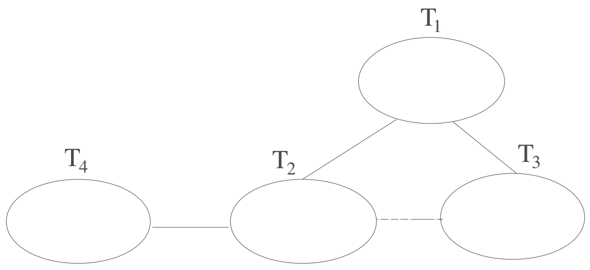


Figure 5.16. Selection problem of the minimal valid relation combination.

This relation combination is the most efficient because if we start from T_2 we can retrieve all the types T_1, T_3 and T_4 with one direct access.

5.6. Examples

In this Section we give some examples of how different parts of a semistructured database can be depicted with an AdSchema. In Figure 3.1 we have a semistructured database called *premiership* (Nestorov et al. 1997) some parts of which we presented with the help of an AdSchema.

First, in Figure 5.17a we have the first part of the premiership object starting from the root node *&0*. It presents a data path *&0.Club.&1.Name.&2.Offical.&3*. This data path can be constructed with an AdSchema depicted in Figure 5.17b consisting of three types *Club*, *Name* and *Official* and the relations between them.

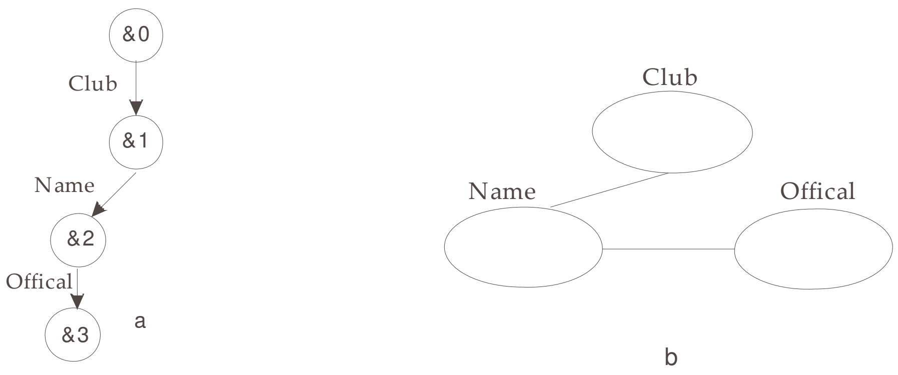


Figure 5.17. A portion of a graph of Figure 2.1 depicted with an AdSchema

Next we will extend Figure 5.17a with a new type called *Player* and its successor *Nationality* (Figure 5.18a). The second data path is *&0.Club.&1.Player.&14. Nationality.&19*. *Player* is the second edge emanating from the edge *Club*, which means that we have to insert the type *Player* into the schema as well as its successor *Nationality* (Figure 5.18b).

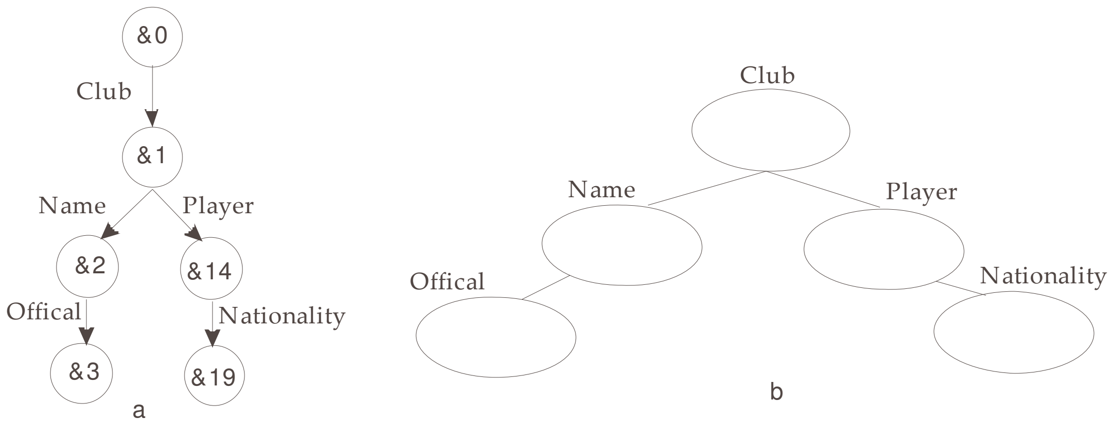


Figure 5.18. Adjacency of types *Name* and *Player* defined with respect to a third type *Club*.

The types *Club*, *Name* and *Player* are interrelationship defining types, which means that we do not need to draw an interrelationship between the types *Name* and *Player*. Besides, we need not to have the interrelationships between the types *Club* and *Nationality* or between the types *Club* and *Official* either because they are also interrelated with each other with respect to the types *Player* and *Name*, respectively.

In Figure 5.19a the edge *Name* has two different expressions. In the first case it is a complex object and in the second case it is an atomic object. Both of these expressions are allowed in an OEM graph. This deviation does not pose any problem in AdSchema, because we have only one type for each occurrence of the *Name* object. The corresponding AdSchema is depicted in Figure 5.19b, in which we can see that the types *Club* and *Name* are again interrelationship defining types as in Figure 5.18b, although the situation is somewhat different.

This is one characteristic of the AdSchema that makes it possible to increase the degree of the data structure, i.e. some relations between the types can be derived from the existing, stored interrelationships.

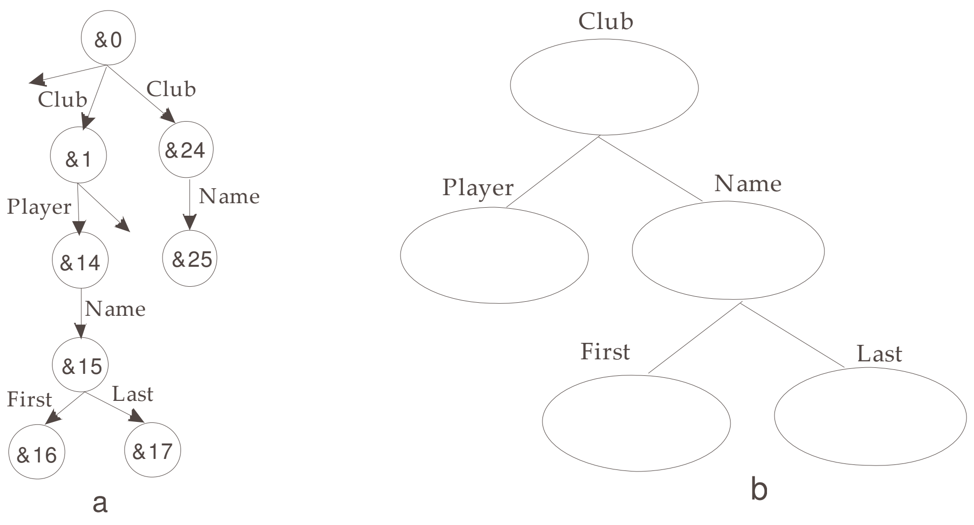


Figure 5.19. Two different occurrences of the edge *Name* depicted with an AdSchema.

In Figure 5.19b the interrelationships $R_3 = (Name, \{Player\})$ and $R_4 = (Player, \{Name\})$ can be derived with the help of the interrelationships $R_1 = (C\text{lub}, \{Name\})$ and $R_2 = (C\text{lub}, \{Player\})$. We can also use the symmetric variants of the named interrelationships R_1 and R_2. Also the interrelationship between the types *First* and *Last* can be derived based on the same definition.

Next we will consider cycles. In Figure 5.20b we can see how a cycle of a *premiership* object (Nestorov et al. 1997) is implemeted with the AdSchema. We can see that the types *Club* and *FormerClub* are interrelated to each other with respect to the type *Player*. There is also a interrelationship between the types *Player* and *FormerClub*, but we need not to show it in the AdSchema. The reason is the interrelationship defining set *Player*. It can be used to derive the missing interrelationship between the types *Club* and *FormerClub*.

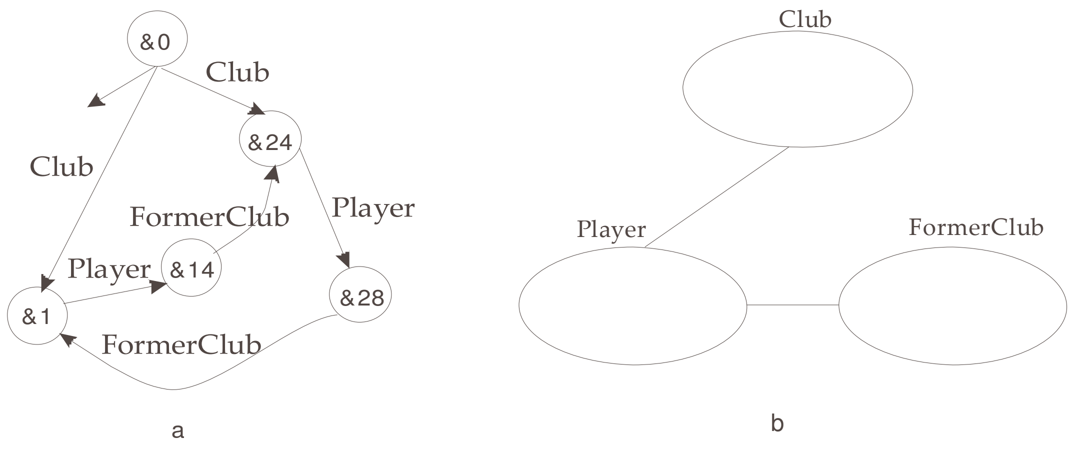
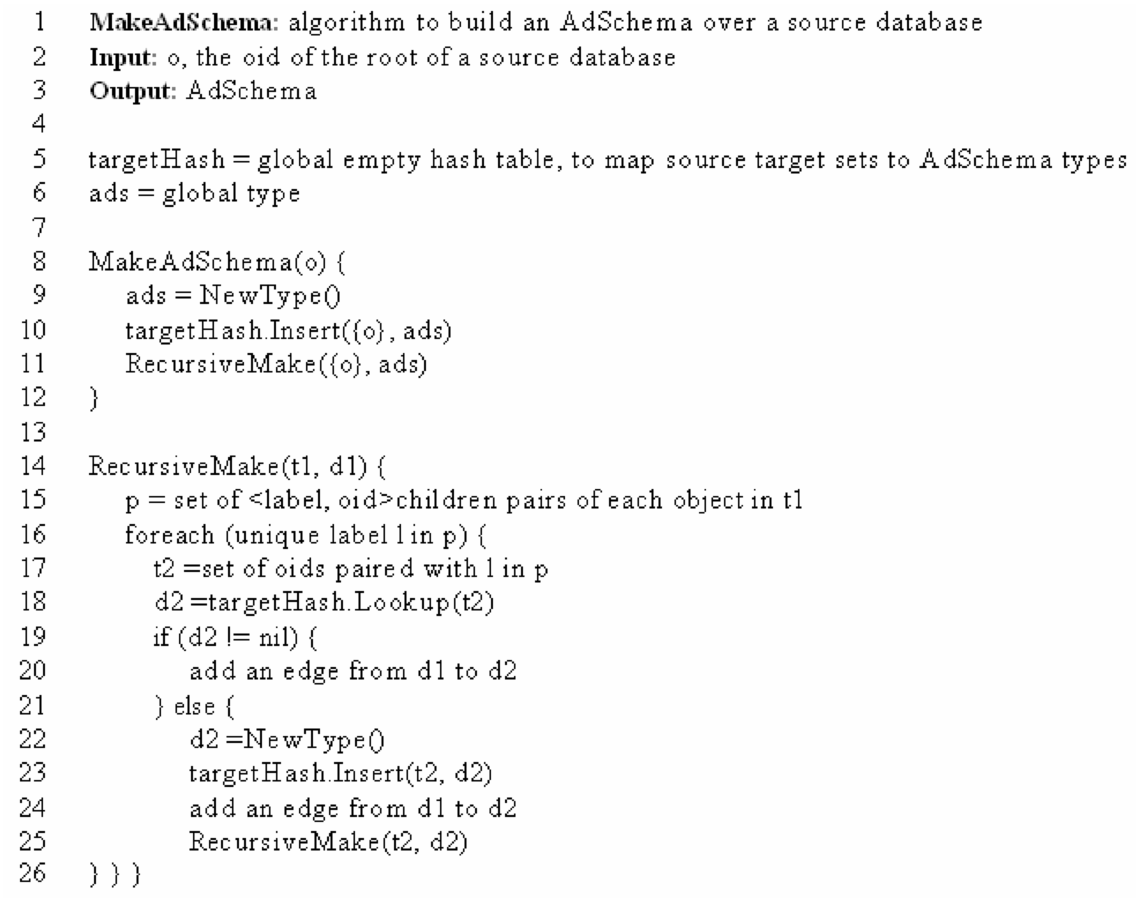


Figure 5.20. Representing cycles with the AdSchema.

5.7 Creating an AdSchema from Semistructured Data

In this Section we present an algorithm for schema creation for semistructured data. The original source database is read in a depth-first fashion. Every label path is considered and all distinct edges are read and new types are created from them. Equally all the interrelationships between the types are created of the nodes of the source database. We use a hash table where all the object and edges are stored. The hash table is used in order to prevent multiple types with same name to be inserted into our AdSchema. We also create a starting type (usually the incoming edge to the root node) because in this way we can avoid disconnected types. Next we go through the steps of the algorithm (Figure 5.21).

The first six lines represent the initialization phase of our algorithm. On line 1 we have the name of the algorithm. Line 2 contains the input file to our algorithm. It is a semistructured database and will have the object identifier of the root node *o* of the source database as a starting point. On line 3 the output file is introduced. It is the AdSchema. On line 5 a hash table is declared . The purpose of the hash table is to map the objects/edges of the source database to the types of the AdSchema. On line 6 a global type called *ads* is created. This variable contains the first type that will be inserted into the new AdSchema. Usually it is an edge that comes into the root node.



```
1   MakeAdSchema: algorithm to build an AdSchema over a source database
2   Input: o, the oid of the root of a source database
3   Output: AdSchema
4
5   targetHash = global empty hash table, to map source target sets to AdSchema types
6   ads = global type
7
8   MakeAdSchema(o) {
9       ads = NewType()
10      targetHash.Insert({o}, ads)
11      RecursiveMake({o}, ads)
12  }
13
14  RecursiveMake(t1, d1) {
15      p = set of <label, oid>children pairs of each object in t1
16      foreach (unique label l in p) {
17         t2 =set of oids paired with l in p
18         d2 =targetHash.Lookup(t2)
19         if (d2 != nil) {
20             add an edge from d1 to d2
21         } else {
22             d2 =NewType()
23             targetHash.Insert(t2, d2)
24             add an edge from d1 to d2
25             RecursiveMake(t2, d2)
26  } } }
```

Figure 5.21. An algorithm to create an Adjacency Schema.

Line 8 starts the making of the AdSchema. The method *MakeAdSchema* contains the object identifier of the root node as a parameter. On line 9, the first new type, i.e. the label of the first edge is assigned into the *ads* variable. Line 10 makes the first insertion into the hash table with the method *Insert*() of the object *targetHash*. The insertion contains the object identifier of the root node and the label of the first edge. After that, on line 11, we call the recursive part of that algorithm called *RecursiveMake*().

The effective work is done on lines 14 through 26. On these lines all the nodes and edges of the source database are stored in the hash table, and above all, the types and the interrelationships between them are appended to the current schema.

Line 14 is the header of the *RecursiveMake* method. It contains as parameters the object identifier and the name of the edge. When we call this method for the first time, the object identifier is the identifier of the root node and the name of the edge is the label of the incoming edge to the root node. On line 15 we save all the object identifier and edge label pairs of the children of each object in *t1*.

On lines 16 through 26 we have a for-each loop that searches through the hash table and checks if the given object identifiers are already stored there. In a semistructured database there can be several edges with the same edge label but with different object identifiers. On line 16 we consider unique edge labels but take into consideration all the object identifiers. Line 16 starts the for-each loop. On line 17 the object identifiers of the edge label are picked up and stored to the variable *t2*, and on line 18 we search the hash table for the object identifiers of the variable *t2*. The used method of the object *targetHash* is called *Lookup*(). The result of the search is stored to the variable *d2*.

Lines 19 through 26 check if the given objects are found in the hash table or not. If an object identifier is found we add on line 20 an edge from the second last inserted type to the last inserted type. On the other hand, if we cannot find the search object from the hash table (line 21), we add a new type to the schema (line 22). This new type will then be added to the hash table on line 23. Also the set of object identifiers assigned to that edge are added to the hash table. After that, on line 24, a new interrelationship is added between the two last added types. Finally, a new recursive call is made on line 25, and all the rest of the types and the interrelationships between them are added to the schema.

Next we will consider an example of how our algorithm works. In Figure 5.22 there is a part of an OEM database of restaurants and bars (Goldman 2000). There are two *Restaurants* with some detailed information and one *Bar* without any additional information. The *Owner* of one restaurant is the same as the *Manager* of the other restaurant. The leaf nodes have some data.

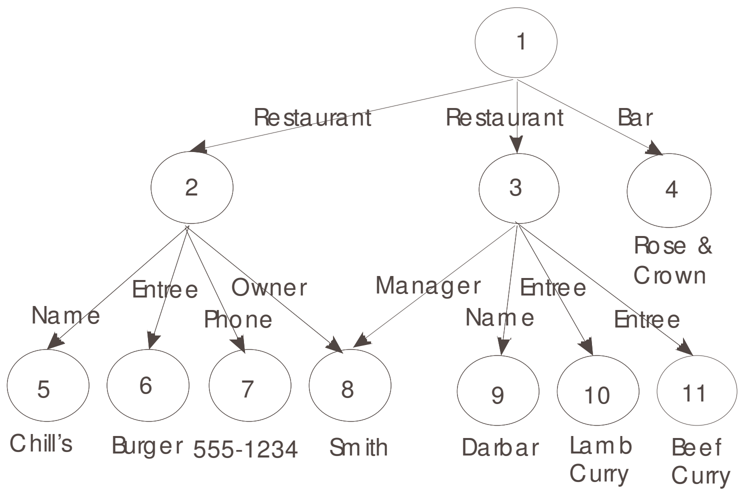


Figure 5.22. A sample OEM database of restaurants and bars (Goldman 2000).

In Figure 5.23 we have the corresponding AdSchema of Figure 5.22. When we create an AdSchema of Figure 5.22 we can see that there are as many types in it as there are distinct edges in DataGuide. In addition, our AdSchema has an extra type (*Entry*) which does not exist in the DataGuide or in the original source database.

In our algorithm the source database with an object identifier *o* contains the input data. The output of the algorithm is the schema of Figure 5.23. First, in the method *MakeAdSchema*(), we assign the type *Entry* as a new type to the variable *ads*. We insert the type *Entry* and the object identifier 1 to the hash table and start the *RecursiveMake*() method.

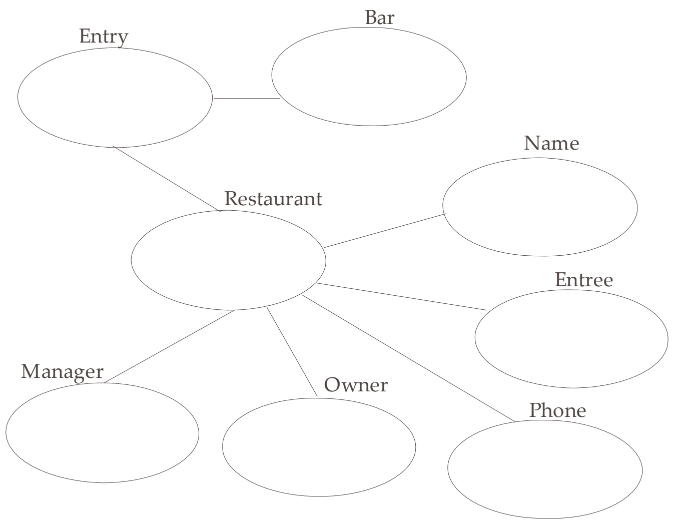


Figure 5.23. An Adjacency Schema of the sample OEM database of Figure 5.22.

When we enter to the method *RecursiveMake*() we assign the <label, object identifier> pairs {(*Restaurant*,2),(*Restaurant*,3)} to the variable *p*. After that, for each unique label in *p*, we assign the object identifiers to the variable *t*2. *t*2 will contain the set {2,3}. Next we will use the method *Lookup*() of the object *targetHash* to lookup if the objects are already in the hash table. The result will be assigned to the variable *d*2. If the objects are not found, then *d*2 will be assigned to the next new type, which in this case will be *Restaurant*. The object identifiers and the new type will then be added to the hash table. This will again be done with the *Insert*() method. A new interrelationship is added between the *Entry* type and the *Restaurant* type, and the *RecursiveMake*() method will be called again with new parameters.

In this second call of the method *t*1 contains the object identifiers {2,3} and *d*1 contains the type *Restaurant*. The variable *p* will now contain the following <label, object identifier> pairs: {(*Name*,5),(*Entree*,6),(*Phone*,7),(*Owner*,8),(*Manager*, 8),(*Name*,9),(*Entree*,10),(*Entree*,11)}. Now, for each unique label, we go through the same procedure. We start with label *Name* and assign the object identifiers (5,9) to variable *t*2, try to find them in the hash table and if not found, we create a new type, add the type and the object identifiers to the hash table and finally create a new relation between the second last and the last types. This procedure will go on until all unique edges and the interrelationships between them are added to the AdSchema.

This example is a very simple example and its only purpose is to show that the algorithm works. The only speciality is that the two edges *Owner* and *Manager* present one and the same person. In the source database this means that the corresponding edges end to the same node. In the AdSchema this can be presented without any specific structural problem.

6. CONCLUSIONS AND FUTURE WORK

6.1. Conclusions

Our research hypotheses stated that it is possible to construct a schema for semistructured data based on the theory of the Adjacency Relation Systems, and that the size of it can be decreased without loosing much of its accuracy. As we have finished our work and given answers to the research questions we can state the following.

The first research question about the adequacy of the theory of the Adjacency Relation Systems to be used as a foundation for an Adjacency Schema was answered positively. This means that the theory is solid and can be used with some smaller changes also with a schema construction for semistructured data. It also turned out, as we stated in the second research question, that it is possible to develop an algorithm that generates the schema from the original source database.

The third question about the fulfillment of the requirements imposed for a schema for semistructured data is partly answered, but this question is still open. We need empirical results to answer this question in full, so in future this is one of the areas we will concentrate.

The fourth research question about the minimal and accurate schema is also partly open. The schema creation algorithm constructs a schema that contains only distinct types which means that the schema is minimal. In most cases it also shares the idea of a strong DataGuide in that there is only one “label path” in the adjacency schema to each of the types.

6.2. Summary of Contributions

The main contributions of this work are the schema proposal for semistructured data, an accurate definition of semistructured data and an algorithm for calculating its degree as well as a general query for a given relation combination. We also contribute with algorithms for the modelling of relation and semistructured data as well as with an algorithm for construction of an adjacency schema for semistructured data.

The accurate definition of the semistructured data helps us to determine the degree of the semistructured data. It states clearly that the degree is between zero and one, and that it can be completed up to one, which means that the structure is valid. The degree calculating algorithm connects our work to the theory of the Adjacency Relation Systems. It calculates the degree from a set of given relations and their adjacency defining sets. The calculated degree can furthermore be increased if we use the known relations to derive new ones. The definition of semistructured data and the calculation algorithm show that the theory of Adjacency Relation Systems could be applied in our work.

The general query presented in this work is a first attempt to develop a query that could use the structure of the adjacency schema to find the information wanted by a user. However, the general query has not been developed to the point in which the whole benefit of it could be used.

The main contribution of this paper is the schema proposal for semistructured data. This proposal follows the same principles imposed for the existing schemas, like DataGuide. The results we have obtained in our investigations are very encouraging. We have shown that the adjacency schema can be used as a schema for semistructured data, however the test examples could be more numerous. This means that more extensive conclusions should not be drawn before more support for our investigations can be shown.

The schema construction algorithm presented in this work is important because it shows that the theory behind the adjacency schema can be utilized in practice. This statement gives us a good base for our future work in this area. The other algorithms presented here support the same thing although they consider the modelling of semistructured data.

If we compare our results with the results obtained by other researchers there are two points that are worth mentioning. First we can say that as far as we know there has not been an exact definition of semistructured data which is general enough. In our work we give the limits between which semistructured data should lie and a motivation for such a convention. Secondly is the schema for semistructured data. As we have described in the earlier sections of this work there is one schema for semistructured data that has reached the position of a de facto standard. Compared with that schema it is hard to come up with new proposals. However, we have presented our proposal because we believe that there is potential in it. There are aspects in the current schemas that need improvements and when we have developed the theory of the adjacency schema we have emphasized these facts.

6.3. Future Research

As it has tuned out in the earlier sections there are some subjects that need more and deeper investigation. The presented schema proposal is our first attempt to give a structure for semistructured data but we lack some experimental results of the effectiveness of our proposal. It will be one of the first things to be considered in future.

Another thing that has not been considered in our work is the modification of the schema. The schema must be consistent even when the source database changes. These changes in the structure of the schema must be handled with the construction algorithm. In this stage our algorithm does not handle changes in the structure of the source database so the second subject to be covered in future is the algorithm. Furthermore, the algorithm should also be able to handle different versions of semistructured data which is not the case right now. The algorithm must be made more flexible so that it can handle cycles and other exceptions in the structure of the source database.

Finally, the subject of queries has not been considered very much in this work. In the adjacency relation systems there are queries whose appropriateness should be tested in this context. As the queries are relatively simple there should not be much work to adapt them to be used with the adjacency schema. However, modification of these queries will be on the list of our future work.

REFERENCES

Abiteboul, S. (1997). Querying Semi-Structured Data. *In Proceedings of International Conference on Database Theory* [online] [cited: 2005-06-01], 1–18. Available: ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-103.ps.gz.

Abiteboul, S., P. Buneman & D. Suciu (2000). *Data on the Web. From Relations to Semistructured Data and XML*. San Francisco, USA: Morgan Kaufman Publishers. 258 p. ISBN 1-55860-622-X.

Broekstra, J., C. Fluit, & F. van Harmelen (2000). *The State of the Art on Representation and Query Languages for Semistructured Data*. On-To-Knowledge (EU-IST-199910132) [online] [cited: 2005-06-01]. Available: http://www.ontoknowledge.org/downl/del8.pdf.

Buneman, P. (1997). Semistructured Data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* [online] [cited: 2005-06-01], 117–121. Available: http://db.cis.upenn.edu/DL/97/Tutorial-Peter/tutorial-semi-pods.ps.gz.

Buneman, P., S. Davidson, G. Hillebrand & D. Suciu (1996). A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* [online] [cited: 2005-06-01], 505–516. Available: http://db.cis.upenn.edu/DL/96/UnQL/UnQL.ps.gz.

Buneman, P., S. Davidson, M. Fernandez & D. Suciu (1997). Adding Structure to Unstructured Data. In *Proceedings of the International Conference on Database Theory* [online] [cited: 2005-06-01], 336–350. Available: http://www.cis.upenn.edu/~db/sue/ICDT96.pdf.

Buneman, P., M. Fernandez & D. Suciu (2000). UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. In *Very Large Data Bases, VLDB Journal* [online] [cited: 2005-06-01]. Available: http://db.cis.upenn.edu/DL/vldbjournal00.ps.gz.

Cong, G., L. Yi, B. Liu & K. Wang (2002). Discovering Frequent Substructures from Hierarchical Semi-structured Data. In *Proceedings of the Second SIAM International Conference on Data Mining, SDM.* Hyatt Regency, Crystal City, Arlington, VA, USA [online] [cited: 2005-06-01]. Available: http://www.cs.sfu.ca/~wangk/pub/siam02.pdf.

Goldman, R. (2000). *Integrated Query and Search of Databases, XML, and the Web*. Thesis. Stanford University, Department of Computer Science [online] [cited: 2005-06-01]. Available: http://infolab.stanford.edu/~widom/theses/goldman.ps.

Goldman, R. & J. Widom (1997). DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases* [online] [cited: 2005-06-01], 436–445. Available: http://www-db.stanford.edu/pub/papers/dataguide_vldb97.ps.

Goldman, R., J. McHugh & J. Widom (1999). From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proceedings of the Second International Workshop on the Web and Databases (WebDB'99)* [online] [cited: 2006-06-01], 25–30. Available: http://www-db.stanford.edu/pub/papers/xml.ps.

Goldman, R. & J. Widom (1999). Approximate DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*. Jerusalem, Israel [online] [cited: 2005-06-02]. Available: http://dbpubs.stanford.edu/pub/showDoc.Fulltext?lang=en&doc=1999-56&format=pdf&compression=&name=1999-56.pdf.

Goodrich, M. & R. Tamassia (1998). *Data Structures and Algorithms in Java*. New York etc.: John Wiley & Sons, Inc. 738 p. ISBN 0-471-19308-9.

Hacid, M-S., F. Soualmia & F. Toumani (2000). Schema Extraction for Semistructured Data. In *Proceedings of the 2000 International Workshop on Description Logics (DL'2000)*. Aachen, Germany [online] [cited: 2005-06-01], 133–142. Available: http://www710.univ-lyon1.fr/~dbkrr/publications.htm.

Hopcroft, J. E. & J. D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Menlo Park, California etc.: Addison-Wesley. 418 p. ISBN 0-201-02988-X.

Linna, M., M. Wanne & J. Töyli (2003). Completion of the Incomplete Data Structure. In *Proceedings of the Second IASTED International Conference on Information and Knowledge Sharing*, Scottsdale, AZ, USA [online] [cited: 2005-06-02]. Available: http://www.uwasa.fi/~jt/IKS_2003.pdf.

McHugh, J., S. Abiteboul, R. Goldman, D. Quass & J. Widom (1997). Lore: A Database Management System for Semistructured Data. *SIGMOD Record* 26:3 [online] [cited: 2005-06-01], 54–66. Available: http://www-db.stanford.edu/pub/papers/lore97.ps.

McHugh, J., J. Widom, S. Abiteboul, Q. Luo & A. Rajaraman (1998). *Indexing Semistructured Data*. Technical Report, the Stanford Database Group [online] [cited: 2005-06-03]. Available: http://www-db.stanford.edu /lore/pubs/semiindexing98.pdf.

Nestorov, S., J. Ullman, J. Wiener & S. Chawathe (1997a). Representative Objects: Concise Representations of Semistructured, Hierarchical Data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, Birmingham, England [online] [cited: 2005-06-01]. Available: http://www-db.stanford.edu/pub/papers/represen tative-object.ps.

Nestorov, S., S. Abiteboul & R. Motwani (1997b). Inferring Structure in Semistructured Data. *ACM SIGMOD Record* 26:4 [online] [cited: 2005-06-01], 39–43. Available: ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-122.ps.gz.

Nestorov, S., S. Abiteboul & R. Motwani (1998). Extracting Schema from Semistructured Data. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* 27:2 [online] [cited: 2005-06-01]. Available: ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-152.ps.gz.

Ni, X & S. Bloor (1994). Performance Evaluation of Boundary Data Structures. *IEEE Computer Graphics and Applications* 14:6 [online] [cited: 2005-06-02], 66–77. Available: http://ieeexplore.ieee.org/iel1/38/7784/0032 9098.pdf?tp=&arnumber=329098&isnumber=7784.

Papakonstantinou, Y., H. Garcia-Molina & J. Widom (1995). Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Taipei, Taiwan [online] [cited: 2005-06-02]. Available: http://dbpubs.stanford.edu/pub/showDoc.Fulltext?lang=en&doc=1995-6&format=pdf&compre ssion=&name=1995-6.pdf

Thierry-Mieg, J. & R. Durbin (1992). *Syntactic Definitions for the ACeDB Data Base Manager*. Technical Report, MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge, UK.

Töyli, J. (2002). *Modeling Semistructured Data by the Adjacency Model*. Licentiate thesis, University of Vaasa [online] [cited: 2005-06-02]. Available: http://www.uwasa.fi/~jt/lisuri.pdf.

Töyli, J., M. Linna & M. Wanne (2002a). Modeling Relational Data by the Adjacency Model. In *Proceedings of the Fourth International Conference on Enterprise Information Systems*. Universidad de Castilla-La Mancha, Ciudad Real - Spain [online] [cited: 2005-06-02], 296–301. Available: http://www.uwasa.fi/~jt/ICEIS_2002.pdf.

Töyli, J., M. Linna & M. Wanne (2002b). Modeling Semistructured Data by the Adjacency Model. In *Proceedings of the Fifth Joint Conference on Knowledge-Based Software Engineering*. University of Maribor, Maribor - Slovenia [online] [cited: 2005-06-02], 282–290. Available: http://www.uwasa.fi/~jt/JCKBSE_2002.pdf.

Wang, K. & H.Q. Liu (1997). Schema Discovery for Semistructured Data. In *Proceeding of the International Conference on Knowledge Discovery and Data Mining*. Newport Beach, AAAI [online] [cited: 2005-06-01], 271–274. Available: http://www.cs.sfu.ca/~wangk/pub/kdd1.ps.

Wang, K. & H.Q. Liu (1998a). Discovering Typical Structures of Documents: a Road Map Approach. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* [online] [cited: 2005-06-01], 146–154. Available: http://www.cs.sfu.ca/~wangk/pub/sigir.pdf.

Wang, K. & H.Q. Liu (2000a). Discovering Structural Association of Semistructured Data. *IEEE Transactions on Knowledge and Data Engineering,* 12:3 [online] [cited: 2005-06-01], 353–371. Available: http://www.cs.sfu.ca/~wangk/pub/tkde98.pdf.

Wang, Q.Y., J. X. Yu & K. F. Wong (2000b). Approximate Graph Schema Extraction for Semi-Structured Data. In *Proceedings of Seventh International Conference on Extending Database Technology (EDBT)*. 302–316. Germany: Springer, Konstanz.

Wanne, M. (1998). Adjacency Relation Systems. *Acta Wasaensia 60. Computer Science 1*. University of Vaasa, Vaasa.

Wanne, M. & M. Linna (1999). A General Model for Adjacency. *Fundamenta Informaticae* 38, 39–50.

World Wide Web Consortium (2004). Extensible markup Language (XML) 1.1. [online] [cited:2006-01-24] Available: http://www.w3.org/TR/2004/REC-xml11-20040204/.