A DATA MODEL BASED ON SEMANTICS OF PROPERTIES

by

Eran Rubin

B.Sc., Technion – Institute of Technology, 1999

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In

THE FACULTY OF GRADUATE STUDIES

(Faculty of Commerce and Business Administration)

We accept this thesis as conforming
To the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 2002

© Eran Rubin, 2002
Abstract

It is well agreed that the conventional database models provide limited semantic expressiveness. These models are designed to enable mapping to simple data constructs, which provide for the expression of only a limited part of the domain knowledge. However, one of the reasons for the proliferation of the entity-relational model is the ease in which the mapping of logical structures to physical structures is done. In this paper we suggest a data model which gives new means for expressing the semantics of data. We demonstrate how this data model can be mapped to the traditional relational model in order to demonstrate its practical appeal. In constructing our model we use the framework of BWW ontology, and emphasis two of its principles. The first principle we follow is that often properties share an important relationship: the relation of precedence. We believe that enabling users to express precedence information and assuring precedence consistency can provide important means for semantic expressiveness. The second principle we follow, is that of separating classification information from entity information. By assuring this ontological principle we enable different users to have different views of the world while sharing data. The combination of these two principles in the data model provides interoperability and evolutionary benefits. Users can concentrate on the commonalities between their view of the enterprise and views of other users. Updates to the database are propagated across different views, and the meaning of information can be extracted. This paper analyses theoretical issues concerning the need for such a model, as well implementation issues such as Meta data constructs, DDL/DML commands, and implicit operations that may be done in the DBMS in order to support such a data model.
## Contents

1 Introduction 1  

2 Background 3  

2.1 The relational model 4  
2.1.1 The Entity-Relationship data model 5  
2.2 The hierarchical data model 7  
2.3 The network data model 8  
2.4 Limitations of the traditional data models 8  
2.5 The Two-Tiered Approach 11  
2.6 Related Research - Different Approaches 12  
2.6.1 Semantic data models 13  
2.6.2 Object oriented data models 14  
2.6.3 Deductive databases 16  
2.6.4 Semi-structured data models 17  
2.6.5 Other research directions 18  

3 Towards a new semantic data model 19  
3.1 Motivation and Approach 19  
3.2 Overview of the data model 23  
3.3 General Example 24  
3.3.1 Example with diagram 25  

4 The proposed data model 31  
4.1 The meaning of properties and precedence 31  
4.2 Formalization 35  

5 Benefits of the model 44  

6 Implementation Issues 48  
6.1 DBMS architecture using the data model 48  
6.2 New Meta-data constructs 49  
6.2.1 Overview 50  
6.2.2 The constructs 51  
6.2.3 An example of an instance of the model with the corresponding relations 57  
6.3 DDL for a DBMS using the data model 58  
6.3.1 Defining fully preceded property-views 60  
6.3.2 Defining partially preceded property-views 63  
6.3.3 Defining a property view which represents a relational/mutual attribute 66  
6.3.4 Defining low-level property views 68
List of Figures

1 An instance of an ER data model. This instance models customers, accounts, branches, and the 'customer at branch' relationship. .................................................. 6
2 The hierarchical form of the ER model in figure 1 .................................................. 7
3 An instance of the data model in which different users define the classes 'course' and 'student' ................................................................. 29
4 An instance of the data model in which different users look at different properties. All properties relate through precedence information ........................................... 30
5 An instance of the data model which supports the properties 'has_weight', 'has_height', and 'has_weight_at_height' ............................................................. 58
6 Relations populated with tuples in an instance of the data model ......................... 59
7 An example of different property views and their relations. These property views can be used to extract academic information on a student ............................... 78
List of Tables

1  Attributes and meanings in $r_{HLPVDefine}$, a Meta-data relation keeping data of definition of property views .................................................. 53
2  Attributes and meaning of $r_{IsClass}$, a Meta data relation keeping information of which property view defines a class .................................................. 53
3  Attributes and meaning of $r_{HUID(PV)}$ a relation holding instances of a higher-level property view .................................................. 55
1 Introduction

A database is a collection of stored operational data that serves the needs of multiple users. A data model is a mechanism for specifying the structure of a database. In other words, a database is an instance of a specific data model [King, 1995]. Conventional database models such as the hierarchical, network and relational database models cannot capture the meaning of the phenomena they represent as clearly as conceptual models do. This phenomenon is reasonable since data models are more implementation dependent than conceptual models are. However, we believe there is a way to narrow the gap between the two models. Moreover, we believe that taking research disciplines used in conceptual modeling to the filed of data modeling would derive many benefits.

The process of developing a database structure from user requirements is called database design [Teorery et al. 1982]. Database design can be decomposed into three successive phases: conceptual, logical and physical design. In the design process, conceptual models are used to produce a high-level description of reality. Conceptual models provide a description of reality, which is easy to understand and communicate about. The logical phase of database design is the phase in which data models play a significant role. Data models support a description of reality which can be processed by a computer. In a data model there should be a clear description how data is accessed, what format of data is stored, and how it can be manipulated. Making this representation close to the representation of the conceptual model would make the translation process between the models easier and more comprehensive. The meaning of phenomena captured in the database would be more coherent, and changes to the database would be easier to process.

Database design concerns have to do with integrity, consistency, recovery, efficiency, and changeability [Vossen, 1991]. It is clear that these concerns are approached either at the logical phase or the physical phase since the conceptual phase is completely independent of any implementation issue. However, the design phase in which these different concerns are considered is not well defined. The general approach is that performance issues are mainly
examined at the physical level, although some indication of performance may be absorbed at the logical level as well. However, many of the problems facilitated at the conceptual phase, such as understandability, are also facilitated at the logical phase. Understandability, evolvability, completeness, and correctness are all of major concern in data modeling.

In this paper we will suggest a new data model which is constructed according to one of the major guidelines used to evaluate conceptual models- ontological expressiveness. We describe the different constructs that are incorporated into the data model in order to derive many benefits which are missing in other data models. We provide a data model in which meaning of data is more easily derived, design guidelines are more accurately defined, and evolvability is better supported.

The research question guiding our paper is whether the two layered approach described in [Parsons and Wand, 2000] can be implemented. We further extend this question to examine what are the major constructs needed to support such an approach and what advantages other than those mentioned in [Parsons and Wand, 2000] can be derived. We examine a way to extend the two-tiered approach and incorporate it into an applicable data model. We wish to formalize the constructs needed to support this approach, and suggest a method to implement this approach.

The objective of our data model is to support the two layered approach, which would be outlined in the next section. We suggest ways to extend it to solve many of the problems apparent in traditional data models as well as in many semantic data models. The major idea in the two tiered approach is that of separating property information from class information. In other words, things can possess properties without belonging to any specific class. In order to support this idea we extend the abilities of the data model and keep information of the semantics of properties. Namely, our data model preserves precedence information of properties. Preserving this semantic information of properties enables the data model to support different classification schemes by different users.

The proposed contribution of our work is in providing a definition of fundamental constructs needed to support such an approach. We further identify new constructs, and provide
new ideas to support integrity in such a model. We suggest a way to enable different classification schemes by different users to co-exist. We also provide a practical way to capture the semantic of precedence information— which properties must be possessed by an object in order to possess another property. We further find that facilitating the two tiered approach results in a data model which better reflects the meaning of data and thus is closer to the way users think about data.

2 Background

Data models are distinguished according to the manner in which they treat relationships among objects in the database. The three conventional data models are the hierarchical model, the network model and the relational model. More recent data models are semantic data models, deductive data models, and semi structured data models. Finally, there are also many data models which fall under the category of research approaches. Each of these models has its strengths and weaknesses and there is no "one true data model". Data models are usually evaluated according to 'data independence' and according to 'extensibility'. There is no generally accepted definition for 'data independence', nor is there any available metric to measure it [Clemons et al, 1995]. The idea is that data independence is a measure of the database system's ability to change representation or content without affecting programs. In the limit, under perfect data independence, a change that does not explicitly remove data needed by an application program will not affect the operations of a program or its production of correct results, except perhaps for changes in performance efficiency. It is well accepted that the cost of restructuring a schema and a database is almost always far less than the cost of revising application code [Astrahan 1976].'Extensibility' is not precisely defined either. Generally, it is intended to measure the ease with which new application programs can be added to the set of applications supported by a database. In this section we will outline the general assumptions imposed by each data model, how they treat relationships, and their implications on extensibility and data independence.
2.1 The relational model

One of the data models which is most widely referred to and used is the relational model. This model provides conceptual simplicity and is based on a rich theoretical foundation - the theory of sets and relations. The bases of relational model is the theory of data dependencies. Relational systems store all information as normalized relations [Lien, 1995]. Codd, which introduced the relational model, identified a certain set of anomalies which may be found in complex databases. In this context the concepts of functional dependencies and normal forms were introduced. The basic constructs in the relational model are the following [Lien, 1995]:

- An attribute is a logical description of some characteristic of an entity. For example: Name
- An attribute-value is the physical contents of the attribute. For example, 'John'.
- A tuple is a set of assigned attribute-values to a set of attributes. For example, assignment of 'John', '14', 'Vancouver' to Name, Age, City respectively.
- A relation is a set of tuples.
- A relation-scheme is the set of attributes for which attribute-values are assigned in order to produce a relation.

The relational model can generally be viewed as a set of two dimensional tables. Each table corresponds to a relation. Rows correspond to tuples, and columns correspond to attributes. In the relational model, there is no deep notion of semantics. The relational model's weak points include its lack of ability to capture underlying semantics of entities and their relationships. Another limitation of this model is that the fragmentation of data into normalized relations makes it necessary to write tedious and lengthy queries to obtain useful information.
The relational model derives its appeal due to its rich mathematical base, and provide only limited semantic notion - that of properties, and property values. A semantic models which augments the relational model with more semantics is the ER model which is described next.

2.1.1 The Entity-Relationship data model

The Entity-Relationship (ER) data model is the most widely used semantic data model. As in the relational model, all information as normalized relations\(^1\). The relational model introduces two basic notions: entities and relations. Entities are "well-distinguishable objects which exist in the real world" [Vossen, 1991]. Relations describe association between entities, or more precisely between entity sets. In this model, entity-sets are viewed as essentially fixed-format tables or matrices. Entity sets have attributes, which are the columns of the matrices, and entities that belong to the set, which are the rows of the matrix. A general assumption about the entity sets, which constrains extensibility, is that while the entities belonging to an entity set are time-variant, the name of the entity set and its attributes are considered time-invariant.

A relation is viewed in a similar manner to an entity set. A relationship can have attributes of its own in order to capture special properties that have meaning only for the relationship, but not for any of the entity sets involved. Each relationship is described by a matrix with columns representing attributes of the relationship, and rows representing instances of the relationship. An example of the ER model is given in figure 1.

Finally, the ER model does not provide much data independence or extensibility. For example, a need to change a one-to-many relationship to a many-to-many relationship may require change in the model in a manner that the relationship is necessary represented as a separate relation rather than a property of an entity. Consider the property street in figure 1. If a customer could have many houses, and thus many streets, every customer would either possess many street properties, or alternatively the street would become a separate

\(^1\)This could be any normal form above 1st normal form.
Figure 1: An instance of an ER data model. This instance models customers, accounts, branches, and the 'customer at branch' relationship.
Figure 2: The hierarchical form of the ER model in figure 1

entity, which relates to the customer.

2.2 The hierarchical data model

The hierarchical data model is the oldest of the three major data models. The intention of the model is to capture one-to-one and one-to-many relationships. Many-to-many relationships can be represented only with additional constructs. The hierarchical model is usually represented in a schema containing tree structures. The trees of the forest will have different heights; trees with single-level height are equivalent to regular entity relations.

All hierarchical implementations possess some mechanism for escaping the limitations placed on relationships that can be represented by the hierarchical model. Namely, one common approach is to change the constraint of having trees to a constraint of having DAGs (Directed Acyclic Graphs). Using this extension redundancy can be avoided, and the expressive power of the model is strengthened.

One of the limitations of the model is that similarly to the previous models, it holds almost no semantic information. Another limitation is that navigation in this type of data model must always start from the root of a hierarchy. An example of an hierarchical data model can be seen in figure 2. In this figure the navigation limitation is apparent as well. In order to access a specific customer, one must start from the branch to which the user belongs.
2.3 The network data model

The network data model can be considered as an ER model with only binary, many-to-one relationships. The network data model introduces two basic types of data: records, and links, or sets. A link between records can be binary only. However, containment and ownership relationships can be represented on sets. i.e. a set is used to capture the relationship between an owner and one or more members. The network data model introduces semantic limitations. It is difficult to understand many to many relationships in a straightforward way. Also, the model introduces limited data independence. A change of a relationships from 1:N to N:M implies different records and links, which will imply different navigation in the schema. Finally, the network model is considered to have potential difficulty in navigation through the database.

2.4 Limitations of the traditional data models

The three conventional data models fail to represent many aspects of the structure and meaning of data. Basically, theses data models assume that all information can be arranged in two-dimensional tables. Each row of the table is a record having the same layout as other rows of the table. Major limitations of the three conventional data models include [Thompson et al. 1989]:

1. The models are fine as long as the data fits into a certain very limited pattern. Records should fit in predefined tables/files.

2. Generally, either there are too many ways of doing things or there is no way at all.

3. Horizontal homogeneity is assumed - there is some construct to entities implying the same attributes to all entity types.

**Example 1** For example, suppose clothing should be described in a database. There are many attributes that are not relevant to all types of clothing or have different meaning depending on context. An example for such an attribute is size: could be
waist size, neck size sleeve length etc. Techniques to overcome these problems include defining a record format to include a union of all the possible attributes, or to allow the same attribute to have different meanings in different records. Both approaches have attendant drawbacks.

4. Vertical homogeneity is assumed - all attributes are to be interpreted the same way. For example, suppose assignments of cars to individuals should be modeled in a database. There are many types of individuals who may be assigned a car: managers, departments, employees etc. Solutions to overcome this problem can be:

- The car record would have a type field indicating the type of assignee
- The record could contain a different field for each assignee type.

Both solutions are unnatural and present difficulties for uniform access of different types. Anyone interested in the assigned-to relationship probably wants to be able to consider the assignees as a single collection of objects. The record model is inherently unable to do this, or at the very least makes it difficult and non-extensive. If it becomes necessary to assign cars to locations as well, the whole structure will probably have to be revamped and all the programs that access it will probably have to be re-coded.

5. Naming- Referring to things is generally done through naming. Symbolic identifiers are generally unique within entity types. As a result, it is necessary to know both field and type if the data is to mean anything. Semantics of data are known only according to the entity type an object belongs to. There is no way to resolve the semantics of data without prior knowledge of the naming convention and typing used in the database.

[Parsons and Wand, 2000] identified that many of the difficulties arising when using traditional data models can be resolved by having a data model in which instances may exist independently from classification. In their article, the problems arising in database operation due to the need to assign every instance to at least one class are classified to design problems
and operation problems.

Design problems are of the following:

1. The Multiple Classification Problem - Usually there is no support for multiple classification unless the classes are within the same classification hierarchy, and imply specialization/generalization of one another. However often an entity would naturally be part of a few classes which have no relationship between them. For example, an entity may refer to an individual which is a student and a customer at the same time.

2. The View Integration Problem - The process of reconciling the different views of different users is called view integration or schema integration. This process involves combining the distinct views of multiple users into a single global schema, a new set of classes which cover the domain. This process is very difficult "since the same portion of reality is usually modeled in different ways in each schema" [Batini, 1992].

3. The Schema Evolution Problem - Adding classes or changing class definitions may necessitate different operations to complete the change. This may include moving instances between classes handling outdated information and more. Often this operations are much to complex to enable natural evolution of the schema.

4. The Interoperability Problem - This problem is similar in nature to the view integration problem. The interoperability problem refers to the ability to exchange information among independent databases. In order to achieve interoperability, the meaning of the classes and the relationships between them need to be resolved.

Operation problems are the following:

1. Handling Exceptional Instances - When all instances must belong to classes, once an instance has some additional unique properties a special class needs to be defined. This may lead to a proliferation of classes, which complicates the database schema. If the exceptional attributes are not identified in database design, schema modification
is needed in order to introduce new subclasses, which was identified as a complex operation above (Design problems 3).

2. Adding and Removing Instances - If instances have to be added or removed from the system, and if they belong to many classes this would have to be explicitly specified by the user. Consistency may be lost and many operations are laid on the user.

3. Removing a Class - In case a class is no longer relevant and it is decided to remove the class definition from the schema, all instances that belong to the class may be lost, and all properties that belong only to this class would be lost. It would have been desirable that the information that may be useful in the future would not be lost even though the class is no longer part of the schema.

4. Redefining a Class - Again in redefinition of a class, instances and/or properties may be lost. In case instances do not comply with the new definition of the class, they would be removed, and therefore lost. In case properties are no longer part of the definition of the class they would be lost.

2.5 The Two-Tiered Approach

The approach guiding our data model is the two-tiered approach proposed in Parsons and Wand [Parsons and Wand, 2000]. In the previous section we have listed some of the problems which are mainly a result of inherent classification in data models. In their work, Parsons and Wand examine classification in information modeling by referring to the fields of ontology and classification theory. Both ontology and classification theory recognize that instances exist independently from any classes. People first recognize the things in the world, and only then they from classes to organize their knowledge about the properties of individual things. Thus, the two tiered approach aims at separating between the knowledge of things and their classification. The major principles and corollaries influencing the two tiered approach, and thus also our data model are the following [Parsons and Wand, 2000, pp 239]:

**Claim 1** The world is viewed as made of things that possess properties.
Claim 2  *Classes are abstractions created by humans in order to describe useful similarities among things.*

Corollary 1  *Recognizing the existence of things should precede classifying them*

Corollary 2  *There is no single correct set of classes to model a given domain of instances and properties. The particular choice of classes (a view) depends on the application.*

Two important ontological constructs which are supported in our data model are the following:

- **Classes**: A class is a set of things possessing a finite set of common properties.

- **Precedence**: Precedence is a relationship between properties. The preceding properties of a property P are all properties that must be possessed by any instance possessing P. Another way in which precedence can be defined—Let P and Q be two properties. Q will be said to precede P if and only if the set of things possessing P is a subset of the set of things possessing Q.

The two constructs above are supported in our data model in a way that matches their ontological definition. In other words, entities belong to a class only if they possess all the properties defining the class. Entities can possess a property only if they possess all the properties which precede it.

2.6 Related Research - Different Approaches

In this section we provide a background of other data models which take a different point of view than the traditional data models. The objective of each data model is to provide a better way to reflect semantics in the data model. In our data model we try to incorporate some of the benefits which are apparent in the models described in this section, while adding new semantic information which is not provided in any of them—property precedence information. We further support multiple classification schemes by deferent users, and support the two-tiered approach.
2.6.1 Semantic data models

The desire to have better data models to support database design has led to the development of semantic data models. Semantic data models aim to provide increased expressiveness to the modeler and incorporate a richer set of semantics into the database [Peckham et al. 1988]. While early database research concentrated on the physical structure of databases and little consideration was given to the user's perception of the data, modern database research, including semantic data models, is more concerned with issues such as understandability and flexibility. Generally, all semantic data models attempt to provide more semantic information than the relational model provides. Semantic data models attempt to extend the semantics, and consistency rules that follow from them. One example for the employment of semantics is by generalization and aggregation, which are provided by virtually all semantic data models. The idea of generalization and aggregation was first introduced by [Smith and Smith, 1977]. An aggregation is abstraction which turns a relationship between objects into an aggregate object. For example, memory, CPU and motherboard aggregate to a computer. Generalization defines a subtype/supertype relationship. For example, a dog is a generalization of Puddle, Labrador etc. As the class of semantic data models has grown, benefits related to them became apparent. This benefits include [Peckham et al. 1988]:

- Economy of expression: Not only is the user able to extract exactly the same information available in traditional data models, but much of this information can be extracted with greater ease.

- Integrity Maintenance: Semantic data models provide mechanisms for the definition of integrity constraints and at the same time allow the user to view the data on a level removed from the low-level record structure.

- Modeling Flexibility: Semantic data models permit the user to model and view the data on many levels, rather than in only one fixed way.

- Modeling Efficiency: Semantic data models provide built-in elementary operations
and constraints. This saves the user from implementing the operation and checking constraints every time a relationship is defined or used.

Some of the existing semantic data models are T AXIS and SDM. T AXIS [Borgida et al. 1984; Mylopoulos et al 1980; Nixon et al.1987; O'Brien 1983], which is in fact a language that places emphasis on classification and generalization/specialization abstraction hierarchies. SDM [Hammer et al, 1981] incorporates a wide range of modeling constructs into a single abstraction called class. SDM provides the user with the capability to define classification, aggregation, generalization, and association. The emphasis is however given on classification and association. The E-R [Chen, 1976]model is also sometimes viewed as a semantic data model as it incorporates some semantic information- entities, relationships and most importantly a multiplicity of constraints. The model supports one-to-one, one-to-many, and many-to-many constraints on relationships, as well as insertion and deletion constraints can be defined using existence dependencies. For example, constraints such as 'workers can’t exist if their company does not exist' can be defined. However, all of the above semantic data models place high emphasis on the role of classes and their definition. This definition requires reconciliation on the view of the world by all users, since eventually the database would consist on a set of classes which are used by all users. It also limits the flexibility and changeability of the view of the world. In our data model we aim to overcome those limitations. If some user changes its conception regarding what is the definition of some specific class, all users must agree to this new definition and must be aware of the change in the definition of that class.

2.6.2 Object oriented data models

One class of semantic data models are the object oriented data models. Essentially these models provide the user with augmented and more natural possibilities for expressing and modelling logical aspects. In this section we will outline some fundamental language elements that are basic to object oriented data modeling [Vossen, 1991, Vossen, pp.200-207].
1. Object - which is what traditionally referred to as entity. What is essential is that a set of objects can be directly represented and manipulated without forcing the user to think in terms of records. Objects are composed of attributes and methods.

2. Encapsulation is supported. i.e. attributes and method implementation is hidden from other objects.

3. Complex objects can be derived from atomic ones through aggregation and grouping.

4. Attributes are described through mappings from one object type to another. That is an attribute has an object type as its domain.
   - Aggregation composes a new object type from already defined types by forming a Cartesian product.
   - Grouping denotes collecting of elements of an already existing type to form a set.

5. Similar object are grouped in a class that contains the data.

6. Classes are organized in a class hierarchy
   - Supertypes and subtypes are naturally represented through IS-A relationships. A supertype inherits all its attributes to each of its subtypes. There is a clear distinction between an IS-A relationship expressing a 'specialization' and one expressing a 'generalization'. Specialization is used to establish possible roles such as a student can also be a TA. Generalization is used to describe a situation in which a set of distinct objects fall into a new type. in generalization a supertype is 'covered' by its subtypes. For example Vehicle is a generalization of car, bus, train etc.
   - Each object of a class inherits all properties of its superclasses in the class hierarchy.

Advantages of the object-oriented data models can be summarized as follows [Vossen, 1991]:

15
1. OODM are object-oriented as opposed to record-oriented and hence allow a user to proceed in a more 'direct' fashion when defining data than is made possible by the constraint to translate observations from the real world into a record-formatted representation.

2. OODM typically provide a much larger repertoire for expressing relationships between data objects or object types than does a record model. Thus OODM, as all semantic models, served to bridge the gap between the semantics of an application as it arises in reality and the way these semantics can be represented in a database.

2.6.3 Deductive databases

The objectives of deductive databases are to provide logic-based extensions of database languages. These extensions would support more powerful queries and reasoning. Deductive databases extend the relational databases technology based on ideas developed in logic programming. Semantics of queries is based on the notion of logical consequence developed in mathematical logic. In fact, the entire logic approach is based on the formalism of the application world using mathematical logic. Deductive databases are first-order logic databases consisting of sentences of a special form and are based on deduction. In other words, the facts about different entities are deducted according to new statements provided by the users. Clauses in deductive databases are divided into two parts: facts, called extensional information, and rules, called intensional information. The intensional database relations are defined in terms of other relations. Thus, relations between different views of the world and different classification schemes can be defined. In this manner, deductive databases provide solutions to problems which our model addresses. However, deductive databases do not provide means of updating data in multiple views concurrently. They do not provide means for propagating updates across different views.
2.6.4 Semi-structured data models

In Semi-structured data models, information about the schema is contained within the data. Semi-structured data has recently emerged as an important topic of study for a number of reasons [Buneman, 1997]:

- There are data sources such as the Web, which we would like to treat as databases but which cannot be constrained by a schema.

- It may be desirable to have an extremely flexible format for data exchange between disparate databases.

- Even when dealing with structured data, it may be helpful to view it as semi-structured for the purposes of browsing.

In a sense, in semi-structured models, different objects may have different attributes and components even if they belong to the same class. This notion provides much flexibility in organizing data and enables different users to instantiate objects from same class in different ways. The prevalence of XML is very much an outcome of this flexibility. While our data model defines more structure than would be defined in semi-structured data models, our model provides more flexibility in structure than would be allowed in traditional models. This flexibility is assured by keeping semantics and relations between attributes in the database.

In our data model we keep precedence information which is hierarchical in nature. Since the structure of semi-structured data in general, and XML in particular, is hierarchical as well, implementation of XML databases is related to our work. [Shanmugasundaram et al, 1999] examine an approach of using traditional relational database engines for processing XML documents. [Tian] describe other approaches for storing XML documents. The query languages and their different features can provide ideas for the type of queries that may be relevant for our model as well. For example, XPath and its query abilities may be adopted to our data model in future research. The primary purpose of XPath is to address parts of an XML document. In addition to its use for addressing, XPath is also designed so that it has
a natural subset that can be used for matching (testing whether or not a node matches a pattern); XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

2.6.5 Other research directions

The problems identified in section 2.4 have brought about many other data models which improve the flexibility and evolvability of both traditional data models and other data models which were presented in this section. Some of these models have to do with enabling objects, that belong to different classes, to take different roles. [Pernici, 1990] Introduces a model which supports roles, and [Gottlob et al, 1996] demonstrate an implementation of the approach in Smalltalk. However, roles are associated with a specific class, and are not independent from classification. Moreover, a class takes one particular role in any given time. [Odberg, 1994] introduces how Schema Modification Management (SMM) can enable changes in classifications over time. In this work, schema versioning allows multiple versions of the schema to co-exist. This work however, ties entities with classes and does not enable the user to define relationships between properties as the two-tiered approach, which derives our model, does. Another approach to augmenting data with semantics is that of making the information explicit using Meta data. Naming conventions, repositories with descriptions of data are some of the ways semantics are made explicit. A description of some of the case tools which help with specifying and extracting Meta data is available in [Heiler et al, 1996]. Other approaches to augment classification problems were also proposed. [Bertino et al, 1995] introduced a model in which objects may be members of multiple classes, and [Ketabchi et al, 1985] formalizes the concept of category, and generalization of categories. While all of these models provide more flexible categorization and classification, they do not provide better modelling of properties and their meaning, as will be demonstrated in our model. The models also introduce limitations such as prohibiting the ability to have multiple categories/roles with overlapping properties. For example, if an object which belongs to class student can have roles of TA and roles of undergraduate. A property
of GPA would generally not be present in both, and won't be updated for both roles.

3 Towards a new semantic data model

3.1 Motivation and Approach

The approach guiding this model is to construct a data model which better represents knowledge about application domain and which has physical layer support. We will show how using research associated with conceptual modeling can assist in modeling databases. All the limitations exhibited in section 2.4 are resolved when using this data model. Having a data model in which entities possess properties gives much flexibility to the schema. Information about classes can change arbitrarily while all information about properties stays in tact. For example, consider we have an entity which has among other properties a student number property. Even if the class student is removed from the database the student number property would stay as it is. As will later be shown, also modifications to instances are very simple in the new data model. The user does not need to know all classes and properties which exist in the schema. A single modification would automatically propagated as needed. For example, if an entity ceases to posses the property 'works for someone' it will automatically cease to posses properties which are preceded by this property. Properties such as 'gets salary from someone'. Another advantage is that no longer do users have to be aware of all properties used by all applications. For example, consider an application which is concerned with processing the salary of all entities which are drivers. Another application is interested in all entities which have a salary, whether they are drivers or not. The most straight-forward way to implement such a database, would be to have one entity type of employees, which has a binary attribute 'is driver'. Users of application B, for which the attribute is meaningless, would have to be aware of its existence, every time they add or modify an entity which is part of this entity type. In our data model only the users of application A would be aware of the property 'is a driver for salary', while users of application B would be aware of the attribute 'has salary' only. Our model supports different views of the world by different users. We
apply ontological principles and refer to classification theory to enable such different views. In ontology, a class is defined according to a set of properties which all members of the class possess. In classification theory we learn that all three significant models of classification [Smith et al. 1981] share the idea that a class identifies a measure of similarity among its instances. One of the major benefits of classification is that it provides cognitive economy. By identifying classes, some knowledge can be associated with the class rather than repeated for all instances. In order to support cognitive economy and to provide multiple views of the world, we support classes in two different ways:

1. One way in which a class can be defined is merely as storing abstract knowledge in the class. Take for example the class student. By knowing that someone is a student we know that they learn something, they are part of an academic institution, they have a certain life-style etc. All this knowledge is part of the class and no instance needs to possess all these properties in particular. Thus, the premise is that when defining a class as a means to merely store abstract information, there is an agreement between all users about the abstract information the class entails. Note that there is no need to store this information in the database. However, the meaning of a class is the same in the view of all users in this case. Since the agreement is only on abstract information rather than specific information\(^2\) (such as takes at least 4 courses, has a student number, registered on a particular date, is part of some specific program etc.), we find this premise to be reasonable.

2. Another type of class is one which is defined according to a set of properties which all members of the class possess. This class is defined according to specific properties\(^3\) which need to be possessed. This type of class definition may differ from one user to another. For example a student may be defined by one user as one which has an enrolment date, has a name, and has accumulated credits. Another user may define a

\(^2\)In other words, value attributes
\(^3\)In other words, value attributes
student as one which had taken a course in some university and which is part of some program in that university.

By having this freedom in definition of classes we allow different users to define what properties they find to be essential in order for an entity to be part of some class. We also provide much knowledge about different entities without storing any property they possess, but only specifying that they belong to some class. Generally, an entity would be part of the class of the first type in one of two cases:

- some user, which did not define the class specifically, explicitly updates the database with the fact that the entity is part of the general class.
- Due to its properties, the entity was found to be part of the class in the second sense, for some user.

In order for an entity to be part of the second class for some user, it should posses all of the properties which define that class by that user.

Note that a class may be defined both globally (the first type) and both by a specific user (the second type). In this case, members of the class for each user would only be those that are both part of the general class and part of the class defined specifically by that user. Each time an entity is found to be a member of the class by a specific user it is also a member of the general class.

To summarize these examples, extensibility and independence of data would be better supported since the representation of data would be closer to our conceptualization of the phenomena it represents. Databases will be tuned according to new observations, in the same manner users adjust their observations and classifications of objects in everyday life. Moreover, the new data model helps map between the different views different users may have on the application domain. In the new data model we enable the construction of databases with extensive semantic definition of properties. By having new means for reflecting semantic relationships between properties observed by different users, many new features come about.
Consistency constraints are derived in a natural way. Each group of users is exposed to properties which are observed by them alone. Moreover, Better reflection of the data semantics will derive better evolvability and interoperability. Objects may change context, and may be classified differently by different users, whereas the database can stay backward compatible to older applications.

The advantages of our model are derived from two fundamental characteristics:

- Classification constraints on objects are lifted. Objects are classified according to properties they posses, and not the other way around. This enables extensive querying capabilities and natural expansion of the schema. This characteristic of the data model specifically enables databases to be forward and backward compatible. For example, in a program it may be assumed that an object might possess some attribute in future schema. However, in the schema relevant at the time the program is coded the attribute in question is not included (for various practical reasons). The application can check if the object possesses the attribute, and continue its flow according to that. Objects also stay backward compatible, no matter how the objects are categorized or what additional properties they acquire, their original attributes can still be accessed.

- Our model extends the ability to define semantics of properties through the definition of precedence relationships between properties - This enables both backward compatibility and natural means to derive consistency rules. Most importantly, our data model enables different users to use different attributes to observe similar properties. Each user knows only about the share of attributes which were found do be necessary for him/her. According to Bunge, precedence is the most general relation between properties. It should be mentioned that our implementation of the model does not support all precedence information. We support only a specific type of precedence which can be incorporated into a data model without having much burden implications on the user. If we were to support all precedence information, the user would have needed to provide extra information each time a property changed values, specifically how the
change in values affect preceded values. Our model supports all property which have values, although other properties can be easily be incorporated. In other words, we support the type of properties which appear in most applications: properties which change value over time. An example for a property we support is 'takes X amount of credits' where X may change over time. We do not support properties such as 'can run' since these type of properties do not change value over time, and are not popular in practical systems.

3.2 Overview of the data model

As people change perspective and gain knowledge, classification of objects and attributes of interest may change. In reality change of perception and knowledge is easily done. Our goal to enable the same flexibility in constructing the database brought us to examine ontological foundations and related research. According to BWW ontology, properties of a thing exist independently from humans awareness of them. Humans conceive of things in terms of models of things. Those models are conceptual things. Properties of conceptual things are termed attributes. Since different users may use different attributes to observe the same property, consistency issues will most likely arise once attempting to provide a data model which is flexible enough to provide each user with its own set of attributes. We propose that many of the consistency issues can be dealt with by one fundamental relationship between properties- property precedence.

Our data model supports storing objects and their properties independently from any classification. It supports viewing these properties through different attributes by different users, while keeping the database consistent to all users. Once a property is lost, all properties which are preceded by it are lost as well. Once a property changes value, all properties which are preceded by this value change as well. Nevertheless, our data model still supports classes to provide the same type of functionality as classic data models do. Generally, in our data model we define a class by a set of properties and one non-value property\(^4\). For a specific

\(^4\)The property of belongin to the class. This property implicitly states the possession of many non-value
user, an object belongs to a class if and only if it possesses all the properties defining the class\(^5\), or if the user did not define the class using properties and the non-value attribute is possessed by the object. In addition, our model supports property-precedence. We use property precedence to define the set of attributes an object must possess in order for it to possess a new property\(^6\).

In the data model, a unique identifier is associated to each object in the database. Each time a user finds it necessary to add an attribute to the set of attributes an object possesses, the object is updated to possess all the attributes preceding the new attribute. If the updated set of attributes possessed by the object, completes a set of attributes defining a class, the object becomes part of this class. Each time a user finds it necessary to remove some attribute from the set of attributes the object possesses, all attributes which are preceded by that attribute are no longer possessed by the object. Finally, Each time a value of an attribute is updated for an object, its effect is progressed to all of the attribute’s preceded and preceding attributes possessed by that object.

### 3.3 General Example

In this section we provide a general example, which exhibits some of the capabilities of the data model. Throughout the rest of this paper we limit property-precedence support only to special types of properties in which the strength of using property-precedence is most apparent, and in which using precedence is most efficient\(^7\). We model precedence only through sets of properties which are related to each other through what we call super-value-manifestation sets. In order to explain super-value-manifestation sets we refer to some definitions made in [Parsons and Wand, 2002].

**Definition 1** A manifestation of a property \(GP\) in an object \(X\), is a property \(P\), possessed attributes

\(^5\)According to that user

\(^6\)Note that from our definition of classes by users and property precedence it can be inferred that a class implies property precedence. In other words, defining a class by a set of attributes implies that the set of attributes defining the class precede the property of belonging to the class.

\(^7\)Note, however, that the model can be expended to support all properties as can be seen in future research section

24
by X and preceded by GP.

**Definition 2** A *value manifestation* is a manifestation of a property GP by P, where P provides additional value data not available in GP.

**Example 2** An example of value manifestation is having GP set to 'works from some date' and P set to 'works from January 8th'.

**Definition 3** *Super-value-manifestation* means that there is another generic property GP₂, which in case possessed by the object X, the value manifestation of GP₂ will have the same values as the value manifestation of GP (which is P) plus additional values.

**Example 3** An example for super-value-manifestation in this context could be having GP₂ set to 'Works from some date for some salary', and having 'Works from January 8th for a salary of 10K' as the value manifestation of GP₂. In this case, GP and GP₂ relate to each other through super-value-manifestation. Clearly, super-value-manifestation implies property precedence, i.e. GP₁ precedes GP₂. This type of property precedence is modeled in our data model.

Note however, that this data model can support any type of precedence information, not necessarily super-value-manifestation. However, in such cases the user would need to provide more input with every update of properties. Namely, the values of the different properties which precede the property.

### 3.3.1 Example with diagram

In this section we will present the basic constructs of the data model and present the benefits of using these constructs. We will use as an example an instance of our model, illustrated in figure 3 and 4.

In the data model properties and classes are defined. Classes are notated using boxes with margins at the top and bottom. Properties are notated with simple boxes. An arrow from a class to a set of properties indicates that these properties define the class (and precede
the class). An arrow from one property to another indicates that the source is preceded by
the destination in a super-value-manifestation relation. In other words, updating some of the
values in some property for an object, results in an update to some descendant properties\(^8\)
of that property. Similarly, some of the values of the ancestor properties\(^9\) are updated. Also,
loosing some property implies that all the ancestor properties are lost. Similarly, gaining a
property implies that all the descendant properties are gained.

A relational property is a property which is shared by a few objects. For example, X
works for Y is a property shared by X and Y. A relational property implicitly imposes
special integrity constraints. Namely, weak properties come about. That is, an object may
lose an intrinsic property once a relationship property was lost.

**Definition 4** A weak property is a property which may exist only in case a property preceded
by it ceases to exist.

**Example 4** In the example given in figure 4, if a course is updated with the fact that it was
not taken by anyone, it loses the property 'course taken by someone', the relational property
'student X takes course W in year Y' is lost. Now, since 'Student takes some course in year
Y' is a weak property, it is lost as well.

This issue brings about another integrity constraint: suppose a student loses the property
'Student takes some course in Year Y'. In this case, the relational property 'student X takes
course W in year Y' would be lost as well. However, the course object may continue to
posses the property 'Course taken by someone'. Specifically, if there is another instance of
the relational property 'student X takes course W in year Y' with the same course object but
a different student/year, the course continues to posses that property. For this reason the
diagram annotates the cardinality of relationships. In the case of the relation just analyzed,
'student X takes course W at year Y', only one 'course taken by someone' precedes the
relational property. However, many relational properties of type 'student X takes course

---

\(^8\)In other words, preceeding properties
\(^9\)in other words preceeded properties
W at year Y' are preceded by the property 'course taken by someone'. If at least one of those relational properties exists, the course object still possesses 'course taken by someone' property. Thus we have 1:N annotated on the arrow. When an arrow is not annotated the default cardinality of 1:1 is applied.

In this instance it is apparent how our data model supports different views of the modeled domain by different users. The major difficulty in supporting multiple views lies in the need to agree on classes. Agreeing on classes amounts to agreeing on the exact set of properties the concept represents. This difficulty is overcome by enabling the users to agree on the general meaning of each concept, while enabling each user to augment the general meaning of each concept with properties they find necessary. In figure 3, the conceptualization each user has for the set of classes in the system is annotated with a separate box. As was stated in section 3.1, for each class in the system a general attribute is_classname is defined. This attribute is possessed by any object which is associated with the class without stating exact properties of the class. These properties are properties which are identical to all members of the class and do not take specific value.

Example 5  The property 'studies at some institute' is a property which is possessed by all members of class student and which does not take any specific value. Thus this property is associated with possessing the property 'is_student'.

Example 6  The property 'takes course X' is a property which while may be possessed by all members of the class, can take different values in different instances. For example, student A may possess the property 'takes course COM 635' and student B can possess the property 'takes course COM 633'. This property can not be associated with possessing the property 'is_student'.

In the example, the database supports class 'course' and class 'student'. The property of possessing the general property of belonging to these classes are 'is_course' and 'is_student' respectively. To show the flexibility of classes using this formation, consider the following:
In the example, user1 updates object objA as being part of the mutual property 'student takes course in some year'. One of the preceding properties of this property is that ObjA is a student. That is, objA now possess the property 'is_student'. Since user1 defined the class student as defined by (i.e. preceded) 'has student_id' and 'takes some course' properties, these properties are automatically possessed by objA as well. Similarly, the object which participated in this mutual property as a course would possess the property 'is_course' and 'has_credits'. Since user3 defined class student by 'registered_in_some_program' and 'takes_some_course', only if objA was previously updated as possessing the property 'registered_in_some_course' will objA be considered a student by this user. In the view of this user, The property 'student takes some course in year Y' will not be possessed by objA either. However, the properties 'is_course' and 'course taken by someone' would be possessed by the object taking part in the newly defined instance of the attribute 'student takes course in some year'.

Another advantage of this model is that it supports modeling different parts of the domain by different users. It also supports the view of each user and keeping the database consistent to all users. Consider the following domains of the three users of this instance: One user is processing information about students. Another user is using information about courses and uses the 'course taken by someone' property. The requirement imposed by this user is that instructors must have taken the course they are teaching prior to instructing it. Note however, that it is not required that an instructor has been a student. A third user is keeping general information regarding instructors. For this example, lets examine how the data regarding the specific course, which the instructor took and is now teaching, is updated. Data used by all users are updated for all. For example, if the number of credits of the course is updated by one user, all user views will be updated. If the third user updates the fact that the person (who is the instructor) did not take the course, the information will be automatically propagated so that the person did not get any grade in the course, and thus the person does not possess the property of teaching the course. Data needed by only one user do not influence the others. For example, if the first user changes the year in which the
Person named X teaches course Y in which he got W grade

Registered in some program

Figure 3: An instance of the data model in which different users define the classes 'course' and 'student'
Figure 4: An instance of the data model in which different users look at different properties. All properties relate through precedence information.
course was taken, the second user is not influenced. Further, in this instance it is apparent that users may query about information which crosses class boundaries. For example, it is possible to ask about all courses taken by someone, whether it is a student or an instructor or some visitor, through the 'course taken by someone' property. It is also possible that if only part of the courses have grades, to know about all the grades registered in the system. This is done through 'Person X has grade Z in course Num Y'.

4 The proposed data model

4.1 The meaning of properties and precedence

As the model is based on BWW ontological foundations, objects, or entities, are not instances of specific classes, but rather are things that possess different properties. We define a new construct, which applies to a general attribute, a property-view. A property-view is a means for representing an attribute statement. A property view represents attributes in general form, and may have instances which represent specific attributes, i.e. specific statements. This issue will be further elaborated in this section.

The new data model is based on this new construct, which will be notated as PV or property-view interchangeably throughout the rest of this work. In the model, property-views have two major roles: defining a general attribute, and providing means for manipulating all information associated with that attribute. These roles can be fulfilled only if the model will keep information indicating which properties are affected as a result of changing a single property. In other words, the model holds precedence information for each attribute. Property-views keep both the specific value of a property and its precedence information. This information can be kept by exploiting the structure of general attributes in predicate form.

Definition 5 Predicate form of general attributes has the following structure [Wand et al. 1999, Wand, pp 499]:

31
\[ A : T_1 \times \ldots \times T_n \times V_1 \times \ldots \times V_m \rightarrow \text{Statement regarding } A. \quad (1) \]

Where \( T_i \) are sets of possible things and \( V_i \) are sets of possible values. In many cases having a similar general attribute, by omitting some of the sets which take part in the definition of the property, will yield a preceding property. For instance,

\[ A' : T_1 \times \ldots \times T_{n-1} \times V_1 \times \ldots \times V_m \rightarrow \text{Statement regarding } A'. \quad (2) \]

An example, for such precedence could be 'driver \( T_1 \) of a bus \( T_2 \) for company \( T_3 \)' which may apply to a statement in expression 1, in the following manner

\[ \text{driver}_{-} \text{of\_bus\_for\_company} : T_1 \times T_2 \times T_3 \quad (3) \]

and 'driver \( T_1 \) of bus \( T_2 \)' which applies to a statement in expression 2. In the following manner:

\[ \text{driver}_{-} \text{of\_bus} : T_1 \times T_2 \quad (4) \]

In such cases we can define the general attribute in equation 1, using equation 2, and an additional set \( T_n \). In our example we can have:

\[ \text{driver}_{-} \text{of\_bus\_for\_company} : \text{driver}_{-} \text{of\_bus} \times T_3 \quad (5) \]

By defining the general property 'driver_{-}of\_bus\_for\_company' in such a way, we have both defined precedence and enabled access to all the relevant information regarding any statement this general property is instantiated to. Another implication of this construct is that once we change information associated with property (statement) 4 we change the information associated with property (statement) 5, since the value of \( T_1 \) and \( T_2 \) in equation 3 is accessed using the property 4.

**Example 7** Take for example the statement \( A: \text{driver}_{-}t_1 \text{ of bus } t_2 \), which applies to expression 4. Consider the case that this statement applies to a driver of some company. In this

\[ t_1 \in T_1, t_2 \in T_2 \]

[32]
case, the statement $A'$, applying to expression 5, would use statement $A$ with some $t_3 \in T_3$.  

$A': A \times t_3$.

Consider the case that the driver is assigned another bus in the company. I.e. we change $t_2 \in T_2$ in this equation to hold $t_2'$, statement $A'$ would automatically be updated.

We can now go further and define the property 4 to the following two properties

$$\text{Is}_\text{a}_\text{driver}_\text{of}_\text{some}_\text{bus} : T_1$$

$$\text{Is}_\text{a}_\text{bus}_\text{driven}_\text{by}_\text{some}_\text{driver} : T_2$$

We now expect any instance of the of $\text{Is}_\text{a}_\text{driver}_\text{of}_\text{some}_\text{bus}$ to be an object which is a member of group $T_1$. Similarly, we would expect any instance of the of $\text{Is}_\text{a}_\text{bus}_\text{driven}_\text{by}_\text{some}_\text{driver}$ to be an object which is a member of group $T_2$. The natural way to define membership in groups is through classification. However, our model's advantages are based on the fact that classification is not assumed. One way to define groups, is by using the attributes 'is_classname' to indicate if some object is a member of the group. Since different users can overload the meaning of belonging to a class, this way implies that different objects belong to groups $T_1$ and $T_2$ in the views of different users. An alternative way would be to define groups in terms of properties they posses. Using this approach we can give immediate access to the properties, rather than search for the relevant properties in the another object. This is the approach taken in our definition in property-views. Let us assume that possessing the property of being a driver is preceded by having a license and having a name. thus we would like to define $\text{Is}_\text{a}_\text{driver}$ in the following way:

$$\text{Is}_\text{a}_\text{driver} : \text{license} \times \text{name}$$

$\text{license} : V_1$

$\text{name} : V_2$

where $V_1$ is set of values for licence number and $V_2$ is the set of legal names

Let us assume that possessing the property of being a bus is preceded by having a license plate and having some number of seats. Thus we would like to define 'Is_a_bus' in the
following way:

\[ Is\_a\_bus: license\_plate \times has\_seat\_number \]

\[ license\_plate: V_3 \]

\[ has\_seat\_number: V_4 \]

where \( V_3 \) is the set of values for licence plate number

and \( V_4 \) is the set of legal number of seats

It would thus be tempting to omit \( T_i \)’s in our model, and have properties represented only in terms of \( V_i \)’s. However, there is a need to distinguish which attributes belong to which object. Obviously, the driver does not have seats, and the bus does not have a name. To resolve this issue, we associate an object identifier to each specific attribute given in a property view. This object identifier gives indication which object is the subject of the property. For instances of mutual properties, which have implications for several objects we associate a group identifier as well. In the example, the statements which apply to the general property 'driver _of _bus' ( instances of expression 4 ) will hold a group identifier and the object identifier of objects possessing the different preceding properties: The object id of the driver, \( t_1 \in T_1 \) and the object id of the bus, \( t_2 \in T_2 \). For this reason \( v_1 \in V_1 \) and \( v_2 \in V_2 \) would be associated with the object \( t_1 \). Whereas \( v_3 \in V_3 \) and \( v_4 \in V_4 \) would be associated with a different object id, \( t_2 \in T_2 \). Finally, we note here that license_plate, has_seat_number, license_number, and name are attributes at the lowest level. They can not be further broken down. They apply to an intrinsic property of a single object. In our model, these general attributes are distinguished in that their instances are the only ones which values are actually manipulated in case any of the property’s preceding properties is manipulated. These general attributes are represented and manipulated through special property-views, named low-level property-views\(^{11}\), which would be further described in the

\(^{11}\)Note that low-level property-views may apply to values which have no meaning without a higher level property view. For example, the property 'works from date' will imply a low level property-view 'from date'. This property view has meaning only in the context of 'works from date', and thus will be accessible only using it.
As mentioned, PVs are closely related to the general attributes. Property-views provide access to attributes representing different properties. For this reason we will often distinguish between different property-views according to the set of general attributes they relate to. Users can access those attributes and manipulate them through property-views. A specific object can be accessed through a property view if and only if the object possesses properties which correspond to the PV. In this way, if an object does not possess a property, there is no way an attribute corresponding to that property will be investigated. This is an important characteristic of the model which is consistent with the ontological framework in which not having a property is not a property.

Since classification is defined in terms of sets of attributes, and since different users may use different property-views to manipulate an object, objects may be viewed as instances of different classes by different users. This is another important property of the model, consistent with the ontological foundations.

### 4.2 Formalization

We now formalize the model through a series of definitions and notations, which will be used throughout this paper. We start by defining three different levels of property, and thus three different levels of property-views:

- properties which apply to a single intrinsic attribute of an object;
- properties which apply to a set of intrinsic attributes of an object; and
- properties which apply to a relational attribute.

This distinction of three types of property views will be used in other definitions and explanations given in this work.

**Definition 6** Let $A$ be a general attribute given in a predicate from, which involves a set of things and a set of values.
A \colon T_1 \times V_1 \rightarrow \text{Statement regarding } A.

The low-level property-view of } A \text{ is composed of an identifier, the set of things } T_1 \text{, and a set of values } V_1. \langle A, T_1, V_1 \rangle

\textbf{Example 8} The property has\_name: AllObjects \times Strings, has a corresponding low-level property-view. \langle \text{has\_name, AllObjects, Strings} \rangle

\textbf{Definition 7} Let } A \text{ be a general attribute given in predicate form, which involves one set of things.

\[ A : T_1 \times V_1 \rightarrow \text{Statement regarding } A. \]

The \textit{Middle-level property-view} of } A \text{ is composed of an identifier, the set of things } T_1\text{, and sets of values } V_i. \text{ An ordering is imposed only between identical sets of } V_i\text{s.}

\textbf{Example 9} The property has\_phone\_number\_at\_office\_number\_since\_date:
All\_Buildings \times office\_numbers \times Phone\_numbers \times All\_dates,
has a corresponding middle-level property-view.

Note that there is a one to one mapping from middle level property views to attributes in general form in case it is an intrinsic property.

Finally, we define property-views, to apply in a way that applies to all attributes in general form.

\textbf{Definition 8} Let } A \text{ be a general attribute given in predicate form.

\[ A : T_1 \times \ldots \times T_n \times V_1 \times \ldots \times V_m \rightarrow \text{Statement regarding } A. \]

The \textit{High-level property view} of } A \text{ is composed of an identifier, sets of things } T_1\text{, and sets of values } V_i. \text{ An ordering is imposed only between identical sets.}
Example 10  The property

Works_for_company_since_date: All_Persons × All_Companies × All_Dates

has a corresponding high-level property view.

Proposition 11  Have the set of things, Ti, in the definition of a high-level property-view be defined in terms of the properties which the set of thing must posses.

Claim 3  Every set of things can be defined in terms of an attribute it possesses

Proof.  Assume there is no attribute that defines a set of things Ti in the general attribute A. consider the property 'Is applicable to be part of the predicate form of the general attribute A in position i'. This attribute defines the set of things Ti.

Claim 4  Using proposition 1, middle level property views can be defined in terms of low-level property-views. High level property views can be defined in terms of middle and low level property views.

Middle level property-views can be defined in terms of low-level property views which satisfy the following:

- Ti in all low level property views is identical to Ti in the middle level property view
- There is a one to one mapping from each Vi in the middle level property view to a low-level property view with the same Vi.

Example 12  Consider the low-level property view which corresponds to the property

has_name: All_People × strings, and has_birth_date: All_People × All_dates.

The middle-level property view which corresponds to

Was_Born_at_Date_and_named : All_people × All_dates × All_names

can be defined in terms of the two low level property views:

Was_Born_at_Date_and_named: has_name × has_birth_date

37
High level property views can be defined in terms of low-level, and middle-level property views, by having property-views that satisfy the following:

- For each $T_i$ there is a middle level property-view defining the set $T_i$.

- There is a mapping from each $V_i$ in the high-level property view to another $V_j$ in a middle or low level property view where $V_i = V_j$. There is also a mapping, possibly many to one, from every $V_i$ in the low and middle level property-views to a $V_j$ in the general form of the high-level property-view.

**Example 13** Consider the middle level property views corresponding to:

- $is\_course\_given\_at\_semester\_for\_X\_credits : All\_courses \times \{fall \mid winter\} \times \text{integers}$, and $has\_completed\_some\_course\_with\_grade : all\_students \times \{A-D\}$.

The high-level property view corresponding to $got\_grade\_Y\_at\_course\_at\_semester\_for\_X\_credits: All\_courses \times \{fall \mid winter\} \times \text{integers} \times all\_students \times \{A-D\}$,

can be defined in terms of the above middle level property-views.

From the claims above we see that property-views, which are not low-level, can be defined in terms of other property-views. We now expend our definition of high-level property-views, so they can be defined recursively, using other high-level property views as well. This new definition enables us to incorporate property precedence into the model.

**Claim 5** A property view can be one of three types of property-views.

- a low-level property-view.

- a set of low-level property-views, and an ordering on low-level property-views with multiple occurrences- named middle-level property-view.

- a property view defined in terms of other property views: named higher-level property-view.
**Definition 9** The set of PVs defining a specific property view, \( X \), will be called the *defining PVs* with regard to \( X \).

**Example 14** In example ?? The defining PVs with regard to the PV \( \text{got\_grade\_Y\_at\_course\_at\_semester\_for\_X\_credits} \) are the PVs corresponding to \( \text{is\_course\_given\_at\_semester\_for\_X\_credits} \), and \( \text{has\_completed\_some\_course\_with\_grade} \).

Using the above definition of different PVs, we can notate the different PVs in the following way:

**Notation 1** A property view can be expressed in terms of defining PVs, an ordering of repeated defining PVs, and a unique identifier in the following manner:

\[
\text{UID}: \{PV_1, PV_2, \ldots, PV_n, PV_a \times \ldots \times PV_m\} | T_i \in PVs.
\]

Which can be eventually be broken up to low-level property views only

\[
\text{UID}: \{LLT_1, \ldots, LLT_w\} | LL \in \text{Low - LevelPVs}.
\]

**Definition 10** The set of low-level defining PVs, are said to *explicitly define* a component of the general attribute, represented by the defined property view. Specifically, they represent a \( V_i \) Component.

**Definition 11** Also, the set of middle-level defining PVs, which are used to represent \( T_i \) are said to *explicitly define* a component of the general attribute, represented by the defined property view. Specifically they represent a \( T_i \) Component.

**Definition 12** All components of the general attribute which are not explicitly defined are said to be *implicitly defined* by the middle or high-level defining property-views.

Up to now, we have formalized the notion of a property-view, and how it relates to general attributes. We have also shown how property-views can be defined. Now we examine how we populate property-views with instances. An instance of a property view represents a specific property of the general property represented by the PV.
Example 15 Consider the property-view which corresponds to the general attribute -
works_from_date_to_date: All_Objects × date × date.

An instance of this property view will map to a specific attribute that relates to the above attribute -
works_from_date_to_date: Joe × August_11_2000 × August_11_2001

Since property views can be defined using other property views, an instance of a property-view can avoid keeping the specific values of the statement it represents, but rather specific instances which it relates to.

Example 16 In the above example, consider that there is also a property view which corre­
sponds to the general attribute-
works_from_date: All_Objects × date.
This PV may have an instance that relates to the specific attribute -
works_from_date : Joe × August_17_2000
If, for example, the identifier of this instance is UID_1, the instance of the general property
works_from_date_to_date given in the previous example, would keep the following: UID1 × August_17_2001

An instance of a property view represents a specific attribute of some object. An instance of a PV with respect to some object O assigns each PV_i a value pv_i ∈ { set of instances of PV_i}. Properties may be relational or intrinsic. In case a property is relational it is not possessed by any object, rather it is possessed by all participating objects. However it is always preceded by properties possessed by each object which is part of the relational property. Namely, the property stating they are part of that relational property.

Example 17 The property view which corresponds to the general attribute-
works_for_company: All_people × All_companies, is preceded by the following two general attributes-
works_for_some_company : All_people.
has_some_employee : All_companies.

Every instance of a PV would be associated with some object using an object identifier.
Each instance would be associated with instances of its defining property views, as stated
above. For an instance of a property-view, with respect to some object O, the following
would apply to the instances of its defining property views:

- For intrinsic properties, any pv_i which is an instance of a low-level property-view,
contains the value v_i of some defining V_i of the general property represented by the
property view.

- Any pv_j which is not an instance of a low-level property-view, and which is associated
with an object^12 which is equivalent to the object O, defines a property preceding the
property represented by the property view^13.

- Finally, any pv_j for which the object identifier is not equivalent to the object identifier
of O, will hold an identifier of a relational property preceding the defined property.
This relational property will implicitly define part of the T_i,s of the general attribute
represented by the property-view^14.

In short, any defining high or middle-level property view, which has instances with the
same object id imply property-precedence of the property represented by the instance of
the property-view^15. On the other hand, any high or middle-level property view in the
definition, which have instances with a different object id, define part of the general form of
the attribute they represents. Finally, any low-level property view would have instances with

^12i.e. an object identifier attached to the pv
^13Note that different components of the general form of the attribute may be encapsulated in the preceeding
property.
^14It will probably implicitly define part of the V_i,s of the general attribute represented by the property
view as well
^15Note that different V_is and T_is of the general form of the attribute may be encapsulated in the preceeding
property.
the same object id as the object the intrinsic property applies to, and defines a component of the general form of the attribute it represents.

We now add two notations which would simplify our explanations in section 6

**Notation 2** Let HPV be a high-level property-view representing a general attribute GA. The components of GA defined in HPV using low-level property views are called non-preceded components. The rest are called preceded components.

**Example 18** Consider the property-view representing the general attribute:

\[ \text{is\_named\_and\_works\_for\_company}: \text{All\_Objects} \times \text{All\_Companies} \times \text{strings}. \]

Consider defining this property view using the property views representing the general attributes

\[ \text{has\_name}: \text{All\_Objects} \times \text{strings} \]

\[ \text{works\_for\_company}: \text{All\_Objects} \times \text{All\_Companies} \]

i.e. the property view \( \text{is\_named\_and\_works\_for\_company} \) will be represented as follows:

\[ \text{is\_named\_and\_works\_for\_company}: \text{has\_name} \times \text{works\_for\_company}. \]

Since \( \text{has\_name} \) is a low level property-view, \( \text{strings} \) is a non-preceded component of \( \text{is\_named\_and\_works\_for\_company} \). On the other hand, since \( \text{works\_for\_company} \) is not a low-level property-view, \( \text{All\_Objects} \times \text{All\_Companies} \) are preceded components of \( \text{is\_named\_and\_works\_for\_company} \).

**Notation 3** The path from property-view \( PV_i \) to another property-view \( PV_n \) is an ordered list of property views \( \{PV_1, \ldots, PV_n\} \) where each property view is part of the defining PVs of its predecessor. In case some \( PV_i \) has multiple occurrences of \( PV_{i+1}, PV_{i+1} \) will hold an index indicating its order in the definition of \( PV_i \).

If there exist a path between two property-views

\[ \{PV_1, \ldots, PV_n\} \]

\[ \text{It may be argued that All\_Object is defined both explicitly and implicitly since it is both part of the low-level property view and the high level property view. With out lose of generality, in such cases we call it an implicitly defined component.} \]

42
, PVₙ is accessible from any property view on the path.

\[ PV_i \quad 1 \leq i \leq n \]

**Example 19** Consider the property-view which applies to the general property `has_name`: all_objects × strings. Consider the property view which applies to the general property `was_given_a_name_on_date`: all_objects × strings × dates. The later property view can be defined using defining property views as follows: `was_given_a_name_on_date`: `has_name` × dates. In this case there is a path from PV `was_given_a_name_on_date` to the PV `has_name`.

We summarize this section by stressing out important points regarding the meaning of property-views in our model, and the means for defining them.

Property-views are unique to our model and are different than classes or roles. Indeed, classification can be defined in terms of sets of properties their members posses. However, classes, by definition, impose that an object has some classification. Property views imply which properties are possessed by an object, provide means to express the way properties are interrelated, and capture their semantics in terms of precedence. An object may belong to many property views concurrently, each involving different properties. Moreover, in our model we state that classes are defined in terms of sets of properties. Any object possessing this set of properties belongs to a class. On the other hand, property views are defined differently. Any object possessing a property must possess all its preceding properties. These preceding properties are captured by the definition of the property-view. Property views are also different from what the literature defines as roles in that property views do not define a context of interpretation of an object. A property view can relate to only one property. It is hard to think of a role, or a class which relate to one property only. In short, the following properties of PVs distinguish them from being roles or classes:

- They have no classification meaning.
- An object may have many different property views. Some may even overlap.
• They impose no conceptual role.
• They merely define a set of general attributes an object possesses.
• They capture semantics of property by property-precedence.

Property views have a different meaning according to the set of PVs defining them:

• Low-level property views- consist of one attribute. They represent an intrinsic property.
• Middle-level property views- consist of a set of attributes and/or an attribute, which in general form involves a few values. This property view represent a set of intrinsic properties.
• High-level property views- consist of a set of attributes. They represent a collection of intrinsic, and/or mutual, and/or relational properties.

Apparently, if a high-level property view does not represent any mutual/relational property, it could have been equally defined using low-level PVs only, and thus be a middle-level property-view. However, in our model we make an important distinction between high-level PVs and middle-level PVs. A high-level property-view should be constructed if and only if it possesses all attributes represented by its defining PVs. On the other hand a middle-level property view is constructed if and only if it does not precede any property represented by some high/middle level property-view.

5 Benefits of the model

General Benefits

In the model, users can access the general attribute a PV relates to and manipulate it through the PV. A specific object can be accessed through a property view if and only if the object possesses properties which correspond to it. In this way, if an object does not posses a property, there is no way an attribute corresponding to that property will be investigated.
for that object. This is an important characteristic which is consistent with the ontological framework in which not having a property is not a property.

Since classification is defined in terms of sets of attributes, and since different users may use a different property-view to manipulate an object, objects may be viewed as instances of different classes by different users. This is another important property of the model consistent with ontological foundations. Objects in the model are globally and uniquely identified. Different users can access different attributes of objects from different property views. For example, a user who queries for objects having the properties of a 'technician', and is interested in the properties that are part of 'person', may access them through the 'technician' property view. Another user may query for objects having the properties of student, and may still access the same 'person' properties of student. Other users may want to access all objects holding attributes of 'person', no matter if those have attributes of students, technicians, or others. In this case, if an object applies to both the attributes defining a technician and the attributes defining a student, it would be retrieved only once.

Finally, this model solves many of the deficiencies of traditional models mentioned above (section 2.4):

1. The pattern of data is not limited. Data is broken according to sets of attributes by property views. The graining of the breakdown may change as the database evolves

2. There is only one way to model objects and relations: This is in terms of objects' attributes. Attributes may represent mutual properties (relations) or intrinsic properties. Attributes are accessed through property views.

3. There is no 'horizontal homogeneity'. For example, suppose clothing should be described in a database. There are many fields that are not relevant to all types of clothing or have different meaning depending on context. The data model could represent the property view of clothing as an intersection of 'color(v)', 'year(v)', 'made_in(t)' etc. Each specific clothing item will have a unique property view consisting of the intersection of clothing property view and a specific size property. For example skirt
property view = clothing AND waist_size(v). Property views for different clothing items can accept different domains for the property size.

4. Vertical homogeneity is not assumed; attributes may be interpreted differently. For example, the assignment of cars to individuals problem can be modeled in this data model. There are many types of owners who may be assigned a car: managers, departments, employees etc. Having a property view with respect to car-owner to each one of these objects will enable users to handle this group in a uniform way no matter if the objects are in fact different. If it becomes necessary to assign cars to locations as well, adding a car-owner property view to locations will enable the change.

**Evolution**

Changing objects in the proposed model is much easier than in record-based models. There is no need to lose data regarding the object as it evolves, since different views can coexist. If, for example, a student becomes a TA in some database, the view of the TA as a student may co-exist with the view of TA using different property-views. The properties that a particular user assumes are possessed by a student may be defined using a strict subset of the those possessed by a TA, completely distinct, or may partly overlap. Older applications, which were not updated regarding the possibility of TAs, can access the object in the same way they used to rather than using different naming schemes/table. Moreover, since the model uses property-views, in order to access and manipulate objects, users need not use a specific property-view UID to query an object. Rather they can use low-level property conjunctions to define the properties needed. For this reason, it is important that all users will have the same UIDs for all low-level property-views. ('color', 'weight', 'name' etc.). However, high-level property views may have different UIDs for different applications. For example, a specific application may define 'Works from date for salary in address' to consist of low-level property-views UID 1,2 and 4. The database, on the other hand, may define it to consist of different low-level property-views, say UID 6,7,8, and 'does X hours of work per day' to consist of 1,2 and 4. By comparing the property-views of the two
definitions, the DBMS can retrieve the latter whenever the user asks for the former. It should be mentioned that this kind of usage assumes that the combination of the defining PVs of the a property-view implies its meaning. If for example, the user’s property-view of 'has_weight_and_height' is defined using 'weight' and 'height' alone, the database will retrieve all objects that possess these attributes. This may include the PVs corresponding to 'has_maximum_height_and_weight', 'each_number_of_centimeters_weighs' etc. For this reason, it is reasonable that all property-views will have unique identifiers at most levels. This emphasis the distinction between high-level property-views and low-level property-views. Many high-level property views can be defined using the same set of low-level PVs.

**Interoperability and Compatibility**

The use of property-views to access objects helps in achieving interoperability as well. The meaning of objects and data is understood through properties, rather than some class name. Accessing specific attributes of an object can be done in a straightforward way, without completely agreeing on the meaning or classification of an object. Finally, our data model supports backward compatibility using the evolution relation. Moreover, forward compatibility is more easily achieved. Querying the database regarding available information for a specific object is easily incorporated into the model. In order to achieve forward compatibility applications need to foresee possible future changes in the database, and query whether an object is accessible through a certain property-view. For example, an application may be written assuming that students may be augmented with a 'start_date' attribute, at a future time. However, when the application was written no such property was used to define the student class. In order to achieve forward compatibility the application can query whether the student class includes 'start_date' property in its definition, and continue the flow accordingly.
6 Implementation Issues

6.1 DBMS architecture using the data model

After explaining the conceptual ideas and how they were derived, it is useful to see how an implementation of such a data model would look like. In this section we will outline how the data model can be incorporated in a database management system in which the logical model is the relational data model. This section would demonstrate the applicability of our data model. It will also help understand the meaning of property-views and the distinction between classes and precedence information. This section also helps understand what kind of information is needed from the user and what needs to be stored in Metadata constructs in order to support such a data model. Finally, in this section we see how ontological ideas fit with implementation needs. For example, without careful examination of ontological constructs, it might seem a precedence relation graph might contain loops. i.e., A precedes B and B precedes A. However, in our implementation, it is sufficient to have only one precede the other and all data would be consistent in both. In a matter of fact, if a user defines 'loops' of precedence, the database would be flawed and inconsistent. Precedence information reflects the users' semantics. Therefore, it is up to the user to avoid providing precedence information with 'loops'.

A DBMS should provide interface for communication between the three database levels: the internal, logical and external levels. One way in which an existing DBMS can incorporate the new data model is by providing some interface from the new data model to the existing data model. By using this kind of approach there is no need to define an interface between the logical and the internal level. This kind of approach can be used if all additional Meta-data constructs needed in the new data model can be constructed using logical data constructs given in the existing data model. As will shortly be shown, the new data model can be utilized by simply adding a few internal relations. This enables us to incorporate this data model in a DBMS with a relational logical model, and to provide an interface between the new data model and the relational model. We choose to outline the implementation of our
data model in a relational model due to its simplicity, its acceptance, and its practical appeal. Other methods to implement our data model may include using an OODB, and a deductive data model.

The fundamental construct in the new model is the property view. Thus, new Meta-data constructs should include relations holding information about property-views. The information needed includes the definition of each property view and which instances of each property-view exist in the system. When considering methods to store instance information, two fundamentally different approaches may be considered. The first one is efficient in space consumption, but is expensive in transaction time. In this approach, each instance of a property view will have one entry with pointers to instances of its defining property-views. Values of instances of each property-view can be retrieved only by navigating through all the its preceding property-views.

Example 20 Consider the property view corresponding to 'works for company for salary' which is defined using the PVs 'works for company' and 'works for salary'. Consider a user who queries the value of the salary with respect to some object. This value would be retrieved by finding the instance of 'works for salary' which is used to instantiate 'works for company for salary', and then find the value of salary in that PV.

The second approach is more efficient in transaction time but is more expensive in space consumption. This approach will be demonstrated throughout this section. In this approach, the number of entries for each instance of a property view will be equal to the number of low-level property-views to which the property-view can eventually be broken down to. Values of instances of each property view can be retrieved without visiting any of the property-view's preceding property-views. We chose to use this latter approach, and thus have a model which is efficient in time. However, our model can be mapped to another model which applies to the former approach in a relatively straight-forward manner.

6.2 New Meta-data constructs
6.2.1 Overview
The suggested implementation involves the addition of new relations that will be part of the schema or will include the data. In this section we will give an overview of those new relations. In the next section we will give a more thorough explanation regarding each relation. Following that, in section 6.2.3 we will provide a full example with the population of tuples in the relations.

As a naming convention relation would be named \(r_{relation-name}\). When relation-name ends with (PV), the specific suffix of the relation name is determined according to the PV unique identifier.

The schema relations are:

1. \(r_{HLDefine}\) : This relation will be used to keep information about the definition of all higher-level PVs. The definition of higher-level property-views in terms of lower level property views would be kept here.

2. \(r_{root}\) : This relation is used for practical reasons only. It helps resolve how new relations of property-views, which define classes, can be populated according to instance in other relations.

3. \(r_{isClass}\) : A binary relation which identifies whether a property view also defines a class or only a property. If it defines a class, once all the defining properties of the property view are possessed, the object becomes a member of the class the PV defines.

4. \(r_{evolution}\) : This relation will be used to keep information about the change of definition of PVs for the benefits of interoperability and compatibility with different application versions.

5. \(r_{NameUID}\) : This relation will keep the mapping between names and UIDs as seen by different users. Since users may have different interpretation for attributes as representing different properties, this relation is needed. Specifically the definition of classes is different for different users.

The data relations are:
1. \( r_{HLUID(PV)} \): This relation will be used to keep values of instances of other property views which provide means to access the value of an instance of the PV this relation is associated with.

2. \( r_{LLUID(PV)} \): This relation will be used to keep values of instances of lower level property views.

### 6.2.2 The constructs

For every defined low-level property view, PV, a new relation \( r_{LLUID(PV)} \) over the set of attributes

\[
X = \{Key, Value, ObjectUID\}
\]

is used in the DBMS. The attributes of each tuple in the relation correspond to an identifier of the instance of the low-level property-view for an object, the value of the property, and the object or group\(^\text{17}\) to which the specific tuple in the relation applies respectively.

**Example 21** Consider property-view with UID 3 is associated with the general property 'has_weight'. Consider object with UID 0\(_1\) possesses this property, and weighs 4kg. In such a case we would the tuple \((1, 4, O_1)\)\(^\text{18}\) in the relation \( r_{LLUID(3)} \).

With the creation of a new higher-level property-view PV, two relations, \( r_{HLDefine} \) and \( r_{isClass} \) will be updated. Also, a new relation \( r_{HLUID(PV)} \) will be defined. \( r_{HLDefine} \) will be

\(^{17}\)ObjectUID is an identifier of an object in the case of an interistic property. It is a group identifier in the case it has a meaning in the context of a relational property only. For example, if we have the property 'works for company from date'. The low level property view holding the value of date will hold a group identifier of the instance of 'works for company from date'. Note that the value of date has meaning only in the context of the relational property it is part of.

\(^{18}\)the value of Key=1 in this relation is arbitrary. Key is used to reference this tuple from other relations.
over the set of attributes

\[ X = \{ \text{Key}, \]
\[ \text{DefinedPVUID}, \]
\[ \text{DefiningPVUID}, \]
\[ \text{Type}, \]
\[ \text{Represents}, \]
\[ \text{CardinalityFrom}, \]
\[ \text{CardinalityTo} \} \]

This relation will be used to keep information about the definition of a higher-level property view. Namely, every tuple in this relation will indicate that property view DefinedPVUID is defined, among other property-views, by DefiningPVUID. This defining property-view may represent either a \( T_i \) from the attribute general form, or some \( V_i/\)precedence. This information is available through the value of Represents in the tuple. 'V' indicates that DefinedPVUID represents a \( V_i/\)precedence. 'T' indicates that DefinedPVUID represents a \( T_i \). Finally, CardinalityFrom defines cardinality constraints as described in the introduction section. This information is summarized in Table 1.

\( r_{isClass} \) would be a binary relation in which each property view will be marked as defining both property precedence and classification information, or property-precedence only. This relation will be over the following set of attributes:

\[ X = \{ \text{PVUID}, \text{IsClass} \} \]

The attribute IsClass would have the value 'true' if and only if the property-view defines both a class and property-precedence. The attributes and their meaning in this relation are summarized in table 2.
### Table 1: Attributes and meanings in $r_{\text{HLPVDefine}}$, a Meta-data relation keeping data of definition of property views

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>A Unique identifier within this relation to enable reference the tuple.</td>
</tr>
<tr>
<td>DefinedPVUID</td>
<td>A Unique identifier of a PV which is defined using DefiningPVUID</td>
</tr>
<tr>
<td>DefiningPVUID</td>
<td>A Unique identifier of a PV which defines DefinedPVUID</td>
</tr>
<tr>
<td>Type</td>
<td>Whether DefiningPVUID has multiple occurrences in DefinedPVUID. I.e., it is a list or a set within DefinedPVUID</td>
</tr>
<tr>
<td>Represents</td>
<td>DefiningPVUID represents a Vi or a Ti from the general attribute associated with DefinedPVUID</td>
</tr>
<tr>
<td>CardinalityFrom</td>
<td>Whether loosing the property associated with DefinedPVUID will imply the loss of the property associated with the instance of DefiningPVUID.</td>
</tr>
</tbody>
</table>

### Table 2: Attributes and meaning of $r_{\text{IsClass}}$, a Meta data relation keeping information of which property view defines a class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVUID</td>
<td>A unique identifier of the property-view being referred to</td>
</tr>
<tr>
<td>IsClass</td>
<td>A boolean value indicating whether the property-view defines a class or not.</td>
</tr>
</tbody>
</table>

Table 1: Attributes and meanings in $r_{\text{HLPVDefine}}$, a Meta-data relation keeping data of definition of property views

Table 2: Attributes and meaning of $r_{\text{IsClass}}$, a Meta data relation keeping information of which property view defines a class
The new relation $r_{HLUID(PV)}$ will be over the set of attributes

$$X = \{ \text{Key}, \quad \text{LowerLevelPVUid}, \quad \text{LowerLevelPVKey}, \quad \text{LowestLPVUID}, \quad \text{LowestLPVKey}, \quad \text{ObjectUID}, \quad \text{Order} \}$$

The first 6 attributes of each tuple in the relation correspond to an identifier of the higher-level property-view for an object, a unique identifier for another property-view which is part of the definition of the property-view, a key for that other property-view which is part of the definition of the property-view, a lowest level property-view which will be eventually accessed through the lower-level PV, the key to that low-level PV, and the object (or group, in case of a relational attribute) identifier to which the specific tuple in the relation applies respectively. The last attribute, Order, is explained in the next paragraph. This information is summarized in table 3.

In addition, a new relation named root over the same set of attributes will be created. The objects with property-views at the highest level, i.e. the property-views which are not part of the definition of any property view, will have corresponding tuples in this relation. Note that in most cases $r_{HLDef(PV)}$ could be extracted from $r_{HLUID(PV)}$. However, $r_{root}$ is needed in cases $r_{HLUID(PV)}$ is empty, and when $r_{HLDef(PV)}$ includes defining PVs of type 'T'. This latter case will be elaborated in section 6.3.3.

In the relational schema, $r_{HLUID(PV)}$ and $r_{LLUID(PV)}$, the attribute Key is used since each object might have multiple occurrences of the same attribute. The Type and Order attributes in the higher-level property-view relational schema are needed in order to handle

\[\text{Thus the combination of 'ObjectUID' and 'LowerLevelPVUID' are not sufficient to define a key}\]
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>A Unique identifier within this relation to enable reference the tuple.</td>
</tr>
<tr>
<td>LowerLevelPVUID</td>
<td>A Unique identifier of a PV which is part of the defining PVs of this PV.</td>
</tr>
<tr>
<td>LowerLevelPVKey</td>
<td>A key of a tuple within the relation which holds tuples associated with LowerLevelPVUID.</td>
</tr>
<tr>
<td>LowestLevelFVUID</td>
<td>A unique identifier of the PV of a tuple within a relation which represents a low-level PV, which LowerLevelPVUID eventually breaks down to.</td>
</tr>
<tr>
<td>LowestLevelPVKey</td>
<td>A key of a tuple within a relation which represents the low-level PV, LowestLevelPVUID, that holds the value relevant for this instance of the PV</td>
</tr>
<tr>
<td>ObjectUID</td>
<td>A unique identifier of the object this instance of the property view is associated with.</td>
</tr>
<tr>
<td>Order</td>
<td>In case LowerLevelPVKey occurs multiple times in the definition of this property view. Order will resolve the ambiguity regarding the meaning of each of those property-views.</td>
</tr>
</tbody>
</table>

Table 3: Attributes and meaning of $r_{HLUID(PV)}$, a relation holding instances of a higher-level property view

cases in which the same property view occurs a few times in the definition of the same property view.

**Example 22** Consider the property-view 'Has phone number' which may occur multiple times in the context of 'has_personal_info' property view. In case the number of T_i's stays constant we can simply use the Order attribute in order to differ between the exact meaning of each property-view in the context of the defined property view.

However, sometimes the number of occurrences of a specific property view is dynamic.

**Example 23** For example, if we have a property-view of account-info, it may be represented as-

account-info: Id X Account-Balance.

Where having a general-property of Account-Balance is meant to represent account-balance as balance information in different dates. Property-view Account-Balance may be represented as-

55
Account-Balance : Account-State \times \ldots \times \text{Account-State.}

However, we do not know a-priori how many Account-State properties an individual object may have.

In the above example, in order to support any number of Account-State information, we would need an infinite number of different property-views, each defined to hold a specific number of Account-State information. While according to our ontological foundations acquiring or losing a property should change the classification of an object, having a different amount of recorded history is not equivalent to acquiring or losing properties. Generally, the properties of an object change according to its current state rather than its history. For these reasons, it is important to have a special representation operator that will have a domain of a set of specific properties; each is a specific property of the same property-view type. We define such a representation operator as follows:

\[ X \in \text{SetOf}(T) \Leftrightarrow X \in \text{p}(T). \]

In the same manner, it is natural to have a representation operator of ListOf(X), for cases in which there is a meaning to the ordering of general properties of the same kind.

In the example, it is more natural to represent Year-Account balance as follows:

Account-Balance: SetOf(AccountState). For this reason relations of type, \( r_{\text{HLDefine}}(PV) \) have the property type, which can be either 'List', 'Set', or 'Individual'. Higher-level property-views also have the attribute Order to support property-views of type 'List'. In case the database need not support property-views of type list the attribute 'Order' can be omitted. These new representation operators can be applied to the model using the same definitions and assumptions made thus far. However, we redefine property-path in the following way.

**Definition 13** The path from property-view \( PV_i \) to another property-view \( PV_n \) is an ordered list of property views \( \{ PV_1, \ldots, PV_n \} \) where each property view is part of the defining PVs of its predecessor. In case some \( PV_i \) has multiple occurrences of \( PV_{i+1} \), or \( PV_i \) is a property-view of type list, \( PV_{i+1} \) will hold an index indicating its order in the definition in \( PV_i \).
Finally, we have two additional relations \( r_{\text{evolution}} \) and \( r_{\text{NameUid}} \). As the advantage of the proposed data model lies in its ability to support different views of the world by different users, the DBMS is naturally assumed to support multiple users. All UIDs have the same interpretation among all users. However, the mapping between names and UIDs can be different for different users. The meaning of low-level property views, the building blocks of property-views, is agreed upon between all users. As higher level property-views are being added to the system, different users may conceive of the attributes reflect different properties. Thus, users should be able to have their own naming schemes to property-views. While the unique identifier of a property-view is guaranteed to stay consistent, naming convention may vary as a function of time and user. \( r_{\text{NameUid}} \) will be over the set of attributes

\[
X = \{\text{User}, \text{Name}, \text{PropertyViewID}\}
\]

The attributes of each tuple in the relation correspond to an interpretation of a name by a user to a unique identifier of a property-view. \( r_{\text{evolution}} \) will be over the set of attributes

\[
X = \{\text{OldPVUid}, \text{NewPVUid}\}
\]

\( r_{\text{evolution}} \) will be used for compatibility and interoperability purposes as will be explained in the next section.

6.2.3 An example of an instance of the model with the corresponding relations

In the example of figure 5, the database has two low-level property view representing 'weight' with UID 2, and 'name' with UID 5. There is also one high-level property which its defining property views are 'weight' and 'name'. The UID of a higher-level property-view can be in an ideal setting be given according to the lower-level property views it is composed of. To illustrate this issue, the UID of tool will be 2x5. In the figure, Low-level property views of the object are at the bottom of the diagram. Higher-level property views are above the lower level property views and arrows define which property views are part of the higher-level property views.
Figure 5: An instance of the data model which supports the properties 'has_weight', 'has_height', and 'has_weight_at_height'

The DBMS should have a relation for the low-level property views, name and weight. Since the UID of the property-view of name and weight are 5 and 2 respectively, relation rLL2 and rLL5 are added. The DBMS should also hold a relation for the high-level property view tool. Thus, a relation supporting the tool property-view, rLL2x5 will be added to the DBMS. Consider the case where the database contains information about two tools. One has UID 114, is named hammer, and weighs 10lbs. The other has UID 115, is named screwdriver, and weighs 2lbs. The relations, will look as illustrated on the next table:

6.3 DDL for a DBMS using the data model

In this section we describe what information is needed from the user in order to create and maintain the database. This is done by providing a basic DDL which can be used in order to define the database schema. In constructing our DDL we distinguish between different types of property-view:

- Fully preceded property-views: property-views in which all the Vₖs and Tₖs of their
Table LL2

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>ObjectUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>115</td>
</tr>
</tbody>
</table>

Table LL5

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>ObjectUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'Hammer'</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>'Screwdriver'</td>
<td>115</td>
</tr>
</tbody>
</table>

Table HL2x5

<table>
<thead>
<tr>
<th>Key</th>
<th>Order</th>
<th>LowerLevelPVUID</th>
<th>LowerLevelPVKey</th>
<th>LowestLevelPVUID</th>
<th>LowestLevelPVKey</th>
<th>ObjectUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>114</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>115</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>115</td>
</tr>
</tbody>
</table>

Table HL2x5Define

<table>
<thead>
<tr>
<th>Key</th>
<th>DefiningPVUID</th>
<th>DefinedPVUID</th>
<th>Type</th>
<th>CardinalityFrom</th>
<th>CardinalityTo</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2x5</td>
<td>Individual</td>
<td>1</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2x5</td>
<td>Individual</td>
<td>1</td>
<td>1</td>
<td>V</td>
</tr>
</tbody>
</table>

Table Root

<table>
<thead>
<tr>
<th>Key</th>
<th>Order</th>
<th>LowerLevelPVUID</th>
<th>LowerLevelPVKey</th>
<th>LowestLevelPVUID</th>
<th>LowestLevelPVKey</th>
<th>ObjectUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2x5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>114</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2x5</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>114</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2x5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>115</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2x5</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>115</td>
</tr>
</tbody>
</table>

Table NameUid

<table>
<thead>
<tr>
<th>Name</th>
<th>UID</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>2x5</td>
<td>1</td>
</tr>
<tr>
<td>Weigt</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6: Relations populated with tuples in an instance of the data model
corresponding general property, can be reconciled from preceding properties.

- Partially preceded property-views: property-views in which some of the \( V_i \)'s and all of the \( T_i \)'s of their corresponding general property, can not be reconciled from preceding properties.

- Relational property views: property-views define a relational property. These property views explicitly define some of the \( T_i \)'s of their corresponding general property.

6.3.1 Defining fully preceded property-views

The syntax for defining high or middle level property-views may have the following construct:

\[
\text{Create individual PV < property – view name > by ( < existing property – view name 1 >, < existing property – view name 2 >) [is class]}
\]

This command will create a new individual property-view in terms of its defining property-views. According to the following rules:

- The mapping between the UID of the new property-view and its name is equivalent for all users.

- The meanings of the view-names, which define the new property view, are resolved according to the mapping between names and UIDs defined by the user.

- The property-view is also a class only if the is class flag is included in the command.

Note that executing an identical command of the above format by two different users may result in the creation of two different property views.
An alternative DDL command for creating a higher-level property-view is by defining
the property view in terms of the UIDs of the property-views which are part of the new
property-view.

Create individual PV < property — view name >
by ( < existing property — view UID 1>, < existing property — view UID 2>)
[, existing property — view UID 3,..., existing property — view UID n ]
[is class]

This command will create a new individual property-view in terms of its defining property-
views as well. However, since the UIDs have the same mappings to property views for all
users, such a command will have the same meaning, no matter which user executes it.

As mentioned in section 6.2, high-level property-views can be either of type Individual, set
or list. So far we have only described the syntax for creating new individual property-views.
In order to create list, or set property-views, statements of the following syntax should be
used:

Create {List | List} PV < property — view name > By < existing property — view name >

Or
Create {Set | List} PV < property — view name > By < existing property — view UID >

By creating these property views the cardinality_from and cardinality_to attributes in the
r_HLDefine(PV) are set to 'n'. Examining these statements reveals that a set or list type
property-view can be defined only with regard to a single property view. Moreover, we see
that it is possible to recursively define property-views as lists of sets, set of lists, sets of sets,
list of list etc.

A property view can be created empty or populated. As property-views define properties
rather than classes, the defining property views are not sufficient to determine which object
possess the defined property. In other words, if an object possesses all properties defining the property view, it does not necessarily possess the newly defined property. However, some property-views may be classes as well. If possessing the preceding properties is sufficient to derive that the property is possessed, the property-view defines both a class and properties precedence relations. In those cases once the new property-view is created, all objects which possess all the properties that correspond to the defining property-views, should be added to the relation that corresponds to the new property-view. This is done by setting the new relation according to the contents of the following relational queries:

\[ R_{\text{components}} = \bigcup_{R \in \text{HLPV}} (R \cap \text{add}) \]

\[ \rho_X (\text{LowerLevelPVKey}, \text{LowestLevelPVUID}, \text{LowestLevelPVKey}, \text{ObjectUID}, \text{LowerLevelPVKey}) \]

\[ \Pi(R) \times R_{\text{PVUID}} \]

\[ \text{Key, LowestLevelPVUID, LowestLevelPVKey, ObjectUID} \]

\[ R_{\text{HLPV (newUID)}} = R_{\text{components}} \times (\Pi_{\text{ObjectUID, LLPVUID}} R_{\text{components}} \div R_{\text{add}}) \]

Here \( R_{\text{add}} \) corresponds to a temporary relation having one attribute- \( \text{LowerLevelPVUID} \). A tuple is part of the relation if and only if the definition of the new property-view holds a name which corresponds to the tuple UID. \( R_{\text{PVUID}} \) is another temporary relation which holds one attribute- \( \text{LowerLevelPVUID} \). The relation holds one tuple, a tuple with the UID of the property view. First, the relation \( R_{\text{components}} \) is constructed. This relation is the union of all records in the property-views defining the new property-view. For each tuple in \( R_{\text{components}} \) the Key attribute is renamed to \( \text{LowerLevelPVKey} \), and the name of the property-view it was extracted from replaces the original value in \( \text{LowerLevelPVUID} \). In the second phase, only those objects which possess all defining properties are left. In case the above relation is not empty, the root relation will be added with tuples that correspond to the new tuples in the relation. All the property-views, which are part of the existing-property-view list in the creation command, are removed from the root relation, in case there are tuples corresponding to them there.
6.3.2 Defining partially preceded property-views

So far, we enabled creation of a property-views only in terms of existing property views, and thus have only enabled definition of properties in terms of aggregation of existing properties. However, recall that we can create new low level property-views to represent different $V_i$s from the general form of the property. We can also create new middle-level property-views to represent different $T_i$s from the general form of the property. We can then use these new low-level property-views to define a high-level property-view, which could not have been defined in terms of precedence with respect to existing property views only. This point raises two questions:

1. Should a property, which is partially preceded by other properties, be defined in a few steps:
   - One defining the new lower-level property views which represent $V_i$s/$T_i$s (which can not be represented using preceding property views).
   - The second, defining the new property view in terms of existing property-views only, some of which were just created.

2. If partially preceded property-views should be defined in one step, according to what criteria should the relation, representing a property-view which is also a class, be instantiated. Specifically, how do we instantiate the values of $V_i$s which are not implicitly defined.

To better explain those issues, consider, as an example, a database with property-views representing the following general properties:

\[
\text{Works\_for\_company}(T_1 \times T_2) \\
\text{Student\_at\_university}(T_1 \times T_2)
\]

represented by property view $PV_1$ and $PV_2$ respectively.
we want to create a new property view to represent the following general attribute

\[ student\_at\_university\_works\_for\_company\_from\_date(T_1 \times T_2 \times V_1) \quad (8) \]

Since this property is preceded by the properties represented by property-views PV_1 and PV_2, it will be defined in terms of PV_1 and PV_2. Note, however, that date is not available in any preceded property. Thus this property is only partially preceded: the value domain of 'from_date' is not preceded. Since this property is only partially preceded there is a need for a new low-level property-view to represent V_1 in the property's general form 8. Thus the new property view will be represented as

\[ student\_at\_university\_works\_for\_company\_from\_date(PV_1 \times PV_2 \times PV_3) \quad (9) \]

where PV_3 is a new property-view representing V_1.

The first question applies to whether we should define the property-view 9, in two steps: one creating PV_3, and the second creating property-view 9. If we create the property-view in two steps, property view 9 will be populated in a regular manner\(^{20}\). As a result, even if PV_3 defines both a property-view and a class, if PV_3 will not be instantiated with values between step 1 and step 2, no instances would be part of the new property-view. The relation associated with the new property-view will be empty. However, if property view 9 is to be defined in one step, the second question arises. We find that it is necessary to create a partially preceded property view in one step. The reason is that the non-preceded V_i has no meaning without the context of the new property-view. In other words, in this case 'from date' is not an independent property and thus should not be made accessible to the user unless accessed in the context of \( student\_at\_university\_works\_for\_company\_from\_date \) property. Whether or not the DBMS stores the information internally in a manner similar to the way accessible property-views are stored does not matter. The issue is that the user does not see 'from_date' as a property and thus should not declare it as a property view. This is an important

\(^{20}\)Note that it will be populated only if it is both a property-view and a class.
characteristic of our data model: it closely maps to properties an precedence information as viewed in ontology, and there is no spurious data despite decomposition. Decomposition is done only when it maps to properties.

To handle the issue of one step definition of partially preceded property-views we note the following:

Claim 6 Let $A$ be the class of objects defined by a partially preceded property-view $PVA$. Let $PVB$ be the set of property views which represent properties preceding $A$. Let $B$ be the class of objects which possesses all the properties represented by $PVB$. $A \subseteq B$.

Claim 7 When defining a class by a partially preceded property-view, the value of all implicitly defined-parameters of the general-attribute represented by the property view, is known for all instances of the class at initialization time.

Proof. Let $GenPV$ be the general form of the attribute represented by property-view $PVA$. Let $o \in A$ be an object which possesses the general attribute represented by $PVA$. Using claim 6, we know $o \in B$ holds. All components of $GenPV$ which are implicitly defined using some $pvb \in PVB$ have the same value as they have in the general form of the attribute represented by $pvb$, by the very definition of property-views. Given that each $B$ has instances in all members in $PVB$, All implicitly defined $V_i/T_i$ in $GenPV$ are known upon creation of $PVA$. ■

Thus, in case the partially-preceded property-view defines a class as well, it would be reasonable to have an instance of the property-view for all objects that have instances in all property-views representing properties preceding the property-view. However, since in claim 6 we note that $A \subseteq B$ the user should be able to exempt instances with respect to a set of objects. Using claim 7, the value of the preceded components is given in the instances of the object in preceding property-views. The value of the non preceded component should be given a default value. Therefore, the syntax for defining a partially-preceded property-view, which is also a class, would be of the following format:
Create individual class PV <property-view name>
by (<existing property-view name 1>, <existing property-view name 2>)
[,]<existing property-view name3,...,existing property-view name n>
new [<low-level property-view name 1>, type <type> = <valid value>]
[,]< new low-level property-view n>, type <type> = <valid value>]
[except <objectUID1>..., <objectUIDn>]

In case the property-view does not define a class, the following type of command would suffice:

Create individual PV <property-view name>
by (<existing property-view name 1>, <existing property-view name 2>)
[,]<existing property-view name3,...,existing property-view name n>
new [<low-level property-view name 1>, type <type>],
[,]< new low-level property-view n>, type <type>].

6.3.3 Defining a property view which represents a relational/mutual attribute

As was previously mentioned, a mutual/relational attribute is an attribute which includes at least two T_i group in its general form. Using property views, it may be that all T_i components are implicitly defined using preceding properties. In other words, it may be that the relational factors of the attribute are defined in preceding relational property-views. In such cases, the creation of a relational property view will be created using the DDL commands introduced so far. However, if some T_i should be explicitly defined in the property view, the property view should be created in a number of steps:

1. Create the different property-views which will define the group of objects that belong to the group of possible t_i ∈ T_i.
2. Create the new relational property view, using the property-views created in the first step. The user should explicitly indicate which of the defining property-views represent explicitly defined $T_i$'s\textsuperscript{21}, and which define explicitly defined $V_i$'s. Thus the DDL command should have the following syntax:

\[
\text{Create individual PV} \quad < \text{property-view name} > \\
\text{by} \ ( < \text{existing property-view name 1} >, \\
\text{existing property-view name 2} >) \\
[, \text{existing property-view name 3}, ..., \text{existing property-view name n}] \\
\text{new value}[ < \text{low-level property-view name 1}, \text{type } < \text{type} >], \\
[... < \text{new low-level property-view n}, \text{type } < \text{type} >]. \\
\text{[new relational } < \text{property-view name 1}>...,<\text{property-view name n}>]
\]

The relation $r_{\text{HLPV}}$ for this property-view will be created empty. Population of instances for this property view would be done using the following DML command:

\[
\text{Insert into MPV} \quad < \text{property-view name} > \\
\text{Relation } < \text{RelationUID}> \\
[\text{relational } < \text{property-view name 1}, \text{object UID 1} >,... \\
,<\text{property-view name n}, \text{object UID n}>]
\]

where $< \text{RelationUID}>$ is the UID of the relation. Similarly, the name of the relation can be used. The value of each $t_i$ for the created instance would be determined as follows:

- Tuples with values of instances of preceding property-views are populated into the relation representing NPV. These tuples give values to implicitly defined $T_i$'s. They

\textsuperscript{21}The tuples of these defining property-views in $r_{\text{HLPVDefine}(PV)}$ will have value 'T' for the Type attribute.
can be determined using the following relational expression.

\[ \bigcup \sigma_{\text{ObjectUID}=\text{ObjUID}}(R_{LLPV}(\text{id})) \]

Basically, the above expression extracts the information regarding which of the defining property-views define T_i using the \( r_{HLPV\text{Define}} \) relation. In all of those defining property-views, the tuples which correspond to the object which is populated into the new property view are linked from the new property-view.

- Tuples with values of instances of other property-views, which implicitly define T_i are populated into the relation representing NPV in the following manner:

Let TPV be a property view which explicitly defines some T_i, and let O(T_i) be the object unique identifier given by the user in the DML command to populate the property-view. Each of the set of tuples representing the instance of TPV will have the following values: LowerLevelPVUID = TPVUid, ObjectUID= O(T_i). LowerLevelKey will have the Key value of the specific tuple in TPV. LowestLevelPVKey , LowestLevelPVUID, and LowerLevelPVKey will have a different value in each of the set of tuples representing TPV. These values can be determined according to the following relation expression:

\[ \pi_{\text{LowerLevelPVUID}, \text{LowestLevelPVKey}}(\sigma_{\text{ObjectUID}=O(T_i), \text{Key}=\text{LowerLevelKey}}(R_{HLPV}(\text{TPV}))) \]

### 6.3.4 Defining low-level property views

Interestingly, thus far we have not specified the types of values the constructs of the general attribute can take. This is possible because types are defined in the low-level property views which explicitly or implicitly define a value-domain of the represented general attribute. For this reason, when defining low-level property-views, different information type is needed from the user. Every low-level property view must clearly hold a name and the domain of the attribute it represents. The syntax for defining a low-level property-view should resemble the following:
6.3.5 Populating property-views with instances

In section 6.3.3 we have described how property-views which represent relational attributes can be populated. In this section we describe the type of information required from the user in order to populate other types of property views. We also describe the implicit operations done in the DBMS as a result of populating property-views with new instances. These operations are done in order to maintain database consistency. Naturally, an object can possess a property only if it possesses all of the property's preceding properties. Therefore, once the user instantiates some property view, PVA, with respect to some object, the user implicitly declares the object to possess the property represented by PVA and all properties preceding it. Thus, all the defining property-views of PVA should have instances with respect to that object. Also, their defining property views should have instances with respect to that object, and so forth. For this reason, once the user populates a property-view with a new instance, the DBMS implicitly populates all of the property-views, which represent properties preceding the property represented by the populated property-view. This operation is done in a DBMS process which follows the following guidelines:

- Populating instances starting from the lowest level property views preceding the property-view.

- A property view preceding the subject property-view is populated once all its defining property-views were populated with new instances. In fact, all property-views which also define a class, and in which one of the defining property-views was just populated with an instance, has the potential to have just possessed the last property defining the property-view. Thus, every such property-view should be checked whether it should be populated with respect to that object.
• For this reason the set of all the property-views which were populated with respect to the object is maintained during the process.

• As a final step, this set is expanded using bottom-up fix point resolution in order to find the set of all new properties possessed by the object. In this step only property-views which define classes can be instantiated, since all other relevant properties were already instantiated. Instances for these properties are added to the relations applying to their property-views.

The proposed DDL command to support populating property views has the following syntax:

\[
\text{Add } < \text{objectUID} > \\
\text{into } < \text{property-view name} > [\text{at } < \text{order} >] \\
< \text{low-level PV name-list} >, < \text{low-level PV value-list} >
\]

Each item in \(\text{low-level PV name-list}\) is a low-level property view which is used either explicitly or implicitly to define the property view \(\text{property-view name}\). The assumption here is that no low level property-view in \(\text{low-level PV name-list}\) has multiple occurrences. All occurrences of the same low-level property-view name in \(\text{low-level PV name-list}\) should be distinguished by providing the path to the low-level property-view, or some portion of the paths which uniquely define each low-level property-view.

In case \(\text{property-view name}\) is of type list, \(\text{order}\) is used to define which item in the list is inserted (first, second etc.), otherwise \(\text{order}\) is not used.

### 6.3.6 Removing instances from property-views

The counter operation to populating property-views with instances, is the operation of removing instances from property views. The conceptual meaning of removing an instance from a property view is of updating the database with the fact that an object ceases to possess a property. Since the object no longer possesses a property, it can't possess any other
property preceded by that property. Thus, in order to maintain precedence consistency, all
instances of property-views which use the removed instance should be removed. This oper-
ation is done recursively each time a new instance of a property-view is removed in order to
maintain precedence consistency. Additionally, using the cardinality precedence information
available in $r_{HLDef ine}$ additional precedence consistency can be maintained, as described in
section 3.2

6.3.7 Schema modifications

In section 3.1 we have stated that classes are individually defined by different users. Being
part of the class is a property shared by all users$^{23}$, but does not entail possession of any
specific properties. However each user may consider the fact of being part of the class to
entail that some properties must be possessed by the class member. Since the property of
belonging to the class, which is shared by all users, is given independently from possessing
any specific property, each user can change its view regarding the additional properties
needed in order to belong to the class without effecting the view of any other user. In order
to determine whether an object possesses a property that is preceded by a property stating
class membership, the DBMS examines if the additional properties required by the user in
order to belong to the classes are possessed by the object. If they are not, the object does
not possess the property represented in the property view, otherwise it does. However, once
a property-view is created its UID never changes. Only if a property-view defines a class, it
may be modified without creating a new property-view. In case the property-view defines
some general property, a new property-view is created. The new property-view is supposed
to replace the general use of the old one, and may be assigned the name of the old property-
view. If the old property view is not supposed to be used at all, it may be removed. However,
in order to enable interoperability and backward compatibility, evolutionary information is
kept. Using the following syntax to modify a property-view’s definitions can provide this
information.

$^{23}$Namely the property is_classname

71
Modify \(<old-property-view-name>\) to \(<new-property-view-name>\) \(\{\text{Remove } <property-view-list>\}\) \(\{\text{Add } <property-view-list>\}\)

The implication of this command is that a new relation is created for the new higher-level property-view. The contents of the modified property view will be determined according to the following rules:

- The relation will be created by adding every object, which possesses all properties that the defining property-views represent. In case the add list is empty, the relation will include all tuples from the old-property-view relation, except those which have \(\text{LowerLevelPVUID}\) that is part of the remove list.

- The implicit transaction made at the logical level will correspond to a relational expression as follows:

  - Let \(R_{\text{remove}}\) be a temporary relation having one attribute- \(\text{LowerLevelPVUID}\). A tuple is part of the relation if and only if the remove list holds a name that corresponds to the tuple's attribute value.

  - Let \(R_{\text{add}}\) be a temporary relation having one attribute- \(\text{LowerLevelPVUID}\). A tuple is part of the relation if and only if the add list holds a name which corresponds to the tuple's attribute value.

  - Let \(R_{\text{oldPV}}\) be the relation, which corresponds to the property view as it is prior to the change being processed.

  - By removing tuples which have more than one duplicate in the following relational expression, the tuples to be populated in the new property view can be extracted:

\[
\bigcup_{R \in HLPV} R \cong (\prod_{LowerLevelPVUID, ObjectUID} R_{\text{oldPV}} \div ((R_{\text{oldPV}} \times R_{\text{remove}}) \cup R_{\text{add}}))
\]

Using the DDL/DML statements presented so far, backward compatibility is assured. Since property-views UIDs are constant throughout the lifetime of the database, accessing
data through 'old' property-views is enabled. However, keeping all property-views throughout the lifetime of the database might eventually fill up the database. Our aim is to enable clearing of the database from old property-views, while staying backward compatible with old applications. Of course, it is impossible to provide information which was removed from the database, however if the information is available under a different property view, it should be retrieved. To better understand the type of compatibility we are aiming for, we provide an example. Suppose that the database was created with a property-view defining the 'registered to courses under personal info' property which includes 'has Personal Information' and 'has Registered-Courses' property-views. After a while, the DBM decides that the registered courses information is irrelevant and creates a new property view to define the property 'registered under personal info'. This property is the same as 'registered to courses under personal info', only 'Registered-Info' is not part of its internal property-views. It is known to the DBM that the applications using 'registered to courses under personal info' do not use 'Registration-Info' anymore, and thus the property view 'registered to courses under personal info' is removed. However, old applications still access the property-view 'registered to courses under personal info' in order to get the 'Personal Info'. The applications do not access the 'Personal Info' PV directly because the result of doing that will be a relation with the personal information of every individual in the system. That is, it may include objects which are not registered to any course.

In order to enable the users to examine which property-view is potentially relevant we added the last Meta-data table, holding relation $r_{revolution}$. $r_{revolution}$ will have two properties (OldPVUID, NewPVUID). Using this table, the user indicates that the new property-view is guaranteed not to include objects that could not be accessed by the old property view. In this manner, if the database has many different users that are not familiar with the structure of the database schemes it may be that they will try accessing information through a well defined property-view rather than a property-view local to the DB. In order to stay compatible with such users, the $r_{revolution}$ relation will hold the same tuple. This type of work enables interoperability between different databases.
6.4 Compatibility with relational model capabilities

In this section we show that our data model is complete as it supports all the operations
supported in relational databases. Namely, SQL operations are supported. We will demon­
strate this capability of our data model by showing the model support DDL commands,
DML commands, and finally showing compatibility with DQL supported in SQL.

6.4.1 Support of DDL commands

The DDL command in SQL practically creates an class (entity type) or relation using table
structure. The DDL command has to following structure:

```
CREATE TABLE table-name ( column-name-1 type-1 [NOT NULL [WITH DEFAULT]]
[, column-name-2 type-2 [NOT NULL [WITH DEFAULT]]...]
[, PRIMARY KEY (column-name-list) ]
[, FOREIGN KEY [fk-name] (column-name-list) ]
```

The consequences of executing this command is the creation of an entity type with
attributes applying to column-names. Some of the attributes serve as unique identifiers of
entities within the entity type. Some of the attributes are actually relations with other
entities which belong to different entity types. These attributes will be found in the foreign
key list. An entity type of this kind would be a class in our data model. In order to create
such a class the following operations need to be done:

- Find a set of property-views which contains the set of properties which the column-
  names apply to.

- Define a new property-view as a class using the property-views found in the previous
  step.

Note the following

- In our data model unique identifiers within a property-view are automatically generated. Thus there is no need to define primary keys.
• There is no need to define foreign keys. All properties of a certain entity are defined as intrinsic attributes. These intrinsic attributes are weak properties of a mutual/relational attribute. For example, in the relational model we might want to model the property a student takes course by have in a foreign key attribute in the student entity type which relates to course. In our data model the class student would not posses a foreign key attribute. Rather, it would posses an intrinsic attribute 'takes some course'. This attribute would preced the mutual property 'student x takes course y'. It would be a weak property as well since once the student does not posses any instance of the property 'student x takes course y', the property 'takes some course' is lost as well.

6.4.2 DML commands

In SQL a relational schema which has already been defined can be extended at any time using the following command:

\[
\text{ALTER TABLE } table \text{ name} \\
\text{Add column name data type}
\]

Conceptually, this command adds properties to all members of the entity-type represented by the table. In our data model adding properties is done by creating a new property view and populating them with instances as needed. Thus this command is not needed in case the it is used merely to add properties in the system. However, in case a property-view represents a class, the exact same DML command would suffice to update the definition of a class by a specific user. Membership of an object to a class in the view of a specific user is examined by checking if all properties defining the class according to the user are possessed by the object. Also the property 'is_classname' needs to be possessed by the object in order for it to be part of a class.
6.4.3 Query language

The new data model supports all relational model queries and introduces capabilities to support new types of queries. In this section we demonstrate the models compatibility with SQL. Naturally, the new model imposes some new requirements on traditional SQL queries. These new requirements are demonstrated and explained in this section as well.

Selection and Projection

SQL selection queries are of the form:

\[
\text{Select Attribute - List from Class - Name where Condition.}
\]

Where \text{Attribute-List} corresponds to a list of attributes of members of Class. \text{Condition} applies to the selection condition of instances.

This query can be supported under the new data model, since classes are specific types of property-views. In other words, this SQL type query can represent only a subset of the set of queries available in our model.

The new data model will support the exact same query under the following conditions:

1. Use a \text{property-view-name} instead of \text{Class-Name}. Since a class is naturally defined in terms of sets of properties possessed by all the objects in the class, and since our model supports representing an aggregation of properties as a new property, as well as having properties indicating membership to classes it is obvious that \text{Class-Name} could be defined in terms of a property-view.

2. \text{Attribute-List} corresponds to a list of property-views which are derived only once from property view \text{property-view-name}

3. \text{Condition} is with respect to a low-level property-view which is derived only once from property view \text{Class}

In case condition two is not applied, \text{Attribute-List} must hold the exact path from \text{property-view-name} to the low-level property views requested. In a similar manner, in case
condition three is not applied Condition must hold the exact path to the property-view for which the condition should be applied. This way name conflicts are resolved.

**Join**

Join operations are used in relational languages in order to recombine data that have been split into several relations due to normalization. A join is used when there is a need to provide an answer regarding mutual or relational properties. Our model naturally supports joins since property-views are used to examine mutual properties in the same manner they are used to examine intrinsic properties. For example, using the relational model, in order to retrieve the academic status of a student, one would join the student relation with course relation via mutual properties of students and courses.

\[ \text{Select course.name from student, course where Course.id = student.major_course}. \]

In our model, the academic information would be retrieved using a single property-view, 'student X taking course Y'. This property view would connect the different property views which precede it- its defining property-views. specifically, 'student taking some course' and 'course taken by some student' would be accessible. Note that 'student taking some course' and 'course taken by some student' would hold information regarding different objects. In other words, the ObjectUID of properties accessed through 'course taken by some student' would be different from the 'ObjectUID' of properties accessed through student. This can be seen in the following diagram.

**Union and Set-difference**

SQL supports queries which operate on sets of results. The result of a union operation is a relation consisting of the union of to queries. In set difference, the result is the difference between the the set of instances in two queries. These type of operations are supported in our data model in the exact same way they are supported in the relational model. In other words the exact same syntax for the operation would have the same meaning in our data model.
Advanced Queries

As the proposed data model can be classless, the advantages mentioned in [Parsons and Wand 2000] are apparent in this model. Specifically, classless selection and projection is enabled. No longer do users need to define the scope of the query using the from clause (i.e. the set of objects on which the query will operate). By using a property view, queries such as the following are possible:

\[ \text{Select ObjectUID where Color = red.} \]

This query will retrieve all the object UIDs of objects which possess the general property of having color, and which the specific property for them is red. Having the general property of color implies that the object is accessible through the low-level property view of 'color'. Having a specific value of red implies that the value of the low-level property-view of color for the object is 'red'. When selecting ObjectUIDs from low-level property-views we simply access the property view mentioned in the where clause. Similarly, the new model supports queries regarding objects which possess different properties without knowing about higher
level property views. For example, queries such as the following are supported:

\[
\text{Select color, weight} \\
\text{from color, weight} \\
\text{where color.ObjectUID = weight.ObjectUID}
\]

This query retrieves the color and weight of all objects which have color and weight, no matter what other properties they may or may not possess.

Finally, the new data model supports queries regarding objects which possess general properties. For example, queries of such as 'The weight and name of all objects which have a car' can be queried in the following way:

\[
\text{select weight, name} \\
\text{from weight, name, has_car} \\
\text{where name.objectUID = weight.ObjectUID = has_car.ObjectUID}
\]

7 Summary

In this thesis we have described a data model that keeps semantic information associated with property precedence. It appears that many benefits can come about in using such a model. The model is based on the approach proposed in [Parsons and Wand, 2000] and further extends this work. In [Parsons and Wand, 2000] the need to separate classes from instances was identified and different problems arising due to inherit classification are described. The contribution of our work is in providing a definition of fundamental constructs needed to
support such an approach and in expending it. We identify new constructs, and provide new ideas to support integrity in such a model. Namely, we find that there is a need to present properties and the precedence relation between properties in order to enable a two-tiered approach. We introduce the property-view construct which provides means to represent properties as they are described in ontology. In other words, a mapping from property views to general-form properties in ontology was described. We expended the two-tiered approach to support more flexible classification scheme. We suggest a way to enable different classification schemes by different users to co-exist. While in the two-tiered approach all users have to agree on classes, classification in our model is more flexible. Only the property of being part of class is shared by all users. Each user may define a different set of properties which overwrite\textsuperscript{24} the conditions of possessing the property of belonging to the class. This overwriting would not influence other users. Finally, we show the applicability of the two-tiered approach and its extensions by presenting a DBMS architecture and implementation to support such a model.

Indeed there are some limitations to the model in general, and to the proposed implementation specifically. While the model supports ontological constructs which are not supported in most data models, it may be somewhat difficult to understand the data model conceptually. In traditional data models the different attributes of object can be gathered easily by looking at objects which represent the entities. In our data model, the information is scattered in different property views and users may find it difficult to find all information associated with a particular entity. Another limitation of our model is in the proposed implementation. There is no doubt that efficiency in time and space is lost when compared to traditional data models. It seems like this traditional trade-off between expressiveness and efficiency always comes about. One of the major reasons for the success of the relational model is due to its efficiency which is hard to be reached in other data models. As is indicated in the next section, future research in finding other more efficient implementation mechanism is needed.

\textsuperscript{24}In other words, refine
Nevertheless, there are many benefits which come about using this data model. Specifically, consistency interpretability, extensibility and data independence may improve. Moreover, we believe that this model can prove more understandable to users, as it is consistent with some of the principles of BWW ontology. Ontology is a branch of philosophy which can serve as a sound foundation to guide how real-world phenomena are perceived and naturally understood by humans. In our work, we first defined the type of precedence information which we aimed to incorporate in our data model, the super-value-manifestation precedence information. We showed how incorporating such information in the data model can improve consistency and enable different views of the data. Different users can examine different properties. Modifications to properties propagate to their related properties: the preceded and preceding ones. We also showed how using distinct properties can bring about a very loose classification scheme. Every class can be associated with a property indicating possessing all non-value properties. Every user can augment his own definition of the class with additional value properties he/she finds necessary. We proposed the Meta data constructs for holding the necessary precedence information. We further analyzed what type of information is needed from the user in order to construct and populate the database. In this manner, additional precedence information and constraints were suggested. These include cardinality information, as well as set and list type information. We then outlined some of the implicit operations done by the DBMS when processing user commands. Most of the internal operations are associated with enabling property-views to define classes as well as precedence information. We demonstrated how our data model solves different problems evident in traditional data models. Moreover, it is evident that using this data model traditional queries are still supported, while additional query capabilities are enabled. Using this model queries that cross-cut class boundaries are supported, and queries regarding the definition of properties according to properties that precede them are enabled as well. This latter querying ability highlights the interoperability implications of the data model. Different users can extract information about properties and their meaning.
8 Future Research

In our work, we have developed a data model to support a very specific precedence type. Precedence which was based on what we named super-value-manifestation. In this type of precedence the database is compatible with all traditional database operations, and exhibits much more flexibility. However, future research can be done to extend the model and support representing other precedence relations. For example, representation of precedence rules which include properties which do not relate to each other in a super-value-manifestation relation. Most of the consistency operations can be expended in a straightforward manner, however inserting new data would probably imply more extensive modifications in the model. Expanding the representation capabilities may lead to different needs in the model, such as querying the user for values associated with preceded properties, or having rules to extract values from existing ones. For example, if we allow any precedence information to be defined using the data model. Information such as 'goes_to_university_from_date' may precede 'pays_tuition_fee'. Once the user would add the property 'goes_to_university_from_date' to some object, the DBMS would need to query the user what tuition fee is payed by the object. Alternatively, some rule may be applied such as 'multiply start year by 2 to get tuition fee'.

In this paper we presented one way in which the model can be implemented, and proposed some of the alternatives which were similar in nature. More research can be done in finding different implementation solutions, and analyzing their performance. This may include using object-oriented data models and deductive data models rather than the relational model as the implementation framework. Also, different algorithms for updating data and assuring consistency can be examined.

Another area for research can be extending property-views to support different units of values. In such a model a mapping between measuring units should be defined by the user. Different property-views will access data with different units of measurement, and the user would be able to choose the desired unit of measurement. Incorporating this type
of flexibility to the model will demand extending consistency checks. Namely, the type of operation permitted with different units of measurements is derived from the type of mapping available between the different units of measurements (one to one /onto etc.). Finally, design and implementation of a database using this data model can provide much information regarding the advantages and disadvantages of using such a model. It can also identify different areas of concern and possible improvements to the model as well.
## Appendix- Table of Notations

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manifestation</td>
<td>A manifestation of a property GP in an object X, is a property P, possessed by X and preceded by GP.</td>
</tr>
<tr>
<td>Value manifestation</td>
<td>A value manifestation is a manifestation of a property GP by P, where P provides additional value data not available in GP.</td>
</tr>
<tr>
<td>Super-value-manifestation</td>
<td>Super-value-manifestation means that there is another generic property GP2, which in case possessed by the object X, the value manifestation of GP2 will have the same values as the value manifestation of GP (which is P) plus additional values.</td>
</tr>
<tr>
<td>A weak property</td>
<td>A weak property is a property which may exist only in case a property preceded by it ceases to exist</td>
</tr>
<tr>
<td>Predicate form of general attributes</td>
<td>Predicate form of general attributes has the following structure [Wand, pp 499]: A: T₁ x ... Tₙ x V₁ x ... Vₘ --&gt; statement regarding A</td>
</tr>
<tr>
<td>A property-view</td>
<td>A property-view is a means for representing general attributes in a database</td>
</tr>
<tr>
<td>An instance of a property view</td>
<td>An instance of a property view is a means for representing specific attribute statements in a database</td>
</tr>
<tr>
<td>Low-level property-view</td>
<td>Let A be a general attribute given in a predicate form, which involves a set of things and a set of values. A: T₁ x V₁ --&gt; statement regarding A The low-level property-view of A is composed of an identifier, the set of things T₁ and a set of values V₁</td>
</tr>
<tr>
<td>Middle-level property-view</td>
<td>Let A be a general attribute given in predicate form, which involves one set of things A: T₁ x V₁ x ... Vₘ --&gt; statement regarding A A: T₁ x V₁ x ... Vₘ --&gt; statement regarding A The Middle-level property-view of A is composed of an identifier, the set of things T₁ and sets of values Vᵢs. An ordering is imposed only between identical sets of Vᵢs</td>
</tr>
<tr>
<td>High-level property view</td>
<td>Let A be a general attribute given in predicate form A: T₁ x ... Tₙ x V₁ x ... Vₘ --&gt; statement regarding A The High-level property view of A is composed of an identifier, sets of things Tᵢ and sets of values Vᵢ. An ordering is imposed only between identical sets.</td>
</tr>
<tr>
<td><strong>Defining PVs</strong></td>
<td>A property view can be expressed in terms of defining PVs, an ordering of repeated defining PVs, and a unique identifier.</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Explicitly define V_i</strong></td>
<td>The set of low-level defining PVs, are said to explicitly define a component of the general attribute, represented by the defined property view. Specifically, they represent a V_i Component.</td>
</tr>
<tr>
<td><strong>Explicitly define T_j</strong></td>
<td>The set of middle-level defining PVs, which are used to T_j, are said to explicitly define a component of the general attribute, represented by the defined property view. Specifically they represent a T_j Component.</td>
</tr>
<tr>
<td><strong>Implicitly defined components</strong></td>
<td>All components of the general attribute, which are not explicitly defined, are said to be implicitly defined by the middle or high-level defining property-views.</td>
</tr>
<tr>
<td><strong>Preceded/non-preceded components</strong></td>
<td>Let HPV be a high-level property-view representing a general attribute GA. The components of GA defined in HPV using low-level property views are called non-preceded components. The rest are called preceded components.</td>
</tr>
<tr>
<td><strong>Path between property views</strong></td>
<td>The path from property-view PV_i to another property-view PV_n is an ordered list of property views PV_1...PV_n where each property view is part of the defining PVs of its predecessor. In case some PV_i has multiple occurrences of PV_n or PV_i is a property-view of type list, PV_i will hold an index indicating its order in the definition in PV_i.</td>
</tr>
<tr>
<td><strong>Fully preceded property-views</strong></td>
<td>Property-views in which all the V_s and T_s of their corresponding general property, can be reconciled from preceding properties.</td>
</tr>
<tr>
<td><strong>Partially preceded property-views</strong></td>
<td>Property-views in which some of the V_s and all of the T_s of their corresponding general property, can not be reconciled from preceding properties.</td>
</tr>
<tr>
<td><strong>Relational property views</strong></td>
<td>Property-views define a relational property. These property views explicitly define some of the T_s of their corresponding general property.</td>
</tr>
</tbody>
</table>
Bibliography


...database model, ACM Transactions in Database Systems, 6,3, September 1981, 351-386.


http://www.w3.org