A formal definition of an object-oriented data/knowledge model*

Jonas A. Montilva C.
Universidad de Los Andes,
Facultad de Ingenieria,
Departamento de Computación,
Grupo de Investigación en Ingeniería de Datos y Conocimiento
Merida, Venezuela
e-mail: jonas@ing.ula.ve

The integration of databases and knowledge bases has become the purpose of many new AI technologies such as Intelligent Databases, Knowledge Base Management Systems, and Expert Database Systems. Object orientation has long been recognized as an appropriate approach for achieving the integration of data and knowledge management. This paper presents an object-oriented model for representing, storing and manipulating data and knowledge. The model is the result of extending the Smalltalk-80 object model, in order to incorporate the constructs needed to support object-oriented data/knowledge management. The emphasis in this paper is the formalization of the modelling constructs provided by the model. Formalization is considered an essential requirement for understanding, communicating, implementing and using properly the model. The formal definition of the model was accomplished using a well-known formal specification language, the Z Notation, which uses a model-based approach to formalization.

1 Introduction

The integration of databases and knowledge bases (DB/KB integration, for brevity) is being addressed by new domains of computer science such as expert, deductive, object-oriented and intelligent databases. A common characteristic of these areas is that the DB/KB integration is achieved by integrating existing software technologies (e.g. models, languages, and systems) from different areas or disciplines. Intelligent databases, for instance, have emerged from the integration of models and languages from the following areas: object-orientation, expert systems, databases, and hypermedia [21].

The ability of the Smalltalk-80 (ST-80) language, and its underlying object model [12], to represent using objects the structure, behaviour and relationships of entities or concepts of a given application domain, as well as its capability to organize these objects into classes, makes it a good candidate for representing both data and knowledge. It lacks, however, the ability to represent default values, attached predicates, constraints, and composite objects, which are

* *ACTAS del V Congreso Iberoamericano de Inteligencia Artificial*. (Caracas, Octubre, 1994). McGraw-Hill, pp. 62-75.

essential in representing knowledge. Besides, the ST-80 language does not support persistence which is crucial in data/knowledge management. Therefore, to be useful for the representation and management of data and knowledge, both the ST-80 language and its underlying object model must be extended with features borrowed from object-oriented databases and knowledge representation.

In this paper, we present an object-oriented model for the management of data and knowledge bases, called the D/K model. This model is the result of extending the ST-80 model with features borrowed from ORION -an object-oriented database management system [13]- and a frame-based knowledge representation scheme. The model is intended to be used in the design, creation, manipulation, access, and query of object-oriented data/knowledge bases. A data/knowledge base is defined here as a collection of persistent objects that represents data and knowledge about entities or concepts in a particular application domain.

A description of the D/K model is given here in terms of the concepts it supports and the structure and semantics of its components. In order to describe the meaning of the constructs, the view of the world assumed by the model is considered here as an important aspect of the definition of the model. The meaning of each construct is explained by defining a denotational relationship between the model and the view. This relationship indicates what construct should be used to represent a given aspect of that part of real world being modelled.

The emphasis here is, however, in the formal definition of the constructs of the model. We believe that the formalisation of a model is an essential step of its design, because it allows the designer to explain with precision and without ambiguity the structure and semantics of the components of the model. The Z Notation [23], a formal specification language based on Set Theory and Predicate Calculus, was used to formalize the constructs of the model. This language is used for specifying the properties of each of the constructs of the D/K model. A formal definition of the notion of data/knowledge base is given using this notation too.

The organisation of the paper is as follows. Section 2 describes how the model was designed. Section 3 introduces the components of the model. The semantics of the constructs of the model is given in Section 4. A formal definition of the constructs of the model and its modelling rules is presented in Section 5. Finally, the operations of the model are enumerated in Section 6. Complete details of the formal definition of the model and the syntax and semantics of its associated language, the D/K Language, are given in [20].

2 The design of D/K model

The ST-80 language is based on an object-oriented model whose basic modelling constructs are the following: object, class, instance variable, method and message. This model is founded on concepts such as data abstraction, encapsulation, multiple inheritance, extensibility and name overloading. A basic feature, which makes the ST-80 model a good candidate for manipulating data/knowledge bases, is that classes are also objects. A class, therefore, has its own variables and methods and can be modified dynamically in the same way than its instances are.

Since the ST-80 model does not support database concepts, such as persistence, schema definition and evolution and query, a data model was therefore required to extend the ST-80 model. The ORION model was chosen because of its compatibility with the ST-80 model, its expressive power for representing composite objects, and its semantics for schema evolution which is essential for the management of DB/KBs.

Concepts such as constraints, attached predicates, default and generic values, which are required in object-oriented knowledge bases, were borrowed from frame-based knowledge representation schemes in AI. The lack of standards in this domain precluded the selection of a particular model or scheme. For this reason, we opted for building a frame reference model which describes formally most of the basic features exhibited by frame knowledge representation schemes. It is described elsewhere (see [20]).

The integration of concepts, constructs and operations coming from three different models was recognized as a complex problem for which a software integration process was needed. Since no methods for the integration of models existed, by the time we started this research, we opted for developing our own method for pursuing the integration of the three models (see [19, 20]).

The integration method is composed by four phases: pre-integration, conceptual analysis, conceptual comparison and conceptual integration. The pre-integration phase is concerned with the selection of: (1) the models to be integrated; (2) the integration strategy to be used (i.e., extension, transference and combination); and (3) the order of integration to be applied if more than two models are to be integrated. The conceptual analysis phase entails the identification and description of the features of each model being integrated. Its purpose is to gain an understanding of the properties of each model. For each participating model, a *core* - a subset made of selected constructs, operations and rules of the model - is defined based on the integration requirements [17]. Each model core is then represented, or formally defined, using an appropriate modelling notation. The result of the representation process of each model core is referred here to as *core schema*. The conceptual comparison phase involves the identification of similarities and differences between the features of the model cores. The resulting list of similarities and differences is used in the next phase to help identify the points of integration between the cores. Finally, in the conceptual integration phase, the model cores are merged using the core schemata and according to the selected strategy. The integrated model is validated against the given set of integration requirements (see [17]) and refined by iterating the process until an appropriate solution is found.

This method was later extended to provide a more comprehensive framework for the integration of data and knowledge bases, in which the integration can be achieved at three different levels: models, languages and systems (see [18, 20]).

3 Concepts and components of the D/K model

The D/K model is an object-oriented data/knowledge model. It includes a set of object-oriented constructs, operations and rules for managing data/knowledge bases. These components are founded on a set of concepts provided by the ST-80 object model [12], the ORION data model [13] and a frame-based knowledge representation model based on KEE [11], STROBE [14], and the Minsky´s notion of frame [16].

From the object-oriented language realm, the model supports object identity by surrogates, a weak view of encapsulation with private and subclass visible protections, class intension separated from class extensions, class specialization, multiple inheritance, metaclasses, parametric polymorphism, message name overloading, redefinition of inherited attributes and constraints, aggregation and message passing. From object-oriented databases, the model inherits composition by attributes using exclusive and shared references, persistence by reachability and orthogonal to instances, and schema evolution. Finally, from frame-based knowledge

representation, the model uses procedural attachment, default values specified as attribute facets, generic values specified as class attributes, and constraint definition.

In addition to its three components - *constructs, operations* and *modelling rules* (see Table 1) - the model includes a rich set of classes, most of which are orientated towards the target applications of the model: multimedia and spatial applications. This set is organized in four levels. The core level includes a set of basic classes for defining integers, floats, characters, collections, etc. The graph level comprises a set of classes for supporting graph concepts, such as directed graphs, undirected graphs, chains, paths, cycles, and circuits. The multimedia level is made of a set of classes which implements multimedia abstract data types (e.g., text, graphics, and images) and hypermedia concepts. Finally, the spatial level includes the notion of spatial object and a set of classes which implement abstract data types required for spatial applications. Basic and specialised classes are the building blocks for creating data/knowledge bases in multimedia and spatial applications. Specialized classes are described in [20].

The purpose of these four level of classes is to make available to the user the support needed to built multimedia and spatial data/knowledge bases. The user creates the schema of a data/knowledge base by specialising classes provided by these levels. The objects stored in the database are created by instantiating either the specialised classes of the model or the classes created by the user.

CONSTRUCTS	**OPERATIONS**	**RULES**
Object	Definition	Inheritance
Class	Evolution	Persistence
Class extension	Manipulation	Class Hierarchy
Instance	Query	Composition
Attribute		
Facet		
Constraint		
Method		
Message		

Table 1: The components of the D/K model

4. The semantics of the constructs of the model

The constructs are the symbols that the user employs to represent knowledge about the objects of interest in a given application domain. Each construct represents some aspect of that part of the world being modelled. Using the constructs as representational tools requires an understanding of the way in which we conceive the world, i.e., a *view of the world*. In order to describe the semantics of the constructs, we therefore describe first the view of the world assumed by the model and then map the constructs of the model to this view.

The view used to design the D/K model is based on the ontological theory proposed by M. Bunge [10]. According to this theory, the world is composed of things. A *thing* may be an entity (concrete object) or an abstract object. All things possess *properties*. Things that possess the same set of properties form a *kind*. The set of all values associated with the properties of a thing at a given time is called the *state of the thing*. A thing changes its state with time. A change of state, called *state transformation*, is caused by an *event*.. Things comply with laws. A *law* is an invariant relation between two or more properties or a restriction on the values of a property.

This view of the world is supported by the model through the following constructs: objects, classes, class extensions, instances, attributes, facets, constraints, methods, and messages. Each of these constructs is used to represent or denote a particular element of the view. The denotational relationship between the constructs of the model and the elements of the view of the world is defined by the mapping shown in Table 2.

Objects are the basic modelling units in the D/K model. An *object* is a construct that represents or stands for a domain object. Each domain object that is relevant to the problem being modelled is represented in the data/knowledge base by an object. An object represents either an entity or a concept from the application domain. Domain objects are organized, in the view, into kinds. *Classes* and *class extensions* are two constructs of the model that capture the notion of kind in the given view of the world. A class represents the intension of a kind of domain objects (i.e., their common properties), while a class extension captures the notion of extension of a kind, that is, the set of the domain objects referred to by the kind. Each class extension is made of a set of objects called *instances*. An instance represents an entity or individual object of the domain.

Element of the View	**D/K Construct**
Domain object	Object
Entity or Concept	Instance
Kind	Class
Property	Attribute
Law	Facet
Law	Constraint
Action	Method
Event	Message

Table 2: Denotational relationship between the D/K model and its view of the world

Each instance is made of another type of construct called *attribute*. An attribute represents a known property of a domain object. Each attribute has associated one or more values. The notion of state of a domain object is captured by a set of attribute values which is also called the *state of the object*. *Facets* are constructs associated with the definition of attributes. Laws that determine the lawful state space of the domain objects of a kind may be represented using facets. Facets are therefore used to specify the domain of the values of an attribute and to constrain the value of an

attribute.

Complex laws are represented using the construct *constrain*t. A constraint denotes a restriction imposed on properties values of a domain object or an invariant relation between two or more properties of the object. Constraints can be either associated with the definition of an attribute using facets or defined explicitly using slots.

Methods are used in the D/K model to represent those actions - operations or processes - of the application domain that change the state of the domain objects. *Messages* represent events that signal the beginning of an action in the application domain. Messages are used in a data/knowledge base to trigger the execution of methods.

The formal definition of these constructs is given in the next section.

5 Formalising the constructs and rules of the model

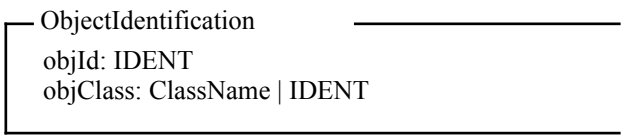
We used a model-oriented approach to the formalisation of the constructs of the model. In this approach, the state of each construct and the operations that change this state are represented in terms of a formal language. The so-called Z notation [23] was used for this purpose. We preferred the model-based approach to others, such as the algebraic approach [9] and the set-tuple approach [15], for two reasons. Firstly, the model-based approach, as used in Z, is more consistent with the orientation of the D/K model. Each construct is defined in terms of its structure and behaviour, using the Z-schema symbol, in a similar way as the D/K model defines classes. Secondly, the Z-schemata of the D/K model can be easily refined in order to obtain a specification for the implementation of the model, which helps to reduce the effort required in building a prototype for the model.

In this section, the main constructs of the model are formally defined in terms of the Z notation. Each z-schema defines the structure of a construct and captures some details of its semantics. A complete definition of all the constructs is given in [20], together with the semantics of the operations which are defined separately (see Section 6).

5.1 Objects

The basic unit of modelling in the D/K model is the *object*. There are three kinds of objects in the core level: instances, classes, and metaclasses.

Each object has a surrogate called *object identifier* (*objId*, for short). In addition to this identifier, an object has associated a class and a set of values. The class describes the structure (attributes) and behaviour (methods) of the object. The values represent the state of the object at a given time. Depending on its kind, an object refers to its class either by the class name or the class identifier. For convenience, we will refer to the *objId* and the class reference of an object by the following z-schema:



5.2 Classes

Classification is essential in the D/K model. Instead of describing each object separately, a group of objects with similar attributes is collectively described by means of a class. A *class* defines the structure and behaviour of each of its objects, called *instances*. The structure of the instances of a class is specified by attribute definitions, in the body of the class; meanwhile the behaviour is specified by methods.

Contrary to other object-oriented data models (e.g., ORION [13] and OSAM* [22]) which overload the construct class with an intensional and extensional role, classes in the D/K model have only assigned the intensional function, that is, to define the structure and behaviour of its instances. The extensional function of a class (i.e., the collection of its instances) is assigned to another construct called *class extension*. The advantages of this approach are discussed in section 5.4. The state of a class in the D/K model is represented by the z-schema shown below.

Each D/K class is an object that, in addition to its object identifier, has a unique name used also for reference purposes. *ClassName* refers to the set of all the class names used in particular data/knowledge base schema.

The class of a class is another object called *metaclass*. A class refers to its metaclass using the metaclass identifier (invariant 1). As specified by the second invariant, a metaclass is unique to each class. A metaclass describes the structure (class attributes) and behaviour (class methods) of a class as an object.

Each class has at least one superclass and zero, one or more subclasses (multiple inheritance). Each class provides references to each of its super or subclasses through their names. The relationships between classes is captured by a separated z-schema named *ClassHierarchy* (see Section 5.3). As specified by the last invariant, all the attributes, constraints, and methods of the superclasses are inherited by the class. Superclasses may be added or deleted dynamically to a class, as described by the schema evolution operations of the model listed in Section 6.

Class

ObjectIdentification [metaclassId/objClass]
name: ClassName
superclasses: F_1 ClassName
subclasses: F ClassName
instAttributes: F NAME
instAttrDefs: NAME → AttrDefinition
constraints: F NAME
constraintDefs: NAME → Constraint
instMethods: F MethodName
instMethDict: MethodName → Method
extensionNames: F NAME
extensionDict: NAME → ClassExtension

---

(1) metaclassId ∈ IDENT
(2) $\exists_1$ m: Metaclass • metaclassId = m.objId
(3) ∀ s: superclasses •
class(s).instAttributes ⊆ instAttributes ∧
class(s).constraints ⊆ constraints ∧
class(s).instMethods ⊆ instMethods ∧
[other invariants omitted]

A class has associated two kinds of attributes: *instance attributes* and *class attributes*. Instance attributes determine the structure of each instance of the class, i.e., each instance has one value for each attribute specified by the variable *instAttributes*. Class attributes , on the other hand, are attributes of the class as an instance of a metaclass and are therefore attached to the metaclass itself. Class attribute values are common and directly accessible to all instances of the class.

Each attribute specified in a class has associated a set of facets called *attribute definition*. An attribute definition specifies the properties of an attribute, e.g., the domain of the attribute values, constraints on these values, attached messages, and the default value to be used when the value is unknown. Attribute names are unique within the class in which they are used. The state of an attribute definition is represented as follows:

AttrDefinition

facets: FacetType → FacetValue

∀ f ↦ v: facets •
(f = domain ∧ v ∈ ClassName) ∨
(f = default ∧ v ∈ Instance) ∨
(f = constraint ∧ v ∈ Constraint) ∨
(f = uniqueOn ∧ v ∈ ClassExtName) ∨
(f ∈ {composite, dependent, exclusive, nullAccepted} ∧ v ∈ Boolean) ∨
(f ∈ {ifNeeded, ifAdded, IfRemoved} ∧ v ∈ Message)

where:

FacetType ::= domain | default | constraint | composite | dependent | exclusive |
ifNeeded | ifAdded | ifRemoved | nullAccepted | uniqueOn

FacetValue = ClassName | Instance | Constraint | Message | Boolean | ClassExtName

A class may also have associated a set of constraint slots which are used to define semantic integrity constraints. The state space of a constraint in the D/K model is captured by the following z-schema:

Constraint

condition: Proposition checkOn: F MethodName ifSatisfied: F Message ifViolated: F Message

A constraint has a *condition* that is evaluated when a specific method, indicated by the variable *checkOn*, is executed. The variables *ifSatisfied* and *ifViolated* indicate the actions to be taken (messages) when a condition is satisfied or rejected, respectively.

The following example illustrates the definition of a class using the syntax of the D/K language, an object-oriented language based on the D/K model [20].

```
DKClass  subclassName: #Road
         superclasses:   { SimpleChain[RoadSegment] }
         classExtName: Roads
         classExtType:  Dictionary  keyedBy: roadNum
         instAttributes:
              { rsegments: { redefines:      progression;
                             composition:  true ;
                             dependent:    true }
                roadNum:   { redefines:      name;
                             uniqueOn:     Roads;
                             nullAccepted:     false }
                roadType:  {  domain:        String;
                             constraint:    { condition:
                               (roadType = "motorway "  |
                               roadType = "roadTypeA"   |
                               roadType = "roadTypeB" ) }
                length:       { domain:  Float;
                             ifNeeded:      [ self  calcLength] }
               ... }
```

5.3 Class hierarchies

The model supports two types of relationships between classes: *specialisation* and *composition*. These two relationships are essential in building a data/knowledge base schema; they define two kinds of hierarchies: *class hierarchy* and *class-composition hierarchy*.

A new class is created by specialising an existing class(es). The new class inherits all the attributes, constraints, and methods defined for each of its superclasses. This relationship creates a class structure called *class hierarchy* which can be graphically represented by a directed acyclic graph whose nodes denote classes and its edges the relationship itself. The state space of

a class hierarchy is modelled by the following z-schema.

For any hierarchy of classes, there exists a root class named in the z-schema as *topClass*. Except for the topClass, each class has at least one superclass (invariants 2 and 3). The set *subclassRel* includes all the direct subclass/superclass relationships between the classes of the hierarchy, as defined by invariant 4. It is assumed that each class has a unique name within the hierarchy (invariant 5). Since a class hierarchy forms a directed acyclic graph rooted at the class *topClass*, a class cannot be a superclass of itself and any cyclic relationship between two or more classes are not allowed (invariant 6). The function *allSupersOf* returns all direct or indirect superclasses of a given class, whereas the function *class* returns a class given its name.

ClassHierarchy

classes: F Class
topClass: ClassName
subclassRel: ClassName ↔ ClassName

---

(1) class(topClass) ∈ classes
(2) topClass ∉ **dom** subclassesRel
(3) ∀ c: classes | c.name ≠ topClass • c.name ∈ **dom** subclassRel
(4) ∀ (n,m): subclassRel • n ∈ class(n).subclasses ∧ m ∈ class(n).superclasses
(5) ∀ x,y: classes | x ≠ y • x.name ≠ y.name
(6) ∀ (n,m): subclassRel • n ≠ m ∧
(m,n) ∉ subclassRel ∧
¬∃ x: ClassName | (x,n) ∈ subclassRel ∧ x ∈ allSupersOf(n)

The composition or *partOf* relationship is supported in the D/K model through the aggregation of attributes. The semantics of the composition relationship has been borrowed from the ORION data model [13]. A composite object is a tuple object in which some of its values are references to its component objects. As in the ORION model, the semantics of a reference in the D/K model has been extended to accommodate the composition relationship. The definition of the composition relationship is therefore embedded into the attribute definitions of the composite class definition, as indicated by the *AttrDefinition* z-schema, in the previous subsection.

5.4 Instances

Each object in the D/K model is created by instantiating its class using the create instance methods. An object created in this way is called an *instance*. The structure and behaviour of an instance is completely defined by its class. Each object has a state which is made of values determined by the kind of the instance. The D/K model has three different kinds of instances: *basic, aggregate*, and *collection*.

A basic instance is a self-identifying object whose class is one of the basic classes of the model: integer, float, character, string, and boolean. A basic instance differs from the two other kinds in the following aspects: (1) a basic instance does not have attributes; its state is made of a single value, and (2) a basic instance does not have a surrogate, because it uniquely identifies itself (i.e. its identifier is its own value).

An aggregate instance is made of a collection of attributes. Its state space is represented below.

The state of an aggregate instance is made of one or more attributes. Each attribute has a value (*AttValue*) which could be a basic instance (e.g. an integer, a float, or a string), a reference to another object, or the constant nil .

The properties of each attribute of an instance are defined by the instance's class. As specified by the third invariant, each attribute value must be consistent with its corresponding attribute definition. In particular, if an attribute value has already been initialised or set to a value different than *nil* , it must be an instance of the class specified as the domain of the attribute in the definition of the class. The function *isInstanceOf* determines the type of correspondence between a value and a class.

A collection instance, on the other hand, is an object that collects one or more instances of one or more classes. A collection instance can be homogeneous or heterogeneous. It is said to be homogeneous if all its members are instances of the same class, otherwise, it is heterogeneous.

AggregateInstance

ObjectIdentification
state: Name $\rightarrow$ AttValue

---

(1) **dom** state = class(objClass).instAttributes

(2) $\forall$ (n, v): state | v $\neq$ nil $\bullet$
$(\exists_1$ (m, w): NAME $\rightarrow$ AttrDefinition |
n = m $\wedge$ m $\mapsto$ w $\in$ class(objClass).instAttrDef $\bullet$
v *isInstanceOf* (w *getValueAt* domain)

Each collection object is an instance of a collection class. A collection class may also be classified as homogeneous or heterogeneous. In the set of system-defined classes provided by the model, String, Text, List, SetOf, ArrayOf, ListOf, OrderedCollectionOf, Dictionary, and DictionaryOf are all homogeneous collection classes; whereas Set, Array, and OrderedCollection are heterogeneous collection classes.

Each instance of a homogeneous collection class is a collection object whose members (other instances) are all of the same specified class. The following z-schema specifies the state space of a collection instance:

CollectionInstance[X]

AggregateInstance
members: F X
kind: HomogeneousCollectionName

5.4 Class extension

In the D/K model, the extensional function of a class is attributed to a construct called *class extension*. A class extension is a collection instance whose members are all references to the persistent instances of the associated class or of its subclasses, as defined next:

ClassExtension

CollectionInstance[InstID]

extName:	Name
class:	ClassName
∀ i: members • (i.objClass = class) ∨ (i.objClass *isSubclassOf* class)	
kind ∈ ClassExtensionType	

The kind of a class extension is an homogeneous collection class, such as arrayOf, setOf, and dictionaryOf. Note that, according to the first invariant, the extension of a class is made not only of instances of the class but also of instances of its subclasses. This is because each persistent instance of a class is also a persistent instance of its superclass. The function *isSubclassOf* determines whether a class is a subclass (direct or not) of another class.

We have favoured, in the D/K model, the separation of the intensional and extensional functions of a class for the following reasons. Firstly, the distinction provides more flexibility to the user by allowing him/her to select the kind of extension that is more convenient for the instances of a class. Instead of having the extension always predefined as a set, the user may specify, in the definition of a class, a preferred or appropriate type for the class extension. Secondly, separating the extension from a class makes it possible to associate more than one class extension to the same class. An instance of a class may appear in more than one class extension. The user decides during instantiation in which class extension(s) the instance should be stored. This feature is particularly important for supporting versions. The different versions of the instances of a class can be organised using class extensions.

As defined by the schema above, class extensions are collection objects and can be manipulated as any other instance of collection classes. In addition to the collection object operations, the extensions of the same class can be manipulated using algebraic operations such as union, intersection, and difference.

This characteristic of a class has an implication for the way queries are formulated. Since class extensions are separated from their classes, a query message for selecting one or more instances of a class *C* cannot be sent to *C*. Instead query messages must be sent to the class extensions associated with *C*.

5.7 Data/knowledge bases

The notion of *data/knowledge base* may now be introduced. A data/knowledge base is a collection of objects (classes and instances) which represent static and dynamic properties of a particular application domain. This notion integrates the concepts of database and database schema, as used in object-oriented databases.

A database in the D/K model is a collection of persistent objects. A persistent object may be an instance of a class or a class extension. Instances, in particular, represent factual or extensional knowledge about the application domain of the database. Class extensions are used for organising and accessing the instances in a database.

The state space of a D/K database is represented as follow:

Database

dbContent: F DBObject
dbTable: IDENT → DBObject

---

ran dbTable = dbContent
dbTable = {p: DBObject • p.objId ↦ p}

The function *dbTable* determines the access to the objects stored in the database through their identifiers.

On the other hand, the database schema is defined in the D/K model as a collection of classes that represent abstract or intensional knowledge about an application domain. Classes in a database schema are related through the specialisation (*isa*) relationship and the composition (*partOf*) relationship. These two relationships are captured by the two hierarchies - class hierarchy and class composition hierarchy - already discussed in Section 5.3. A database schema is formally defined as:

DKSchema == ClassHierarchy ∨ ClassCompositionHierarchy

An important property of databases and database schemas in the D/K model is that both are made of objects. The practical implication of this is that the user is allowed to create, manipulate, and access both classes and instances simultaneously and in the same way, because there is no distinction between them. Classes may be created dynamically together with their instances. Based on this property, we specify the state space of a data/knowledge base by merging the z-schemata *Database* and *DKSchema*, as shown below.

Note that the function *classExtTable* relates the classes of a data/knowledge base with their class extensions. The variable *classNames* defines the class name space for each data/knowledge base.

Data/KnowledgeBase

DKSchema
Database
classExtTable: ClassName → F ClassExtension
classNames: F ClassName

∀ i: dbContent • class(i.objClass) ∈ classes
⋃ **ran** classExtTable ⊂ dbContent
classNames = { c:classes • c.name }
∀ c: **dom** classExtTable • (∃ e: classExtTable(c) | e.class = c)
dom classExtTable ⊆ classNames

In the next section, we introduce the operations provided by the D/K model for the creation, manipulation, modification and query of data/knowledge bases.

6 The operations of the model

Object-oriented models are characterized by a large and ever expandable number of operations associated to classes. In this section, we present only a classification of these operations, as required for defining, manipulating, and querying data/knowledge bases in the D/K model. The classification and semantics of these operations are adapted from the taxonomy and semantics of schema evolution used by the ORION model [1]. Appropriate modifications to that taxonomy

were made in order to accommodate the constructs class extensions, constraints, and attached message which are not supported by the ORION model. The resulting classification of the D/K operations is the following:

1.- Schema definition and evolution operations:
- 1.1.- Class definition operations:
  - 1.1.1.- Create a new class.
- 1.2.- Changes to the definition of an existing class:
  - 1.2.1.- Add a new attribute definition.
  - 1.2.2.- Change/delete an existing attribute definition.
  - 1.2.3.- Add a new instance/class method.
  - 1.2.4.- Change/delete an existing method.
  - 1.2.5.- Add a new slot constraint.
  - 1.2.6.- Change/delete an existing slot constraint.
- 1.3.- Accessing the definition of an existing class.
- 1.4.- Adding class extensions to a class.
- 1.5.- Class hierarchy operations:
  - 1.5.1.- Delete a class from a hierarchy.
  - 1.5.2.- Add/delete a superclass to an existing subclass.
- 1.6.- Class composition operations:
  - 1.6.1.- Add a composite attribute to a class.
  - 1.6.2.- Change/delete a composite attribute from a class.
  - 1.6.3.- Change the dependency/sharability of a composite attribute.

2.- Data manipulation operations:
- 2.1.- Create transitory/persistence instances of a class.
- 2.2.- Delete one or more instances from a class extension.
- 2.3.- Set/update the value of an attribute.
- 2.4.- Get the value of an attribute.

3.- Query operations:
- 3.1.- Select a subcollection of instances that satisfies a predicate.
- 3.2.- Apply a block of code to each instance of a class extension.
- 3.3.- Select a subcollection of instances that does not satisfy a predicate.
- 3.4.- Union, intersection, and difference between extensions of a class.

A complete description of each of these operations and their semantics are presented in [20]. These operations are based on the message-passing computational model used by the ST-80 model. Each operation is a message sent to an object which could be an instance, a class, a class extension, etc. For example, the definition and creation of a class is a message sent to **DKClass**, the superclass of all the classes of the D/K model. Similarly, the modification of a class, and the creation of instances are messages sent to the class itself. Queries are also messages sent to class extensions.

7 Conclusions

We have introduced in this paper a novel model for the integration of data and knowledge bases, called the D/K model. It was described by presenting its structure; defining its constructs,

operations and modelling rules; and introducing its data/knowledge definition, manipulation and query operations.

In addition to introducing a new data/knowledge model, we believe that this paper has also made two important contributions to the integration of data and knowledge bases. First, the model here described was designed by applying a new method for the integration of data and knowledge bases (see [20]). Applying this method to the process of integrating data and knowledge was crucial for dealing with the complexity of the process and ensuring a more comprehensive conceptual integration. Second, the paper has shown a viable and elegant way of formalising a model. We considered here two aspects of the formalisation: the definition of the semantics of the constructs of the model and the formal definition of the static and behavioural properties of these constructs.

The meaning of each construct was presented using the notion of view of the world. A denotational relationship between the constructs of the model and the elements of this view was defined, in order to give an account of the meaning of each construct. The description of the view of the world is an important aspect that complements the definition of a model. This aspect is usually omitted in the definition of many data models and knowledge representation formalisms. The view and the denotational relationship, which explains what each construct denotes, are particularly useful for representing the application domain during the conceptual design of a data/knowledge base.

The formalisation of the constructs of the model was presented in terms of a widely used formal specification language, the Z Notation, which uses a model-based approach. The value of the formalisation effort is the precision and consistency achieved in the definition of each construct of the model. In our case, the most important product of this exercise was, however, the great deal of understanding of the object-oriented approach that we achieved by using a formal approach to the definition of the model.

The attempt to formalise the model is by no means complete. The formal definition of each operation of the model is still pending. Nevertheless, we believe that the formal specification of the constructs as given here is for the time being sufficient to understand the details of the model, as required by its future implementation.

Similar data/knowledge models, that were not influential in the design of the D/K Model, are OSAM [22], MPL/0 [22] and Postgres. OSAM is a semantic data model that supports object-oriented concepts. MPL/0 is a multiparadigm language designed to support the development of data/knowledge bases. It extends the OSAM model to support generic units, typing and behavioural associations. OSAM and MPL/0 use their own notations. Except for the ability to represent rules, the MPL/O and the D/K models have similar expressive power. D/K model features not provided by MPL/O are the redefinition of inherited attributes, constraints and methods and procedural attachment. Postgres, on the other hand, is an extension of the relational data model that incorporates object-oriented and rule-based concepts. The D/K model supports metaclasses, parameterized classes, class attributes, default and generic values, dependent and exclusive composition, and explicit constraints; which are features not supported by Postgres. The main difference between the D/K model and the aforementioned models is, however, the uniformity provided by the former. The D/K model uses the concepts and syntax of the ST-80 language, which is the most representative language of the object-oriented paradigm.

Similar attempts to formalise object-oriented data models are reported in the literature. The formal definitions given by C. Beeri [4], Y. Wand [24], and C. Lecluse, et al [15] are probably

the most relevant ones in the area. Our approach was inspired in the work of Y. Wand. We used the same ontological view of the world, which is proposed by M. Bunge [10], in order to explain the meaning of object-oriented constructs. The main difference between our approach and Wang's is that ours rests on the definition of the structural and behavioural properties of the constructs; whereas the latter concentrates on the denotational properties.

The D/K model is actually being used for developing a prototype of a data/knowledge base object manager and a library of specialized classes for supporting the development of data and knowledge bases in geographical and multimedia applications.

References

[1] Banerjee, J., Kim, W., Kim, H.-J, and Korth, H.F. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. ACM-SIGMOD Int. Conf. on Data Management*, 1987, pp. 311-323.

[2] Banks, B.J., Deen, S.M., Garcia, L.A., Harding, S.M., and Herath, A.C. Design and Implementation of Deal. In Deen, S.M. and Thomas, G.P. (Eds.) *Data and Knowledge Base Integration*, Pitman, London, 1990, pp. 29-62.

[3] Batini, C., Lenzereni, M., and Navathe, S.B. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, Vol.18, No.4, December, 1986, 323-364.

[4] Beeri, C. A formal approach to object-oriented databases. Data & Knowledge Engineering, Vol. 5, No. 4, 1990, pp.353-382.

[5] Beynon-Davies, P. *Expert Database Systems - A Gentle Introduction*. McGraw-Hill, London, 1991.

[6] Brachman, R.J. and Schmolze, J. An Overview of the KL-ONE Knowledge Representation System. In Mylopoulos, J. and Brodie, M.L. (eds.) *Artificial Intelligence & Databases*, Morgan Kauffmann, San Mateo, CA, 1989.

[7] Brodie, M.L., et al. Knowledge Base Management Systems: Discussions from the Working Group. In Kerschberg, L. (Ed.) *Expert Database Systems*, Proceedings from the First International Workshop, Benjamin Cummings, 1986, 19-33.

[8] Brodie, M.L. and Mylopoulos, J. Knowledge Bases and Databases: Semantic vs. Computational Theories of Information. In Ariav, G. and Clifford, J. (Eds.), New Directions for Database Systems, Ablex, NY, 1986, pp.186-218.

[9] Breu, R. *Algebraic Specification Techniques in Object Oriented Programming Environments*. Springer-Verlag, Lecture Notes in Computer Science No. 562, 1991.

[10] Bunge, M. *Treatise on Basic Philosophy, Vol. 3: Ontology I: The Furniture of the World*. Reidel, Boston, 1977.

[11] Fikes, R. and Kehler, T. The Role of Frame-Based Representation in Reasoning. *Comm. ACM*, Vol.28, No.9, September, 1985, 904-920.

[12] Goldberg, A. and Robson, A. *Smalltalk-80, The Language*. Addison-Wesley, 1989.

[13] Kim, W. Introduction to Object-Oriented Databases. *The MIT Press*, Massachussets, 1990.

[14] Lafue, G. and Smith, R. A Modular Toolkit for Knowledge Management. In Mylopoulos, J. and Brodie, M.L. (Eds), *Readings in Artificial Intelligence and Databases*, Morgan Kaufmann, California, 1989, pp. 592-598.

[15] Lecluse, C., Richard, P., and Velez, F. O2, an Object-Oriented Data Model. In Zdonik, S.B. and Maier, D. (Eds.) *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990, pp. 227-236.

[16] Minsky, M. A Framework for Representing Knowledge. In Brachman, R.J. and Levesque, H.J. (Eds.) *Readings in Knowledge Representation*, Morgan Kaufmann, 1885.

[17] Montilva, J.A. *A requirements definition for multimedia intelligent database systems*. Technical report. University of Leeds, School of Computer Studies, Leeds, UK, September,1990.

[18] Montilva, J.A. and Roberts, S.A. *A Methodological Framework for the Integration of Data, Information, and Knowledge Management*. Research Report No. 92-20, University of Leeds, School of Computer Studies, Leeds, UK, September, 1992.

[19] Montilva, J.A. and Roberts, S.A. *A method for the design of intelligent database models*. Research Report, University of Leeds, School of Computer Studies, Leeds, UK, January, 1993.

[20] Montilva, J.A. *An integration method applied to the design of a data/knowledge model for multimedia and spatial applications*. Ph.D. Thesis. University of Leeds, School of Computer Studies, Leeds, UK, January, 1993.

[21] Parsaye, K., Chignell, M. Khoshafian, S., and Wong, H. *Intelligent Databases: Object-Oriented, Deductive Hypermedia Technologies*, John Wiley and Sons, 1989.

[22] Shyy, Y.M. and Su, S.Y.W. MPL/0: a Multi-paradigm Language Facility for Data/Knowledge Base Programming. In Deen, S.M. and Thomas, G.P. (Eds.) *Data and Knowledge Base Integration*, Pitman, London, 1990, pp.63-83.

[23] Spivey, J.M. *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall, 1992.

[24] Wand, Y. A Proposal for a Formal Model of Objects. In Kim, W. and Lochovsky, F.H. (Eds.) *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989, pp.537-559