

**Database Integration using the
Open Object-Oriented Database System VODAK**

Wolfgang Klas, Peter Fankhauser, Peter Muth,
Thomas C. Rakow, E.J.Neuhold

GMD-IPSI

Integrated Publication and Information Systems Institute

Dolivostr. 15

D-64293 Darmstadt, GERMANY

email: {klas, fankhaus, muth, rakow, neuhold}@darmstadt.gmd.de

In Omran Bukhres, Ahmed K. Elmagarmid (Eds.):
Object Oriented Multidatabase Systems: A Solution for Advanced Applications.
Chapter 14. Prentice Hall, Englewood Cliffs, N.J., 1995

Contents

15 Database Integration Using the Open Object-Oriented Database System VODAK	1
15.1 Introduction	1
15.2 Database Integration Environment	5
15.2.1 System Architecture	6
15.2.2 VML—The Common Data Model	10
15.2.3 Transaction Management	25
15.3 Schema Integration	32
15.3.1 Common Data Modeling Primitives	33
15.3.2 Overcoming Structural Heterogeneities by Augmentation	34
15.3.3 Declarative Methodology for Schema Integration	37
15.3.4 Representation of Export and Integrated Schemas in VML	45
15.4 Conclusion	52

Database Integration Using the Open Object-Oriented Database System VODAK

Wolfgang Klas, Peter Fankhauser, Peter Muth,
Thomas C. Rakow, Erich J. Neuhold

GMD-IPSI

Integrated Publication and Information Systems Institute

Dolivostr. 15

D-64293 Darmstadt, GERMANY

email: {klas, fankhaus, muth, rakow, neuhold}@darmstadt.gmd.de

1.1 Introduction

Electronic information management systems are complex human-centered activities that produce and consume diverse kinds of information. Today many of these activities are supported by autonomous systems that employ different data management facilities with heterogeneous data models, for example, a relational model, a hierarchical model, an object-oriented data model, or—specifically valid for public databases—a dedicated file system with specialized retrieval and presentation functionality. In addition, the information is represented at different levels of detail, with mutual inconsistencies in structure, naming, scaling, and behavior. Much of this behavior is hidden in the implementation of the autonomous systems.

Advanced applications, like those in the field of cooperative authoring and publishing or telecommunication services and administration definitely, need *integrated* access to their underlying autonomous, heterogeneous information bases. These applications demand integrated processing because of the need for consistent management of complex interrelated data as well as the integrated exchange of the information produced and consumed by the many participants in an application. Such applications should be provided with individual, partially integrated, global views on the underlying resources.

There exist several approaches and projects that address *interoperability* or *integration* of information bases.[SL90, LMR90, BHP92, EN89] give good overviews

and present fundamental concepts, including the terminology of the different approaches, for example, multidatabase systems, multidatabase languages, and federated database systems. MRDSM [Lit85], OMNIBASE [REC⁺89], and CALIDA [JPSL⁺88] are projects that realized the integration of databases by multidatabase languages, but require skilled users, because a user still needs information about the distribution of data and about how to resolve semantic ambiguities and other typical problems that arise when integrating heterogeneous databases. SIRIUS-DELTA [LBE⁺82], DDTS [DL87], Mermaid [TBC⁺87], Multibase [LR82], and our approach taken in KODIM follow the federated database approach. The tools and techniques developed in KODIM ([SN90, KD90, FKN91, FN92, KN90, MGPF92a, MGPF92b]) for *semantic integration* assist *incremental integration* driven by actual information requests of end users and the *dynamic maintenance* of integrated schemas driven by external schema evolution. This approach tries to meet the requirements of realistic situations with a large number of external information bases (which —because of their autonomy— are subject only to locally controlled constant change), in which completely integrated views valid for all users can hardly be achieved with reasonable effort. In addition, many available information sources (e.g., online databases) do not even provide fine-grained, explicitly structured data as relational databases do. Thus, in KODIM we also develop tools for the *structural enrichment* of data from external information sources that do not provide any kind of schema [GF92, FX94].

Database integration steps in KODIM can be partitioned into two conceptual layers:

- (1) At a base layer, heterogeneous data models have to be mapped to a uniform data model (*syntactic transformation* phase). This requires the translation of manipulation languages and the transformation of diverse data formats as well as the connection of external database management systems or other systems providing external data.
- (2) On top of this bottom layer (i.e., on the basis of the uniform data model) *implicit structure and semantics* have to be made explicit, *inconsistencies in structure, naming, and scaling* have to be overcome, and *semantic interrelationships* between data have to be acquired in order to establish integrated views onto the external resources (*semantic integration* step).

Figure 15.1 shows the variety of transformations that data from diverse resources have to undergo in order to be integrated. KODIM uses the data model VML [KAN94] of the open, object-oriented database system VODAK as the canonical data model the external schemas are mapped into. To use an object-oriented data model as the canonical model is widely recognized to be a very promising choice for easier representation of external data models as well as for schema integration purposes (see [SL90, SN90, KD90, HW91]).

The syntactic transformation step provides for a syntactically uniform VODAK interface to the external information bases, describing their database schemas (in-

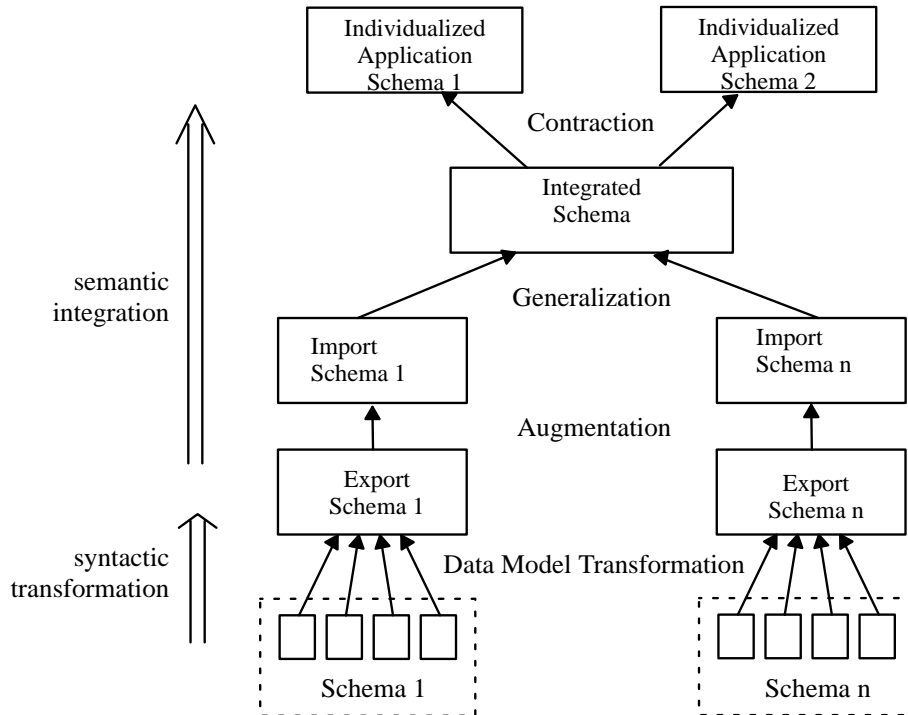


Figure 1.1. Framework for database interoperability.

cluding constraints), retrieval and manipulation capabilities, and file formats. Usually this includes *structural enrichment* of the external schemas or file formats. The transformation is modularized by means of the object-oriented VODAK Modelling Language (VML); that is, for all imported data (schemas as well as instances) object types and classes are established that support the functional capabilities of external information bases in a uniform language.

By syntactic transformation, external information bases are accessible according to a uniform data model, but they are not interrelated semantically. In this paper we do not focus on this syntactic transformation step, but we start with a syntactically uniform VODAK interface to the underlying external database systems. Details about the techniques we employ for the syntactic transformation step are given in [FX94, GF92, KFA94].

Based on the syntactic transformation an additional conceptual mapping step is required to interrelate and merge data semantically. Semantic integration is needed to combine the several schemas. Structural and semantic differences in representation and conflicts in naming and scaling have to be resolved, correspondences between objects have to be identified, and appropriate user views have to be determined in order to specify one (or several) integrated schemas and/or individual

application schemas. This obviously includes semantic enrichment, which makes implicit structure and semantics explicit [CS91] and associates additional behavior, which is hidden in local application programs or even worse in informal local conventions.

In this paper we focus on the *semantic integration* step. We aim at integrated schemas to which heterogeneous schemas are mapped to resolve differences between corresponding subschemas. However, we do not aim at a complete, global integrated schema, which overcomes all heterogeneities, but rather want to enhance the incremental design and maintenance of integrated schemas, according to the specific needs of an integrated application.

For this purpose we develop a *declarative methodology* for schema integration [CHS91, HR90, MNE88, SSG⁺91, SLCN88, WHW90], as opposed to the *procedural* approach of supporting a number of schema transformation operations to overcome inconsistencies [Mot87] or supporting just a multidatabase query language as advocated by the loosely coupled multidatabase approach [LA86, Lit85]. Users can declare correspondences between schema constituents (e.g., classes, types, attributes) that contain overlapping data according to their world view. To allow for the resolution of differences in representation, we consider — as [CHS91] and [SSG⁺91] — not only correspondences between constituents of equal kind, (e.g., between two classes from different schemas or between their direct attributes), but also correspondences between classes and attributes, and more importantly, between paths composed of several references between classes. The declared correspondences are checked for consistency, and from consistent correspondences, an integrated subschema is generated, together with mappings from the heterogeneous local corresponding subschemas. The novel feature of our methodology is that it is based on a flexible notion of schema unification that allows for *augmenting* the structural granularity of corresponding subschemas in order to overcome their heterogeneity. Thus, furthering the approaches of [CHS91] and [SSG⁺91], we can actually test correspondences between arbitrary paths for mutual consistency and turn them into fully transparent schemas *without* explicit correspondences. In addition, we develop concepts to assist the detection of possibly corresponding subschemas using fuzzy terminological knowledge about the application domain. Details about the techniques employed for the final semantic integration steps are also given in [SN90, NS88, SN88, FKN91, FN92, MGPF92a, MGPF92b].

In addition to solutions related to the various integration steps, an operational database system providing access to integrated views to external resources must also offer appropriate global transaction management. Such a transaction model, able to support the concurrent access of multiple users to integrated data, must address two main problems:

- First, it must provide for a high degree of concurrency compared to conventional models, because transactions in an integrated system (Figure 15.1) are relatively long-lasting and complex. Many complex system layers are involved until the data in the existing systems can finally be accessed. We solve this

problem by utilizing semantic information about methods available in the VODAK data model in the VODAK open nested transaction concept. Here, in a subtransaction hierarchy corresponding to the calling hierarchy of VODAK methods, this information is used for concurrency control and recovery.

- Second, the transaction model must integrate probably different concurrency control and recovery schemes of the existing systems into a single global transaction management. Autonomous local transactions executed by the existing systems without global control must still be allowed. The open nested approach allows us to integrate without changes the existing transaction management modules to manage the bottom level of the subtransaction hierarchy. The problem of autonomous local transactions is solved by considering sets of allowed autonomous local transactions together with sets of exported transactions available for global usage. By utilizing semantic information about the transactions in these sets and about the system architecture, we can handle autonomous local transactions without abandoning the conventional ACID transaction properties.

The rest of the paper is organized as follows. Section 15.2 introduces the system architecture of the database integration environment developed at GMD-IPSI and of the underlying database management system, which serves as the basis for the prototypical realization of the integrated database management system. It describes the general architecture, the common data model VML, and the transaction management services needed to incorporate the existing database systems. Section 15.3 describes the methodology and techniques used for the incremental integration of VML schemas. A graph-oriented notation for the data modeling primitives that are needed for the integration of schemas is introduced. Based on that, augmenting transformations are identified, which are employed to overcome structural heterogeneities between corresponding subschemas. In addition, we show how such augmenting transformations can be generated from correspondences between subschemas. An example illustrates how an integrated schema can be constructed in VODAK. Section 15.4 concludes the paper and gives some hints for further improvements.

1.2 Database Integration Environment

In this chapter we present the concepts of the database integration environment developed at GMD-IPSI. First, we discuss briefly the system architecture of the main constituents of the environment, that is, the *schema integrators workbench* and the underlying *open, object-oriented database management system VODAK*. Second, we present the *common data model*, which is used to describe the export schemas of the individual database systems and the integrated views. Third, we briefly discuss the *transaction model*, which allows for the execution of global transactions based on local transactions of the individual existing database management systems.

1.2.1 System Architecture

The database integration environment consists of two components. (1) The *schema integrators workbench* provides for the integration of export schemas and for the construction of integrated views. The export schemas have to be specified in VML. A schema integrator can use the workbench to generate VML specifications for integrated views. (2) The *VODAK database system* provides the common data model VML, a query language, a data dictionary, and transaction management for the execution of global transactions. VODAK sits beneath the schema integrators workbench. It provides database management services during the process of integrating individual databases and constitutes the DBMS, which allows for global access to integrated views by applications. In the following, we give an outline of the systems architecture of these two components of the database integration environment.

Schema Integrators Workbench

Figure 15.2 shows the general systems architecture of KODIM's schema integrators workbench with respect to its system components. It consists of four major modules:

- The graphical *Presentation and Manipulation* component supports the comfortable creation, modification, and deletion of VODAK schemas. It specifically allows for separate design of types and classes, offers a number of abstraction facilities (textual views, structural views, and roadmap) to present large and complex schemas at different levels of detail and allows for importing subschemas from multiple schemas.
- The *Design and Integration Assistant* contains algorithms for checking the consistency of newly designed schemas (acyclicity of inheritance hierarchies, consistent reuse of types in classes), for semantically enriching schemas, and for comparing and unifying preexisting, heterogeneous schemas. The semantic knowledge contains real-world knowledge modelled in the form of a fuzzy terminological network. It is used to semantically enrich pre-existing schemas and to achieve a better basis for comparison and unification of pre-existing schemas.
- The *Schema and View Editor* constitutes the kernel module. It provides the functions and data structures needed to manage the internal data processed by the *Presentation and Manipulation* component and the *Design and Integration Assistant*. In addition, all operations for manipulating the internal representation of schemas are defined here. *Semantic knowledge* is used via the *Design and Integration Assistant* to check the user actions and is made available for the presentation module whenever it is needed. All data exchange between the *Presentation and Manipulation* module, the *Design and Integration Assistant*, and the *VML-Interface* is coordinated in this module.

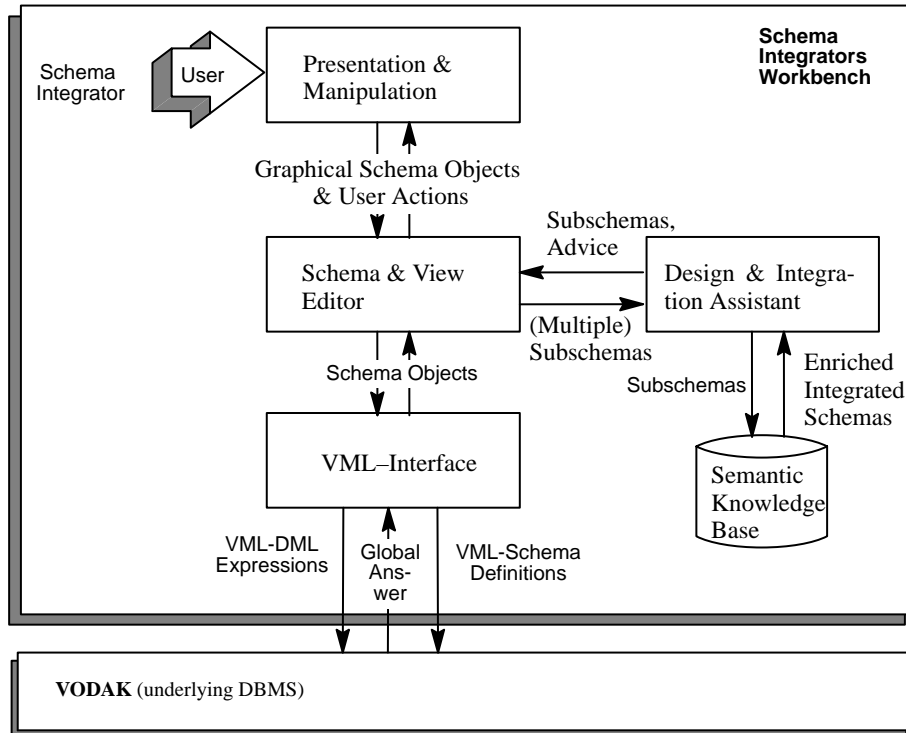


Figure 1.2. System architecture of the schema integrators workbench.

- The *VML-Interface* transforms the internal representation of schemas into VML specifications by stripping all editor and design-specific information and generating VML syntax from the data structures used by the *Schema and View Editor*. It uses the *VODAK VML compiler* to check the specified schemas. Error messages returned from the *VML compiler* are analyzed and passed on to the *Schema and View Editor*, which presents them to the user to assist him in the correction of his schema.

Section 15.3 gives a detailed discussion of the conceptual functionality of the schema integrators workbench according to the declarative integration methodology.

Integrated Database Management System

The modules needed for the global integrated data management for both the schema integrators workbench and for the applications accessing integrated views on heterogeneous databases are provided by VODAK. This includes a global transaction manager, communication manager, object manager, message handler, query processor, and compiler components. Figure 15.3 shows the architecture of the

database management system VODAK.

VODAK applications are written in VML, the *VODAK Modeling Language* (for details see Section 15.2.2), or in a C++-based application programming interface. All VODAK facilities and modeling concepts can be used by developing VML-schemas and application programs. The query and update language of VODAK is called VQL, *VODAK Query Language* [AF95]. VQL queries can be posed interactively by using the *VQL-Interpreter* or can be compiled by using the *VML-Compiler* [CTK94, CT93, CT94]. No specific run-time support is required for compiled queries. A C++ interface is available for direct usage of VODAK modules and for extending a VODAK schema by external methods.

VODAK is a distributed system, that is, the VODAK system modules *Object Manager*, *Transaction Manager*, *Communication Manager*, and *VQL-Executor* are distributed. The *Central Database Environment* controls the distributed executions. Even though the *Central Database Environment* could also be distributed, this has not been implemented in the prototype.

The *VML/VQL-Compiler* compiles VML-schema definitions, including VML-methods, and complete VML-programs into internal representations. The internal representations include entries of the data dictionary representing the schema, information about databases of the underlying database system, and C++ modules representing VML-methods. VML-methods might also include VQL-statements, which are compiled into executable C++ modules.

The *VQL-Interpreter* receives VQL-queries and updates from the *Schema Definition and Data Manipulation Monitor* or an user application. The *VQL-Interpreter* analyzes the queries and generates corresponding execution plans. These plans are passed to the *VQL-Executor*, which either executes the plans using the services provided by the object manager and the transaction manager or passes these plans to the *VQL-Executor* of an external database environment via the *Message Handler* and *Communication Manager*. The results are composed by the *VQL-Interpreter* and returned to the *Schema Integrator Workbench* or to the user. Compiled VQL-queries are treated as ordinary VML-programs with no specific run-time support of the VQL-executor.

Based on specific and efficient mapping mechanisms, the *Object Manager* in the internal database environment realizes the dynamic assembling and disassembling of objects of the underlying storage system to more complex structured VML-objects. It handles persistent and nonpersistent VML-objects and dynamically loads and stores objects from and to the underlying storage system. The object management modules in the external database environment map VML-objects to a specific representation of the external database by using specific *DB-Interface* modules. For instance, it realizes the mapping of relational tuples in SYBASE to VML-objects and the appropriate methods of the VML-objects to access functions for the attributes of those tuples. These mappings can be specified in VML [KFA94].

The *Message Handler* is responsible for exchanging messages between VML-objects. The message handler determines the receiver object of a message and the VML-method to execute. The *Object Manager* is called for this purpose. If

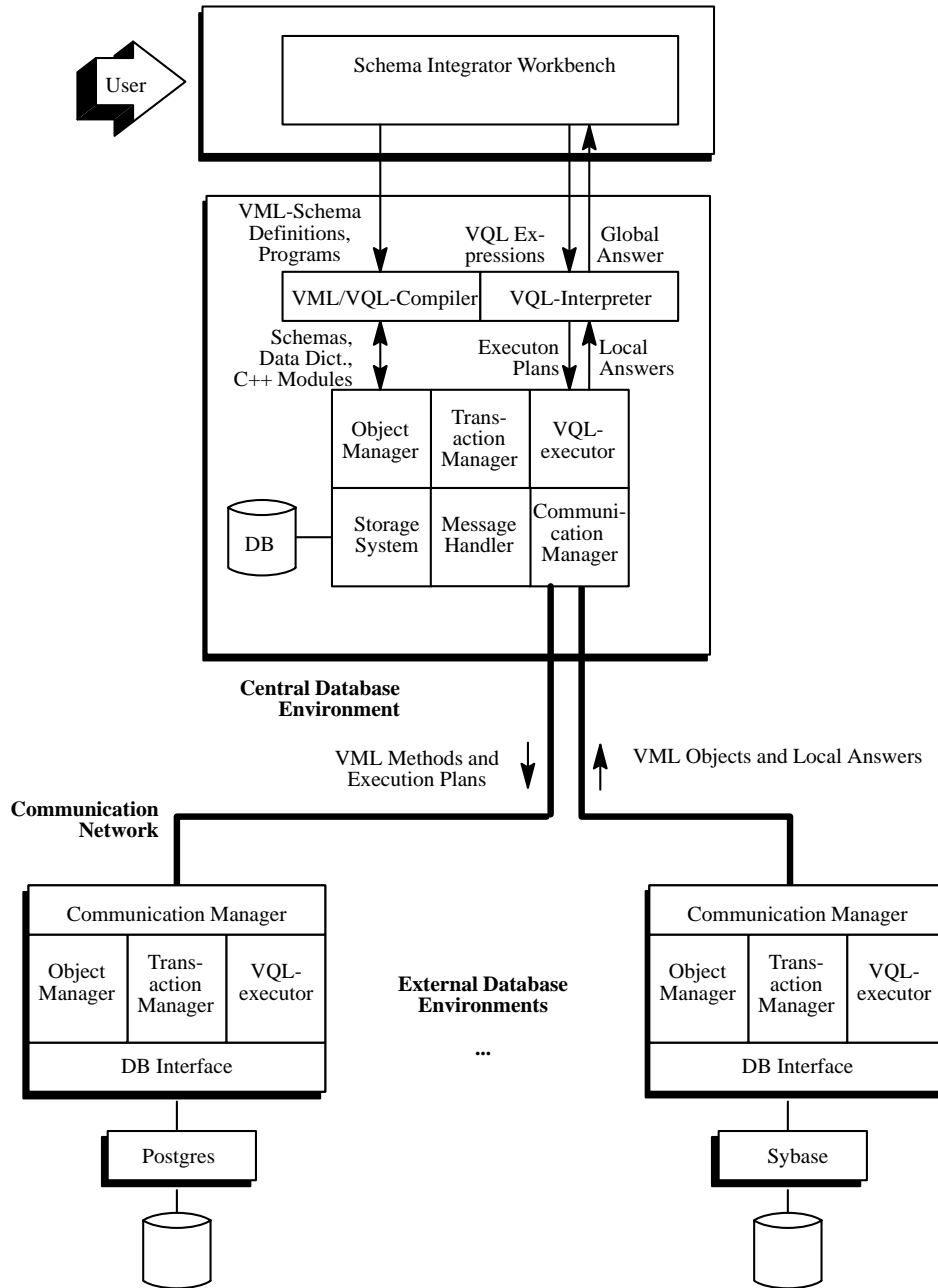


Figure 1.3. General system architecture.

the receiver object is located on a remote node, the *Message Handler* uses the *Communication Manager* to send the message to this node. Finally, the *Message Handler* calls the appropriate C++-function that implements the VML-method.

The *Transaction Manager* provides services for processing transactions. It considers the specific semantics (e.g., commutativity and dependency) of the methods called for the manipulation of objects. The *Transaction Manager* uses the transaction services provided by the underlying storage system and the external database systems. In the internal database environment, the *Transaction Manager* provides services for processing global transactions. The *Transaction Manager* in an external database environment handles the access to a single external database system by providing transaction mechanisms for the VML-objects that represent the external data. Details on the transaction management are given in Section 15.2.3.

1.2.2 VML—The Common Data Model

VML—the VODAK Modeling Language—is an *open object-oriented data model* that can be tailored to the needs of particular applications utilizing the concept of metaclasses (Figure 15.4).

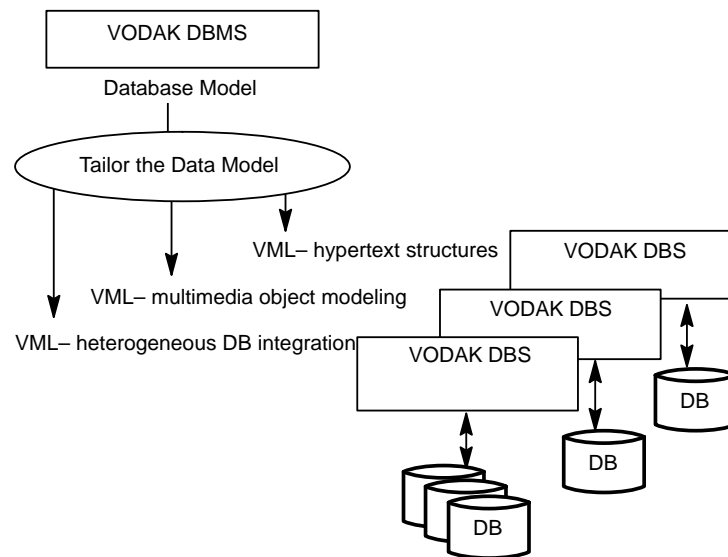


Figure 1.4. Tailoring VODAK to specific application requirements.

The general technique of tailoring VML is illustrated in [KAN94] following the application requirements for modeling hypermedia documents as they are used in the framework of, for example, collaborative authoring environments. Other examples for tailoring VML can be found in [AKF94, BA94, BAH94, HGPK94]. The same techniques are employed to describe the mappings of external schemas to ex-

port schemas specified in VML [KFA94]. Specific requirements and solutions for integrating multimedia data in VODAK are described in [AK93, KNS90b, RLMN93, TR93a, TR93b, AK94, TK95a, TK95b, RNL95, MKK95]. The same techniques are employed to adapt VODAK for the integration of heterogeneous database systems. In this section we present the main concepts of VML as far as we need them for the discussion of the database integration issues. We show how the model is tailored so that it provides the adequate modeling primitives needed to construct integrated schemas.

A comprehensive presentation of the VODAK Modeling Language can be found in [KAN94] and [GMD95]. A concise characterization of VML is as follows. VML distinguishes between *data values* and *objects*. Each data value is out of a *domain* determined by a *data type*. Data values themselves exist only *transiently* inside the scope of a program. An object is out of a domain constituted by a *class*. Objects exist *persistently*; that is, they exist independently of the scope of a program. Each object is assigned exactly one data value, called an *object identifier*, which is the unique identifier of the object by a data value through the object's lifetime.

All objects of a class share the description for their *interface*, structure of the object's *state*, and *implementation*. This description constitutes the type of an object and can be defined through *object type definitions*. The *interface* consists of a set of methods, more precisely a set of *method signatures*. The *state* of an object is constituted by a tuple of data values according to property definitions. Access to the object's state is possible only via the execution of the methods in the interface.

Class definitions determine the domains of classes. They refer to object types and define class objects that model the containers of sets of objects, that is, the *extensions* of classes. Class objects are instances of classes, which are called *meta-classes*. A *VML schema* consists of class definitions, data-type declarations, and object type declarations.

In the following we give more details on those language features that are needed to understand the realization of the integration methodology discussed in Section 15.3.

Data and Object Types

A data type characterizes a set of values (i.e., a domain) and a set of operations applicable to these values. Data types are built up from *primitive data types*, which include object identifiers and some basic *type constructors*.

```
datatype ::=
  primitive_type | structured_type | Data_Type_Identifier |
  Class_Identifier | ... not relevant here
```

The primitive data types are **BOOL**, **INT**, **REAL**, **STRING** and **OID**. The operators available for these data types are the usual ones. **OID** is the type of unique object identifiers. The only relevant operator in this context for **OID** is the infix operator `=`, which tests whether two object identifiers are equal or not.

Structured data types are built from primitive data types, already defined structured data types, object types, and type identifiers using type constructors. We will not discuss structured data types in detail because they are defined from other programming languages.

The structure and behavior of objects stored in the database are defined by object types. An object type declaration consists of a *public* and a *private* part, thereby separating the interface from the implementation. The interface declaration of an object type consists of a set of *structural property definitions* and a set of *behavioral method definitions*. The implementation definition provides implementations for all methods of the interface definition and may introduce additional structural and behavioral definitions. The values of the structural properties constitute the state of an object. The public methods are the means by which objects can communicate with each other. VML provides several alternatives for declaring object types, but we present only one of them:

```

object_type_declaration ::= object_complete_declaration

object_complete_declaration ::=
    OBJECTTYPE Identifier ["[" class_parameters "]" ]
    [subtype] ";"
    INTERFACE object_interface
    IMPLEMENTATION object_implementation
    END ";"

class_parameters ::= ... not relevant here

```

In the following we explain briefly how the interface and implementation sections of an object type can be defined. *Multiple inheritance* allows the build up of object type lattices for modular object type definitions.

```

subtype ::= SUBTYPEOF subtypes
subtypes ::=
    ObjectType_Identifier ["[" class_parameters "]" ] {" ," subtypes }

```

Next we give the object interface definition:

```

object_interface ::=
    [ PROPERTIES
        property_definition {property_definition} ]
        /* public properties */
    [ METHODS
        method_signature {method_signature} ]
        /* public methods */

property_definition ::= Identifier ":" datatype ";"

```

```
method_signature ::= ...
/* see Section 15.2.3, Specification in VML, for details */
```

In the interface only the signatures of the methods are specified. Updating the object (i.e., changing its internal state by changing the value of an object's property) is performed by calling methods (which may also delegate updates to other objects).

Next we give the object implementation definition:

```
object_implementation ::=
  [ datatype_declaration {datatype_declaration} ]
  /* local data types */
  [ PROPERTIES /* private properties */
    property_definition {property_definition} ]
  METHODS /* public and private methods */
  method_definition {method_definition}
  [ no_method_clause ]
  [ commuting_definition ]

method_definition ::=
  method_signature /* see Section 15.2.3 for details */
  method_body
  [ undo_definition ] ";"

method_body ::=
  "{" [variable_declarations] statement_list "}"

no_method_clause ::=
  NOMETHOD method_body
```

The private part of an object type declaration consists of the declaration of an object's private properties and the implementation of an object's methods. The optional data-type declarations introduce auxiliary types that are visible only in the implementation part. The implementation part may contain additional methods that are not defined in the public part. These are private methods that can be called only by other methods of the same object type. Furthermore, the implementation part can specify a **NOMETHOD** clause, which allows specification of a dynamic inheritance behavior, details of which will be explained in Section 15.2.2, *Determining the Structure and Behavior of Objects*.

Method implementations consist of the method heading (specifying the method name and its signature), and the method body. The body of a method is an optional list of variable declarations followed by a list of statements. Inside a method body the object's properties are accessed by their name only. The keyword **SELF** is used to denote the actual object. Methods can be specified as subtransactions (see Section 15.2.3). In this case an undo definition and a commuting definition

have to be specified. For more details on transactions in VML see Section 15.2.3, *Specification in VML*. We now give an example of an object type declaration.

Schema 1. Object type declaration.

```

OBJECTTYPE Person_Type
INTERFACE
METHODS
  city(): STRING;
  street(): STRING;
IMPLEMENTATION
METHODS
  city(): STRING; {RETURN SELF->getValue('city')};
  street(): STRING; {RETURN SELF->getValue('street')};
END

```

Objects, Classes, and Metaclasses

Objects represent material or immaterial real-world entities, or abstract concepts such as data-model primitives, that are stored persistently in the database. Objects that are similar in their structure and behavior are organized into *classes*. Every object is instance of exactly one class. The only way to operate on an object is by sending it a message that identifies a method defined for the object and arguments for that method.

Objects in VML obey the principle of *object identity*. Object identity is realized by *object identifiers*, which are attached to objects. Each object has its own distinct object identifier, which is different from that of any other object and which does not change throughout its lifetime. Object identity is the basis for sharing and updating objects.

A *class* describes the structure and the behavior of a collection of similar objects, called the *extension* of a class. The extension of a class is defined explicitly by creating instances of the class. The definitions needed to define the structure and behavior of instances of a class are specified with object types associated with the class. Because of the distinction of object types and classes, different classes can share the same object type. Details on the relationship between classes and object types are not discussed herein, but the reader is referred to [KAN94], which discusses the parameterization of object types in VML.

Instances of classes are stored persistently in the database and are shared by any application that is allowed to access the database schema and operate on the classes. VML distinguishes between *application classes* and *metaclasses*:

- *Application classes* are defined by an application developer to organize and classify the objects dealt with by the application. They correspond to application specific concepts like author, persons, documents.

- *Metaclasses* are defined by system administrators together with application developers in order to organize application classes and to tailor the database model to the specific needs of an application, for example, to meet the requirements of database integration. Metaclasses are used to describe the common structure and behavior of application classes *and* their instances. Every application class has exactly one associated metaclass. A metaclass may have many associated classes.

Classes (application classes and metaclasses) are treated as *first-class objects*; that is, a class *C* is an instance of its associated class *MC* (which is called the metaclass of the class *C*) as an object *c* is an instance of exactly one class (e.g., the class *C*). Such a class hierarchy can be of any depth. The root of the class hierarchy is the system predefined metaclass *VMLCLASS*. See [KAN94] for a description of this class. VML provides an initial class hierarchy containing a few predefined metaclasses by default. For more details on this initial metaclass system, see Section 15.2.2, *VML Class System for Database Integration*.

Treating classes as first class objects allows the user to operate on classes in the same way as he operates on individual instances of application classes. That is, classes can be retrieved and manipulated in the same manner as individual objects.

Specification of Application Classes To create an application class, one has to specify at least the name of the class with the **CLASS** clause and the properties and methods defined for the instances of the class, either directly with the **INSTTYPE** clause or by specifying the name of an object type that defines and implements these properties and methods. Such object types are called *instance types* of a class because they are used to determine the structure and behavior of instances of the class. The syntax for the definition of an application class is as follows:

```

application_class_definition ::=
    CLASS Class_Identifier [ METACLASS Class_Identifier]
        [ OWNTYPE object_type]
        INSTTYPE object_type
        [ INIT method_calls]
    END

object_type ::=
    ObjectType_Identifier | /* named object type */
    OBJECTTYPE [ subtypelist ";" ] /* anonymous object type */
        INTERFACE object_interface
        IMPLEMENTATION object_implementation
    END

subtypelist ::=
    SUBTYPEOF ObjectType_Identifier
        {"," ObjectType_Identifier }

```

Now we want to discuss in detail the four constituents of an application-class definition, the metaclass, the own type, instance type, and the initialization.

An application class is an object; hence, it must be defined as an instance of a metaclass. This is specified by the clause of the form **METACLASS** *Class_Identifier*. The object representing the application class is created as an instance of the metaclass identified by *Class_Identifier*. The metaclass can be chosen from a set of classes that contains predefined metaobjects like KERNEL-APPLICATION-CLASS for standard applications or user defined metaobjects provided by a metaclass library. If no metaclass is specified, KERNEL-APPLICATION-CLASS is taken as the default metaclass.

An application class may have an associated object type as its own type, which defines application specific properties and methods for the object representing the application class. For example, an application class may require a specific object-creation method that initializes the individual new instances of the application class, or it may require a sort method that sorts the individual objects with respect to a specific property. The own type of an application class is specified by the optional clause of the form **OWNTYPE** *object_type*. This can be done either by an in-place anonymous object type definition or by identifying the object type by name.

An object type is associated with every application class as its instance type, which defines the properties and methods for the individual objects collected by the class. The instance type of an application class is specified by the clause of the form **INSTTYPE** *object_type*.

The application-specific initialization of the object representing the application class can optionally be specified with the *init clause* **INIT method_calls**. It consists of the specification of a sequence of method calls that will be executed after the application class has been created as an instance of the associated metaclass during the database creation process.

Schema 2. Specification of class Person

```

CLASS Person METACLASS PG_MetaClass
INSTTYPE OBJECTTYPE
INTERFACE
METHODS    city(): STRING;
                street(): STRING;
IMPLEMENTATION
METHODS
    city(): STRING; {RETURN SELF->getValue('city')};
    street(): STRING; {RETURN SELF->getValue('street')};
END
INIT Person->init('Person')
    // attach to relation Person in the Postgres database
END

```

Specification of Metaclasses Metaclasses are used to describe the common structure and behavior of classes and their instances, which may not be known at the time a metaclass is defined. This means that the types and the classes of the objects representing the application classes and their instances are not known at the time the properties and methods provided by a metaclass are specified. The designer of a metaclass knows only that the properties and methods provided by the metaclass operate on objects. To overcome this problem, VML provides the primitive data type *object identifier* **OID**, which can be used to refer to an object whose class is not known.

The concept of metaclasses is homogeneously integrated with the other concepts of VML. Metaclasses and application classes are treated uniformly as classes. Hence, the definition of metaclasses follows the same rules as given for application classes. In addition, it is possible to specify the common structure and behavior for the instances of the instances of a metaclass by associating an object type as instance-instance type to the metaclass.

```
meta_class_definition ::=
  CLASS Class_Identifier METACLASS Class_Identifier
    [ OWNTYPE object_type ]
    INSTTYPE object_type
    INSTINSTTYPE object_type
    [ INIT method_calls ]
  END
```

In addition to the clauses of an application-class definition, an object type is associated with every metaclass as its *instance-instance type*. This object type defines the common properties and methods for the individual objects collected by the instances of the metaclass, which themselves are classes. The *instance-instance type* of a metaclass is specified by the clause of the form **INSTINSTTYPE** *object_type*. Examples for metaclasses will be given in Section 15.2.2, *VML Class System for Database Integration*.

Determining the Structure and Behavior of Objects

As previously mentioned, classes determine the structure and behavior of their instances. More precisely, an application class determines

- (1) the *application-specific* structure and behavior of its instances, which represent the real-world objects dealt with in an application program, for example, the content and position of a hypertext node, and
- (2) the *application-specific* structure and behavior of the application class itself, for example, a specific object creation and initialization method for nodes.

The specification of the structure and the behavior is given with object types that are associated with an application class. In (1), an appropriately defined object

type is associated with the application class as its *instance type* (as it affects the *instances* of the class). In (2), an appropriately defined object type is associated with the application class as its *own type* (as it affects the *own* structure and behavior of the class as an object).

In this respect, metaclasses are more powerful than application classes. They determine not only the structure and behavior of their instances, but also the structure and behavior of the instances of its instances. That is, a metaclass determines

- (1) the structure and the behavior of the metaclass itself, for example, a specific object creation and initialization method for classes,
- (2) the *common* structure and behavior of its instances which are classes (i.e., application classes or metaclasses), for example, the general object creation and initialization method for instances of an application class, which does not consider any application-specific requirements, and
- (3) the *common* structure and behavior of the instances of its instances (which are classes), for example, the common method that returns the class of an instance.

In (1), an appropriately defined object type is associated with the metaclass as its *own type* (as it affects the *own* structure and behavior of the metaclass as an object). In (2), an appropriately defined object type is associated with the metaclass as its *instance type* (as it affects the *instances* of the metaclass). In (3), an appropriately defined object type is associated with the metaclass as its *instance-instance type* (as it affects the *instances of the instances* of the metaclass).

Figure 15.5 shows this scheme: The *own type* of the metaclass *MC* affects the structure and behavior of the metaclass itself. The *instance type* of the metaclass *MC* affects the structure and behavior of the instances of the metaclass *MC* (i.e., the class *C*). The *instance-instance type* of the metaclass *MC* affects the structure and behavior of the instances of the class *C*. The *own type* of the class *C* affects the structure and behavior of the application class itself. The *instance type* of the class *C* affects the structure and behavior of the instances of the class *C*.

The *interface* defined for an object is the set of methods that are defined for the object, that is, that can be executed directly for the object. The interface depends on the definitions given with the object types associated with the object's class and with the class's metaclass. In general, the interface defined for any object consists of

- the methods specified with the *own type* of the object, if this object represents a class, and
- the methods specified with the *instance type* of the object's class, and
- the methods specified with the *instance-instance type* of the metaclass of the object's class.

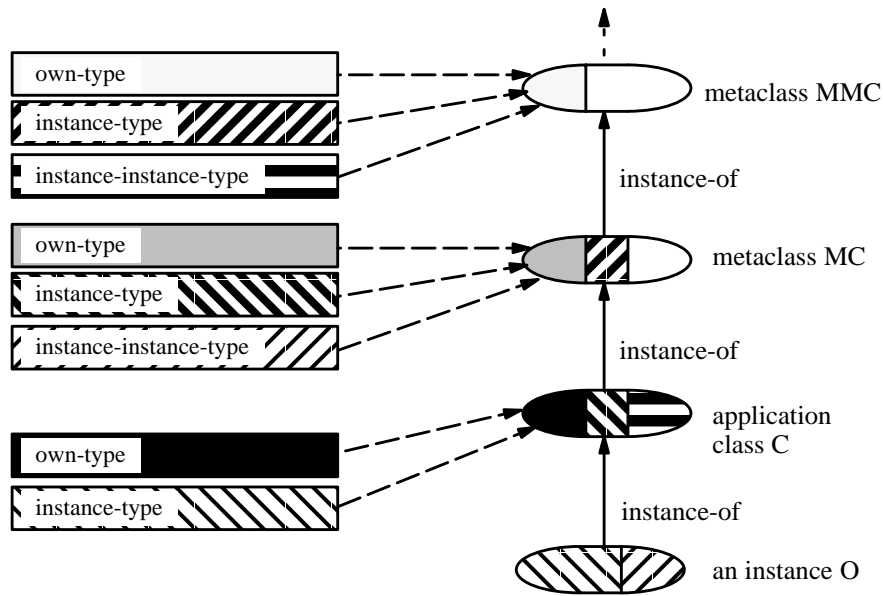


Figure 1.5. The general scheme of determining an object's structure and behavior through its class and the class's metaclass.

For any message that specifies a method contained in the interface defined for the receiving object, it is guaranteed that the method is executed for this object. A message is sent to an object by executing a statement of the form `receiver_object → m(arglist)`. In case the method is not contained in the interface defined for the object, the message-handling system of VML tries to delegate the message to another object by executing the method implementation given in the **NOMETHOD** clause. Within the body of the **NOMETHOD** clause, two predefined identifiers, `currentMethod` and `arguments`, are available. The identifier `currentMethod` is bound to the method name `m`, which has been sent to `receiver_object`. The identifier `arguments` is bound to the argument list of the message. If the **NOMETHOD** clause is not defined, a run-time error is signalled. The implementation of the body of the **NOMETHOD** clause has to be specified by the user. For example, different inheritance strategies can be implemented.

VML Class System for Database Integration

The class system in VML (see Figure 15.6) is logically partitioned into a set of system-internal classes and a set of system-external classes. The latter are organized into four levels:

1. the individual object level (Level 1),

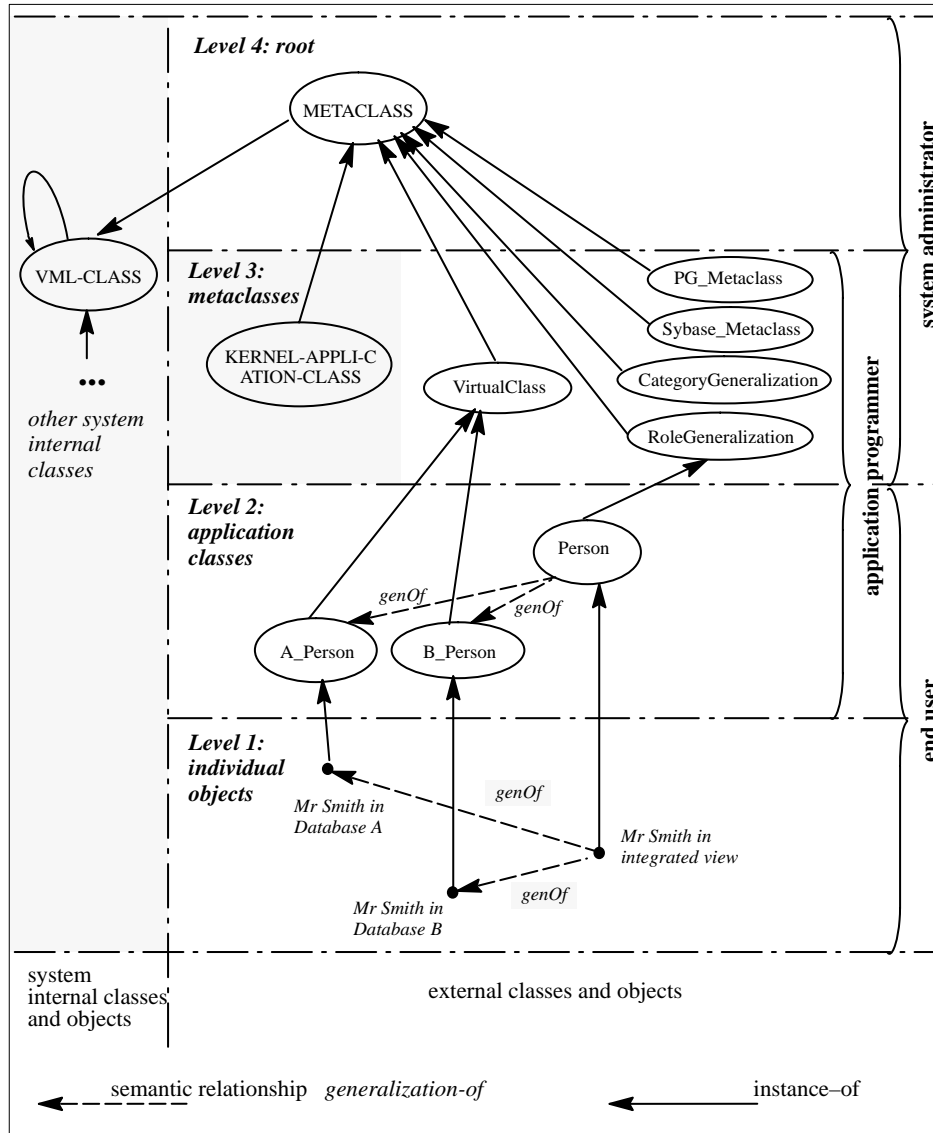


Figure 1.6. A sample VODAK class system for database integration.

2. the application class level (Level 2),
3. the metaclass level (Level 3), and
4. the root level (Level 4).

Presumably, an application user will see and use the bottom two levels. He will query, update, create, and remove individual objects of application classes. It is the task of an application developer, working on levels 2 and 3, to define new application classes as instances of some predefined metaclasses. At the metaclass level the system administrator may define new metaclasses and, thus, enhance the modeling capabilities of the predefined kernel model for, for example database integration.

The initial metaclass system of VML consists of `VMLCLASS` and `METACLASS` at the root level, `KERNEL-APPLICATION-CLASS` at the metaclass level, and a few other system-internal metaclasses, which cannot be changed or modified. They are built-in classes and provide for the basic and system inherent capabilities. Changing or extending the definitions for these classes will change the standard modeling capabilities of VML.

In Figure 15.6 the initial metaclass system is identified by the shadowed boxes. The system-internal classes provide the basic structures and behavior in order to deal with classes, objects, and the data dictionary of a database. The class `VMLCLASS` provides capabilities like object creation, object deletion, and object storage.

The class `METACLASS` serves as the root for any metaclass introduced and defined by a system administrator. The class `KERNEL-APPLICATION-CLASS` is the default metaclass for user-defined application classes. Their corresponding instance types and instance-instance types are derived from the instance type and instance-instance types of `VMLCLASS`, according to Figure 15.7. This figure illustrates the initial object type hierarchy. All object types in the system are organized by that type hierarchy. The root of this subtype hierarchy is the type θ . This type does not define any properties and methods. Every object type is either a direct subtype of θ (which is the default when no explicit supertype is specified), or it is a subtype of another object type; that is, it is an indirect subtype of θ .

Every database schema contains this initial metaclass system. All object types, application classes, and metaclasses are built up using this initial metaclass system. For a detailed description of the definitions of the initial metaclass system, see [KAN94].

To provide special functionality for the integration of existing heterogeneous database systems, we incorporate additional metaclasses. These metaclasses will be used in Section 15.3 to illustrate the realization of integrated schemas.

- (1) To provide functions to access an existing external database system from VODAK, we specify metaclasses that provide for the mapping of relational schemas to VODAK schemas. In this paper we use *PG_Metaclass* to access data stored in an extended relational Postgres database and *Sybase_Metaclass* to access data stored in a relational Sybase database. These metaclasses are used to define the classes constituting the export schemas for an external database in VML (see also [KFA94]). They provide methods like *getValue* to retrieve the value of an attribute in an external database and *init* to attach a VML class to an external relation. Several versions of such metaclasses, differing in the complexity of mappings they support, are available. See [KFA94]

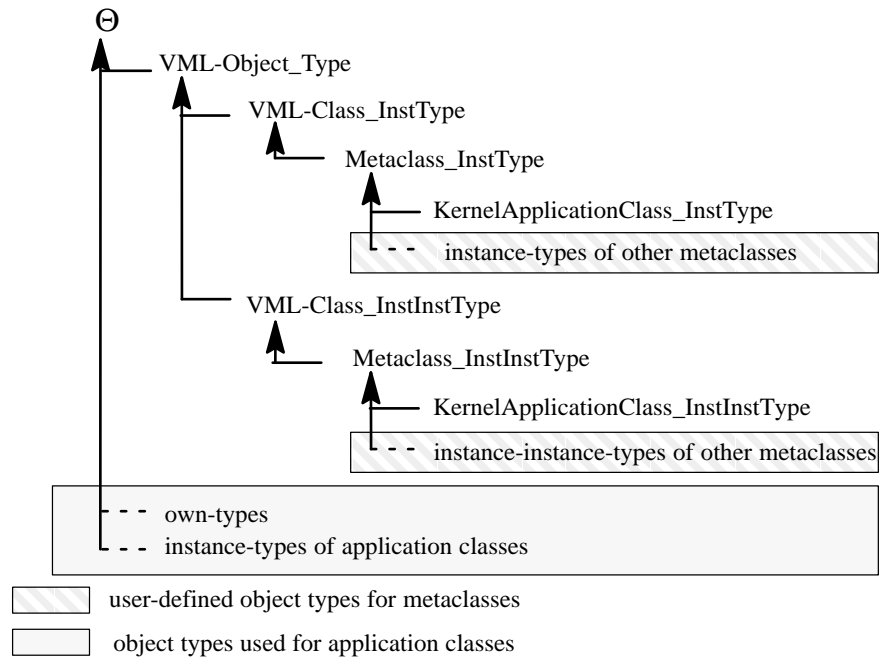


Figure 1.7. The initial subtype hierarchy.

for a discussion of these issues. Schema 3 shows the definition of *PG_Metaclass* and *Sybase_Metaclass*. For the discussion in this paper we can assume that the latter is defined like *PG_Metaclass*.

Schema 3. Specification of metaclasses *PG_Metaclass* and *Sybase_Metaclass*

```

SCHEMA ExternalDatabaseConnectionPrimitives
OBJECTTYPE PG_Metaclass_InstType
SUBTYPEOF VML-Class_InstType
INTERFACE
METHODS init(rel: STRING);
// other defs not relevant here, see [KFA94] for details
END
OBJECTTYPE PG_Metaclass_InstInstType
SUBTYPEOF VML-Class_InstInstType
INTERFACE
METHODS getValue(att: STRING): STRING;
// other defs not relevant here, see [KFA94] for details
END
CLASS PG_Metaclass METACLASS Metaclass
  
```

```

    OWNTYPE PG_MetaClass_OwnType
    INSTTYPE PG_MetaClass_InstType
    INSTINSTTYPE PG_MetaClass_InstInstType
END;
CLASS Sybase_MetaClass METACLASS MetaClass
    OWNTYPE Sybase_MetaClass_OwnType
    INSTTYPE Sybase_MetaClass_InstType
    INSTINSTTYPE Sybase_MetaClass_InstInstType
END;
END_SCHEMA

```

- (2) To support restructuring of import schemas in order to overcome structural heterogeneities we introduce the metaclass *VirtualClass*. It allows the definition of virtual classes derived from local classes specified in import schemas. *VirtualClass* provides the following methods:

- *virtualClassOf* allows assigning a class from an import schema to a virtual class.
- *localClass* returns the local class in an import schema assigned to a virtual class.
- *virtualOf* returns the instance of a local class for a virtual instance of a virtual class.
- *in* allows navigating from an instance of a virtual class to an instance of another virtual class.

Schema 4. Specification of metaclasses *VirtualClass*

```

SCHEMA DatabaseIntegrationPrimitives
OBJECTTYPE VirtualClass_InstType
SUBTYPEOF VML-Class_InstType
INTERFACE METHODS:
    localClass () : Oid;
    virtualClassOf(localclass: Oid);
IMPLEMENTATION
PROPERTIES:    virtualClassOf: Oid;
METHODS:
    localClass () : Oid;
    { RETURN virtualClassOf;}
    virtualClassOf(localclass: Oid);
    {virtualClassOf := localclass;}
END

OBJECTTYPE VirtualClass_InstInstType

```

```

SUBTYPEOF VML-Class_InstInstType
INTERFACE METHODS:
    virtualOf () : Oid; // returns id of local object
    setVirtualOf(object: Oid);
    in(virtualClass: Oid) : Oid;
IMPLEMENTATION
PROPERTIES: virtualOf: Oid;
METHODS:
    virtualOf () : Oid;          { RETURN virtualOf;}
    setVirtualOf(object: Oid);  {virtualOf := object;}
    in(virtualClass: Oid) : Oid
    { VAR s: {OID};
      virtualObject : OID; // defaults to NULL
      if ( SELF->class()->localClass ==
          virtualClass->localClass())
      then {
        s:= ACCESS v
            FROM v IN virtualClass->allInstances()
            WHERE v->virtualOf() ==
                  SELF->virtualOf();
        FORALL (virtualObject IN s) ;
      } // a single elem in s
      RETURN virtualObject;}
END

CLASS VirtualClass METACLASS Metaclass
INSTTYPE VirtualClass_InstType
INSTINSTTYPE VirtualClass_InstInstType
END;
END_SCHEMA

```

- (3) To support the definition of integrated classes, which integrate classes from different restructured export schemas, we introduce a set of additional modeling primitives by metaclasses. In this paper we refer only to two of these modeling primitives, *role generalization* and *category generalization*. They are provided by the metaclasses *RoleGeneralization* and *CategoryGeneralization*. These metaclasses provide for the necessary methods for integrating existing classes and for transforming to and from different representations of the same real-world object. More details are given in Section 15.3. A comprehensive description of the implementation of the metaclasses and their usage can be found in [Kla90] and [KNS90a].

It is important to note that these additional definitions of metaclasses, which tailor VODAK for database integration have been developed by model designers and

not by the end user, for example the schema integrator working with the Schema Integrators Workbench.

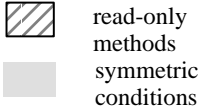
1.2.3 Transaction Management

One important research area in database integration is transaction management. If more than just read-only queries are considered, concurrent transaction executions have to be controlled by transaction management. This avoids inconsistencies due to invalid interleavings of concurrent transactions, transaction aborts, and system failures. The aim of the VODAK transaction management is twofold. First of all, by utilizing information about structure and semantics of operations, VODAK's open nested transactions provide for a high degree of concurrency compared to conventional flat transactions. This is crucial for the performance of transactions on integrated data, because the process of data integration creates very complex objects. Hence, transactions accessing and updating this data will touch many subobjects and will, therefore, be long-lasting compared to transactions in homogeneous systems. Secondly, the VODAK transaction management has to take the heterogeneity and autonomy of the integrated systems into account. VODAK's heterogeneous three-level transactions model this by explicitly considering the interface to the integrated systems, autonomous local transactions, and new criteria for conflict definitions between operations at the interface level of the integrated systems. VODAK's heterogeneous three-level transactions are implemented by using VODAK open nested transactions in the VODAK central database environment and in the VODAK external database environments, respectively. Compared to existing approaches for transaction management in heterogeneous systems (see [BGS92, Elm92] for an overview), our model provides for a higher degree of parallelism, and even more important, for a broader range of applicability in different application scenarios, because it can be parameterized for different requirements in terms of serializability.

Principles of Open Nested Transactions

In this section, we review the main principles of open nested transactions [BSW88, MRKN92, MRW⁺93, RGN90, WS92] (cf. also [Dav78, Gra81]). An open nested transaction is essentially a tree of method invocations, usually referred to as "action". The edges in the tree represent the caller-callee relationship between actions; that is, the children of a node in the transaction tree are invoked to implement the action that is associated with the node. We assume that there exists a semantic conflict test for each possible pair of method invocations. The conflict test for a pair of actions does usually not depend on the descendants of the actions, but refers only to the observable behavior (i.e., the specified semantics) of the two invoked methods. This property is a crucial prerequisite for a modular implementation of the proposed semantic concurrency control and for reasoning about its correctness in a modular manner. The most common form of conflict test is to look up a compatibility matrix. Such a compatibility matrix can take into

DATATYPE VML_SET	includes(f)	get()	insert(f)	remove(f)	set(w)
includes(e)	+	+	e ≠ f	e ≠ f	-
get()	+	+	-	-	-
insert(e)	e ≠ f	-	+	e ≠ f	-
remove(e)	e ≠ f	-	e ≠ f	+	-
set(v)	-	-	-	-	-



read-only methods
symmetric conditions

Figure 1.8. The compatibility matrix for the VML datatype **SET**.

account the actual input parameters of methods, return values, or the state of the accessed object [BBG⁺83, CRR91, LMWF92, O'N86, Wei88]. Figure 15.8 shows the compatibility matrix that is used for the VML datatype **SET**.

A concurrent execution of a set of open nested transactions is essentially a partial order of the actions of the executed transactions. A concurrent execution is called *semantically serializable* if it is equivalent to a serial execution of the transaction roots in the following sense [BBG89]: A serial execution can be stepwise constructed from the original execution by

1. exchanging the order of two adjacent, noninterleaving subtrees if their roots are commuting actions, and by
2. reducing an isolated subtree to its root, where a subtree is called isolated if all its descendants are serial and the entire subtree is ordered with respect to all other actions (i.e., not interleaved with other subtrees).

A locking protocol for open nested transactions proceeds as follows [MRW⁺93]. Each action of an open nested transaction must acquire a semantic lock, according to the invoked method. Thus, exploiting the semantics of the methods allows us to execute two compatible update methods on the same object concurrently, that is, without blocking. However, two conflict-free actions may still have conflicts among their descendants, that is, on the underlying implementation objects. The locks acquired by the descendants prevent unacceptable lower-level inconsistencies and ensure that higher-level actions appear as if they were indivisible. This means essentially that nonleaf nodes of the transaction tree are executed as subtransactions and that semantic serializability is guaranteed for each set of subtransactions, the

roots of which operate on the same object. The locks of the actions in a subtransaction are released upon the completion of the subtransaction, rather than being passed to the subtransaction's parent as in the conventional model of *closed* nested transactions [Mos85]. This means that subtransactions commit, that is, expose their effects, independently of the commit of their ancestors; hence the name *open* nested transactions. However, because the parent of a committed subtransaction holds a higher-level semantic lock, the effects of the subtransaction are visible only to actions that commute with the root of the subtransaction. This locking protocol ensures semantic serializability, that is, equivalence to a serial execution as viewed by the top-level actions of the executed transactions.

Figure 15.9 contains the concurrent execution of three open nested transactions. Transaction T_1 inserts elements e and h into set s . On the next abstraction level, counter c is incremented by 1, which results in *read* and *write* operations on the page level. Transaction T_2 concurrently tests whether element g is included in the set. Because these operations commute with the *insert* operations of T_1 (see commutativity table), both transactions can proceed in parallel. At the next lower level, a conflicting *read* is called. But because of the commuting operations accessing the set s , transaction T_2 can proceed after the commit of the subtransaction *insert*(s, e) called by T_1 . In the same manner, the other *insert* of T_1 can be executed in parallel with the *includes* operation of T_2 . The transaction T_3 reads the counter c directly without accessing first the set s (for performance reasons or because the counter is shared by another set). Thus, T_3 conflicts with transaction T_1 on the toplevel.

Open nested transactions gain concurrency, that is, reduce the number of lock conflicts, by exploiting the semantics of methods at all levels of the transaction trees, and by committing subtransactions early. The consequence of the early commit is that transaction aborts (and the undo phase of crash recovery) can no longer be implemented by restoring the pre-transaction state of the modified storage-level objects (i.e., objects modified by the leaves of a transaction tree). Rather, committed subtransactions must be compensated by means of appropriate "inverse" methods. The necessary compensating subtransactions are again subject to the semantic concurrency control protocol sketched above. Recovery issues for open nested transactions and the special case of multilevel transactions are further discussed in [BSW88, GM83, HW91, KLS90, MGG86, WHBM90, Wei91].

Heterogeneous Three-Level Transactions

VODAK heterogeneous three-level transactions make the benefits of the open nested transaction model available to heterogeneous and autonomous systems. We have to consider two problems. The first is the autonomy of the integrated systems, preventing us from simply using concepts developed for homogeneous distributed systems. The second is the performance, measured in the potential degree of parallelism between transactions. We developed a heterogeneous three-level transaction model, which can be parameterized to different application requirements and different degrees of autonomy by utilizing the semantics of methods like open nested

transactions. In contrast to other approaches, we explicitly consider the interface to the integrated systems in terms of exported local transaction programs instead of simple read/write operations. Exported local transaction programs and autonomous local transactions are assumed to always ensure local consistency constraints if executed in a locally serializable way. The only requirement for existing systems to be integrated is to provide ACID transaction properties [HR83].

Heterogeneous three-level transactions consist of three levels of transactions. Level L_2 consists of executions of methods in the central VODAK database environment. Transactions on this level are treated as open nested transactions. If a method in an external VODAK database environment is called, it will be executed as a transaction at level L_1 . Again, these transactions are handled as open nested transactions. Eventually, operations of integrated systems (i.e. transaction programs) are called. They are executed by the integrated systems and represent the L_0 level in our model. We assume that the integrated systems ensure serializability of L_0 transactions. There is no limit on the number of subtransactions per global transaction and integrated system. Figure 15.10 gives an example. For the sake of simplicity, we do not represent the nesting of the open nested transactions at levels L_2 and L_1 . Instead, we treat open nested transactions at these levels as single actions in the heterogeneous three-level model [Mut94].

TG_1 and TG_2 represent global transactions transferring money between bank accounts at banks $bank_1$ and $bank_2$. They execute transaction programs $trans$ exported by both banks for global usage. TL_3 and TL_4 represent autonomous local transactions as the execution of transaction programs at level L_1 : TL_3 transfers money between local accounts of $bank_1$, TL_4 reads the value of an account at $bank_2$. Consider the schedule at level L_1 . This schedule is not read/write serializable, because in $bank_1$ TG_1 is executed before TG_2 , and in $bank_2$ TG_2 is executed before TG_1 and both update the same data.

Even though the execution shown in Figure 15.10 will not cause any inconsistencies of data or return values. Let us first consider the schedule at $bank_1$. All transactions in the schedule just transfer money between bank accounts. If there is no upper or lower limit on the accounts, all L_1 actions commute. Because the

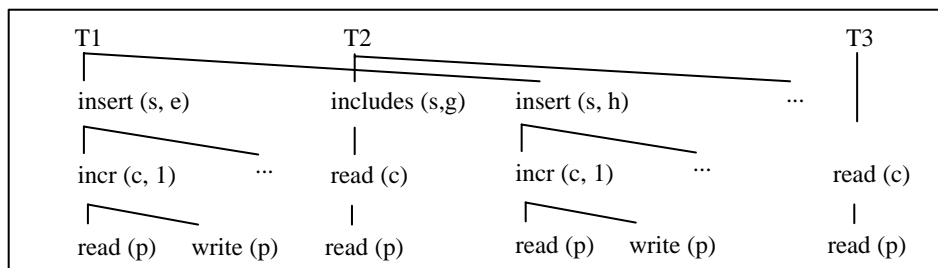


Figure 1.9. Concurrent execution of two open nested transactions.

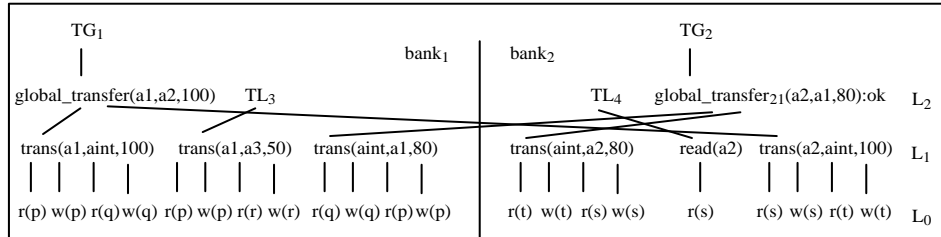


Figure 1.10. Transactions in the heterogeneous three-level model.

corresponding L_0 transactions are serialized by the integrated systems (actually executed serial in the example), the L_1 schedules at both banks are serializable. In addition, the L_1 schedule at *bank1* does not even define an execution order between TG_1 and TG_2 . The situation at *bank2* is different. The *read* action of TL_4 at level L_1 introduces an indirect conflict ([DEL⁺89]) between TG_2 and TG_1 . TG_2 and TG_1 do not directly conflict, but because *read* and *trans* do not commute, the resulting serialization order is TG_2 before TG_1 .

In general, it is impossible for the VODAK transaction management on either the central VODAK database environment or on the external database environments to detect indirect conflicts involving autonomous local transactions. The solution is to make sure that at level L_1 no conflict between actions of global transactions and local transactions are possible. This can only be ensured by disallowing the execution of such global actions or local transactions completely. The sets of allowed global actions and local transactions are parameters of the heterogeneous three-level transaction model. Other systems ensure a similar property by partitioning the data into locally and globally updateable data items [BST90], [MRB⁺92], that is, local and global transactions must not update the same data. The schedule of Figure 15.10 at *bank1* shows that by utilizing then semantics of methods, this requirement is too restrictive. TG_1 , TG_2 and TL_3 update the same data without causing any inconsistent data or return values.

On the other hand, considering the property of commutativity of methods is also not sufficient. TL_4 does not update any data, and the return value is correct from a local point of view for both execution orders of TG_1 and TG_2 , because L_0 serializability is sufficient to ensure consistent results of autonomous local transactions. Global transactions are not affected by reads of autonomous local transactions as well. As a consequence, there is no need to define a conflict between L_1 actions of global transactions and read operations of autonomous local transactions. TL_4 should be allowed to be executed. In the heterogeneous three-level model, we are able to model this by defining less restrictive conflict criteria than commutativity. This enables us to allow more types of global L_1 actions and autonomous local transactions to be executed.

According to architectural properties of integrated systems like VODAK (e.g. autonomous local transactions access and update data in a single existing system

only), we define two new conflict criteria: global and local commutativity. Global commutativity requires only two methods to commute from a global point of view, such as *trans* of TG_1 and TG_2 , and *read* of TL_4 . Local commutativity requires two methods to commute from a local point of view. It can be shown that both conflict criteria ensure that no global or local consistency constraint will be violated and no return value will be inconsistent in the heterogeneous three-level model.

If applications explicitly abandon consistency constraints, the conflict criteria can be even less restrictive. As the heterogeneous three-level model can be parameterized by giving different conflict criteria, it can also be used for these scenarios.

Recovery is handled by executing inverse actions to compensate changes of transactions that are to be aborted. Inverse actions are required for level L_1 and L_2 , because L_0 recovery is handled by the integrated systems as part of their transaction management. The execution of inverse actions is again controlled by concurrency control and recovery of VODAK. This allows us to treat concurrency control and recovery together in a homogeneous way.

Because L_2 and L_1 transactions are executed as open nested transactions in the central VODAK database environment and the local VODAK database environment, respectively, heterogeneous three-level transactions use concurrency control and recovery of VODAK as described in the previous subsection, *Principles of Open Nested Transactions*. Besides communication facilities provided by the VODAK communication manager, the only additional requirement for implementing a heterogeneous three-level transaction management is an atomic commitment protocol. Because we use inverse actions for recovery, there is no need to require the integrated systems to support a kind of two-phase commit [MR91]. We require only a function to detect the outcome of an L_0 transaction after a system crash.

By using open nested and heterogeneous three-level transactions, VODAK provides users with full-fledged transaction support for transactions that spawn several integrated systems. Both criteria, high degree of parallelism and applicability for different kinds of application scenarios, are fulfilled by (1) utilizing the semantics of methods and (2) parameterizability of the model by allowing/disallowing different types of global L_1 actions and autonomous local transactions and by using different conflict criteria at level L_1 .

Specification in VML

VODAK transactions are available via the system internal functions for accessing VML datatypes and via statements specifying commutativity and undo information for object types [MR93, GMD95].

- A *transaction* is a list of VML statements that will be executed in isolation from other transactions completely or not at all. Database objects can be only accessed by statements of a transaction. A specific statement can be used to program the abort of a transaction; that is, all the effects to database objects of already executed statements are undone.

```

transaction_statement ::=
    BEGIN_TRANSACTION
    statement
    END_TRANSACTION

```

```

abort_statement ::=
    ABORT_TRANSACTION

```

The schema programmer *can* specify methods as subtransactions. Subtransactions must be defined together with a specification of commutativity of subtransactions and how to undo updating subtransactions.

- A *subtransaction* can be only specified as a complete method and not as a part of a method. The specification of a method as a subtransaction is independent from the specification of other methods for the same object type, see also Section 15.2.2, *Data and Object Types*.

```

method_signature ::=
    [ SUBTRANSACTION ]
    method-identifier "(" [formal_parameter_list] ")"
    [ ":" return-datatype ] [ READONLY ] ";"

```

- A *commutativity definition* of an object type defines which subtransactions commute with each other under which conditions. The commutativity table is defined for the method of object types (see also Section 15.2.2, *Data and Object Types*) because objects as class members can be accessed only via the methods of the object types used by a class. No interference of methods of different object types for an instance is possible because their sets of properties are disjunctive. Commuting methods are specified as pairs of regular method calls but with a list of *formal* parameters. Two properties of commutativity are taken to implicitly define commutativity. (1) Read-only methods (as defined in the schema) commute with each other. (2) Commutativity is symmetric; that is, if method *a* commutes with method *b*, then *b* commutes also with *a*. If no return values are used in the commutativity specification, the specification of every commuting method pair is applied in the reverse order also. Using a return value (i.e., [**RETURNS** *Identifier*] is used) results in one entry of the corresponding commutativity table.

```

commuting_definition ::=
    COMMUTING
    { commuting_method_pair { "," commuting_method_pair }
    [ boolean-expression_method_body ] ";" }

```

```

commuting_method_pair ::=
    method_call_with_identifiers [ RETURNS Identifier ] ":"
    method_call_with_identifiers

method_call_with_identifiers ::=
    method_Identifier "(" [ Identifier { "," Identifier } ] ")"

```

- An *undo definition* of a subtransaction specified within a method definition (see also Section 15.2.2, *Data and Object Types*) defines which other method undos the effect of the subtransaction. The model of open nested transactions allows *parallel updaters* on the same object. Hence, the application programmer must specify a method to undo updating subtransactions. The undo methods must fulfill the following property. After the execution of a method and the corresponding undo statements, the following must hold:

1. No other method can see the execution of the do and the undo methods.
2. The effect of other methods that can be executed concurrently must not be destroyed by the undo method.

In some special cases the undo method relies on the do method. For example, the undo of deleting an object is inserting the deleted object. Therefore, the insert method must know the deleted object. In general, some information must be saved *before* the do method is executed. An optional statement is available for this case. The information can be stored in variables accessible by the both method bodies of the undo clause and the method call. An optional *after* statement provides access to data modified by the do method and to the return values of the method.

```

undo_definition ::=  UNDO
                    [ variable_declaration_list ]
                    [ RETURNS Identifier ]
                    [ BEFORE Before_method_body ]
                    [ AFTER After_method_body ]
                    INVERSE method_call_with_identifiers

```

1.3 Schema Integration

Independently designed databases typically maintain overlapping information with differences in naming, granularity, degree of abstraction, and scaling. To meaningfully exchange data between them and to provide integrated access to them, these differences have to be resolved. With complex schemas, this cannot be achieved realistically by implementing ad hoc mappings between corresponding subschemas as advocated by the multidatabase approach, but requires a methodology to assist the design and maintenance of such mappings.

We aim at integrated schemas to which heterogeneous schemas are mapped to resolve differences between corresponding subschemas. However, we do not aim at a complete, global integrated schema, which overcomes all heterogeneities, but rather want to assist the incremental design and maintenance of integrated schemas, according to the specific needs of an integrated application. For this purpose we develop a *declarative methodology* for schema integration. Users can declare correspondences between schema constituents that contain overlapping instances according to their world view. To allow for the resolution of differences in representation, we consider not only correspondences between constituents of equal kind, like between two classes from different schemas, or between their direct attributes, but also for correspondences between classes and attributes, and more importantly, between paths composed of several references between classes. The declared correspondences are checked for consistency, and from consistent correspondences an integrated subschema is generated together with mappings from the heterogeneous corresponding local subschemas. The methodology is based on a flexible notion of schema unification that allows for *augmenting* the structural granularity of corresponding subschemas in order to overcome their heterogeneity. In addition, we develop concepts to assist the detection of possibly corresponding subschemas using fuzzy terminological knowledge about the application domain.

The remainder of this section is organized as follows. In Section 15.3.1 we will introduce a graph-oriented notation for the data modeling primitives that we consider for integration. In Section 15.3.2 we will introduce *augmenting transformations*, which we use to overcome structural heterogeneities between corresponding subschemas. In Section 15.3.3 we will illustrate how these augmenting transformations can be generated from correspondences between paths. In Section 15.3.4 we will exemplify, how integrated schemas are realized in VODAK.

1.3.1 Common Data Modeling Primitives

In this section we distinguish between the following kinds of classes: kernel-application classes, virtual classes, category-generalization classes, and role-generalization classes. Kernel-application classes maintain their instances directly, whereas virtual classes derive their instances from instances of other classes (see Section 15.3.4). Category-generalization classes form the *disjoint* (not necessarily covering) union of the instances of their specialization classes, whereas role-generalization classes form the possibly *overlapping* union of their specialized classes. The class behavior of these different kinds of classes is realized by metaclasses (recall Section 15.2.2).

We use the following graphical notation for the data model covered by the integration methodology (recall Section 15.2.2 on VML). Kernel-application classes and virtual classes are represented by labeled \circ , category-generalization classes are represented by labeled \oplus , and role-generalization classes are represented by labeled \odot . By convention all class labels start with a capital letter. Generalization relationships are depicted by \Leftarrow . Single-valued properties connecting classes are represented by labeled \leftarrow , and set-valued references are represented by \leftarrow with two heads. Prop-

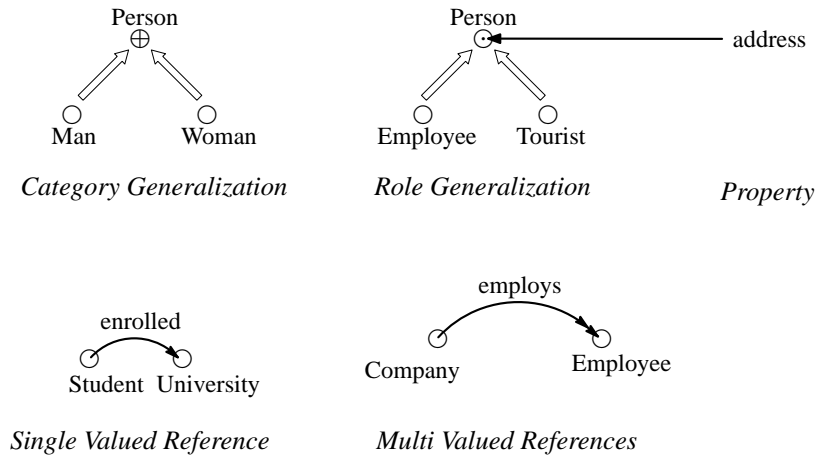


Figure 1.11. The modeling constructs.

erties are represented by their property name and are associated to classes by \leftarrow . Because the focus of the integration methodology lies on the schema level, we do not consider the type of a property. Figure 15.11 gives some examples of the graphical notation.

A schema is thus represented by a graph $G(V, E, label)$, where the set of vertices $V = V_{\odot} \uplus V_{\oplus} \uplus V_{\circ} \uplus V_A$, \uplus denoting disjoint union, and the set of edges $E = E_S \uplus E_R \uplus E_M \uplus E_P$, E_S denoting the set of generalization relationships, with E_R denoting the set of single-valued references, E_M denoting the set of multivalued references, and E_A denoting the set of properties, and $label$ is a labeling function that associates to each vertex in V and to each edge in $E_S \uplus E_R \uplus E_M$, a unique label.

1.3.2 Overcoming Structural Heterogeneities by Augmentation

The goal of schema integration can be stated as follows: Given a number of heterogeneous local schemas design, an integrated schema in such a way that each local schema can be considered a view of it [Bre90]. In particular, corresponding subschemas that maintain overlapping instances should be represented homogeneously in the integrated schema. For this purpose corresponding subschemas usually have to be restructured, because they exhibit differences in structural granularity and in degree of abstraction. For example, a path along class-valued properties in one schema may be represented as a direct reference between two classes in another schema, or a generalization hierarchy in one schema may be differentiated into more levels in another schema.

To cope with these differences, we need a notion of *view*, which comprises not only projections onto subgraphs of a schema and restrictions on the instance graph, but also *contractions* of a schema. *Contractions* restructure a schema by discarding

a class and combining the references to that class with the references going out from the class into direct references. A sequence of contractions thus transforms a path into a single edge.

In the framework of schema integration the, pre-existing heterogeneous schemas are the *views*. Thus, in order to resolve differences between corresponding subschemas we have to perform the inverse of contractions, *augmentations*, which introduce additional classes into a local subschema, to adapt it to a corresponding subschema, which models overlapping instances at a finer grain. We distinguish between four groups of augmentations:

1. *Augmentations that use independent properties of a class to generate roles of the class:* Such a transformation partitions the set of references and properties of a class disjointedly, but does not categorize them. As a consequence, when mapping subschemas to each other, a reference or property may be sought only in *one role specialization*, but may be categorized and associated to *more than one category specialization*. For example, in Figure 15.12 the (possibly null-valued) properties *job* and *favoriteResort* are used to distinguish *Persons* into explicit roles *Employee*, who all have a *job*, and *Tourist*.

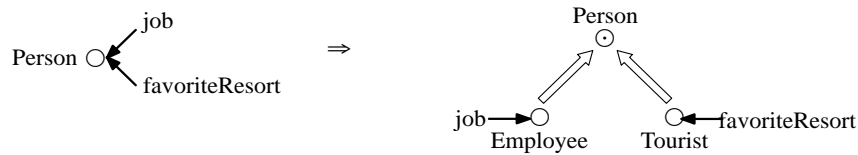


Figure 1.12. Substantiation of role properties.

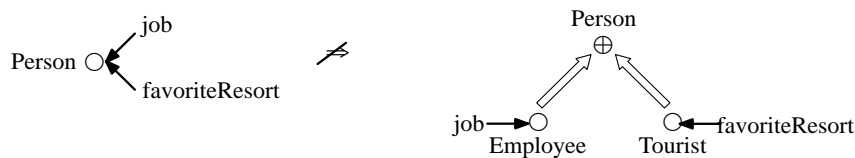


Figure 1.13. Inconsistent transformation.

Two independent references or properties, however, cannot be reasonably distributed among category specializations. For example (Figure 15.13), because of the disjointedness of *Employee* and *Tourist* at the right-hand side, there exists no *Person* that has a *Job* as well as a *favorite Resort* as required by the schema in the left-hand side. Thus an integrated schema on top of these schemas, which identifies *Person.job* with *Person.Employee.job* as well as *Person.favoriteResort* with *Person.Tourist.favoriteResort*, cannot contain any instances.

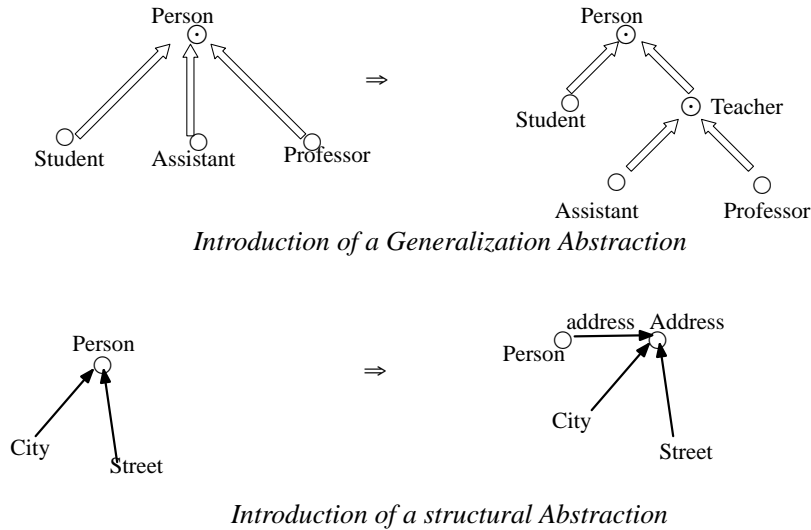


Figure 1.14. Introduction of abstractions.

2. *Augmentations that introduce additional abstractions:* this comprises the introduction of an intermediate generalization abstraction, as well as the introduction of an intermediate class for a reference or a property (Figure 15.14).
3. *Augmentations that use a categorizing property of a class to generate categories of the class:* For example, *Persons* can be distinguished explicitly into *CivilServants* and *(Industry)Employees* on the basis of the categorizing property *Employer* (Figure 15.15).

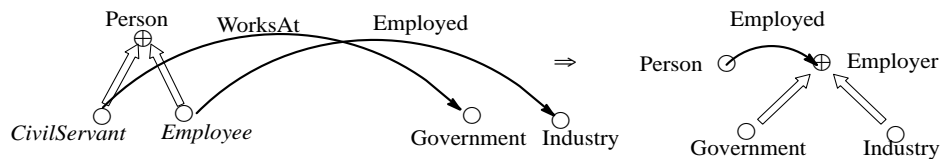


Figure 1.15. Substantiation of categorial properties.

4. *Augmentations that introduce a category generalization* for multivalued properties, which have a meaningful abstraction. For example, in Figure 15.16 the single-valued reference *staff* from *Project* to *Participants*, which in turn has multivalued references to *Internal Participants* and to *External Participants*, is transformed into a multivalued reference *staff* to *Participants*, which is now a category-generalization of *Internal Participants*, and *External Participants*. Whereas in the first modeling, the distinction between *Internal Participants*

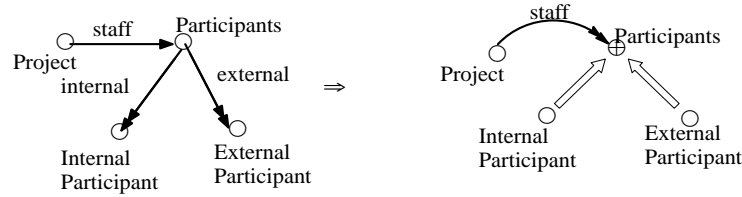


Figure 1.16. Grouping of categories vs. reference to classes with multivalued references.

and *External Participants* is achieved by references, there is an explicit abstraction of them, *Participants*, available in the second modeling, which can, for example, be used to attach common properties and methods.

On the basis of these *augmentations*, we can define the goal of schema integration as follows: *Identify* corresponding subschemas and adapt them to each other by applying *augmenting* transformations so that they become isomorphic. The augmented subschemas then constitute the integrated schema.

1.3.3 Declarative Methodology for Schema Integration

Using the augmentations as introduced in the previous subsection directly would amount in a procedural approach to schema integration. With such an approach users have to comprehend complex schemas in order to determine possibly corresponding subschemas and to intellectually choose appropriate augmentations to make corresponding subschemas isomorphic. In this section we devise a declarative methodology to support the automatic generation of augmenting transformations on the basis of correspondences between schema constituents declared by the user.

Figure 15.17 shows the main components we develop to support such a methodology. By means of a graphical schema editor for multiple schemas and with the help of conceptual background knowledge, to cope with naming inconsistencies [FKN91] users first identify vertices from different schema graphs that possibly contain overlapping instances. Thus, for two schema graphs $G_1(V_1, E_1), G_2(V_2, E_2)$, as introduced in Section 15.3.1, initially an integrated graph $G(V \uplus (V_1 - V/G_1) \uplus (V_2 - V/G_2), E_1 \uplus E_2)$ is formed where V contains all pairs of user-identified vertices, and V/G_i denotes the set of vertex pairs in V projected on vertices in G_i , and E_1 and E_2 are not integrated at all.

Then possibly corresponding paths are determined. We use two approaches to assist users in detecting corresponding paths. For resolving the special case of an inconsistency, in which an edge between already identified vertices in one schema corresponds to a path in another schema, we use a variation of the well-known algorithm by Dijkstra for determining the shortest path between two vertices in a weighted graph [MGPF92a]. To assist the detection of more general inconsistencies, we enrich classes with their immediate attributes and relationships semantically by

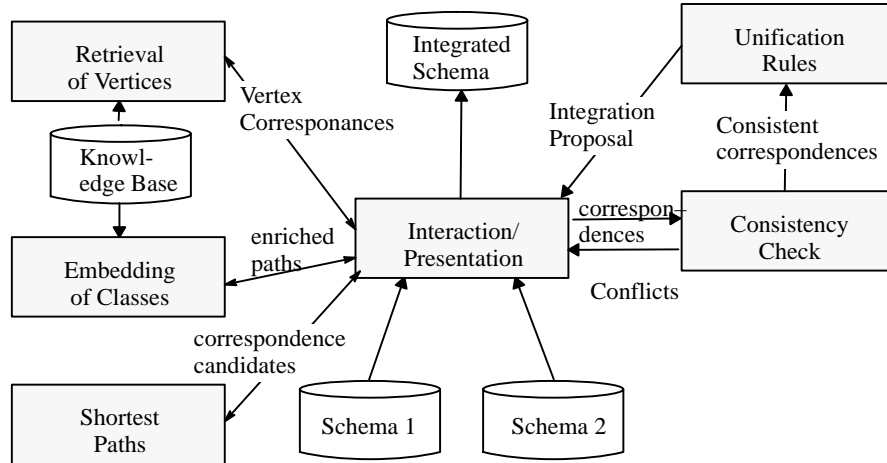


Figure 1.17. Architecture for declarative methodology.

embedding them in a fuzzy conceptual knowledge base, using a restricted form of maximum spanning trees [FN92]. Paths that correspond via their enriched representation also form good candidates for a possible correspondence.

To perform the actual integration, the path correspondences have to be checked for consistency in order to guarantee that the corresponding subschemas can be augmented to form an integrated schema. In case of a conflict the user gets presented the conflicting path correspondences for intellectual resolution. To consistent correspondences, unification rules are applied, which perform the necessary augmentations to arrive at an integrated schema.

In the following we will detail the actual integration step. First, we will introduce the constraints on path correspondences that must be fulfilled so that a sequence of augmenting transformations exists to adapt the corresponding schemas to each other. Then we will present the unification rules that are used to generate integrated schemas from consistent correspondences.

Constraints on Path Correspondences

Because we allow only for augmenting transformations to adapt corresponding subschemas to each other, there cannot be arbitrary correspondences between paths. To rule out inconsistent path correspondences, we use three group of constraints: (a) constraints limiting the number of path correspondences to be investigated, (b) constraints on paths *within a schema*, and (c) mutual constraints on path correspondences.

Throughout this section we use the following notational conventions. For the initial integrated graph $G(V \uplus (V_1 - V/G_1) \uplus (V_2 - V/G_2), E_1 \uplus E_2)$ as defined above, p_i denotes paths with edges from E_1 (possibly of zero length), q_i denotes paths with

edges from E_2 (possibly of zero length), $p_i \sim q_j$ denotes path correspondences, v_i denotes vertices from the common set V , x_i denotes vertices from $V_1 - V$, y_i denotes vertices from $V_2 - V$. Indices, if necessary, are taken from natural numbers. Vertices, edges, and paths with different indices are distinct. Deviations are explained, where they are introduced.

(a) Limiting path correspondences to be investigated: The first group of constraints is more of a practical nature, because it allows us to limit the paths that need to be investigated for correspondence.

1. Completeness: All corresponding paths have to share their start vertex and their end vertex: that is, we consider only paths of the form $v_1.p.v_2 \sim v_1.q.v_2$.

2. Minimality: We consider only path correspondences $v_1.p_1.v_2 \sim v_1.q_1.v_2$, where there exists no path correspondence $v_3.p_2.v_4 \sim v_3.q_2.v_4$ such that $v_1.p_1.v_2$ is subpath of $v_3.p_2.v_4$ or $v_1.q_1.v_2$ is subpath of $v_3.q_2.v_4$. Path correspondences may, however, relate mutually overlapping paths. We can impose this restriction without excluding meaningful correspondences because of the following observation. If $v_1.p_1.v_2 \sim v_1.q_1.v_2$ and $v_3.p_2.v_4 \sim v_3.q_2.v_4$, then also $v_1.p_1.v_2 \uplus v_3.p_2.v_4 \sim v_3.p_2.v_4 \uplus v_3.q_2.v_4$, where \uplus denotes the union of edges. Thus, if $v_1.p_1.v_2$ is subpath of $v_3.p_2.v_4$ or $v_1.q_1.v_2$ is subpath of $v_3.q_2.v_4$, the correspondence of the union of paths can be represented as $v_1.p_1.v_2 \uplus (v_3.p_2.v_4 - v_1.p_1.v_2) \sim v_1.q_1.v_2 \uplus (v_3.q_2.v_4 - v_1.q_1.v_2)$ which can be split into $v_1.p_1.v_2 \sim v_1.q_1.v_2$ and the possibly two correspondences $(v_3.p_2.v_4 - v_1.p_1.v_2) \sim (v_3.q_2.v_4 - v_1.q_1.v_2)$.

If two path correspondences violate this constraint, they are either mutually redundant (Figure 15.18a), or they are inconsistent (Figure 15.18b), because they make a property (*worksFor*) dependent on another property (*takes*) within the same schema via the correspondence with another schema.

The redundant correspondence in Figure 15.18a can be expressed by

- $Student.takes.Course \sim Student.takes.Course$
- $Course.givenBy.Instructor.affiliated.Department \sim Course.offeredBy.Department$

The inconsistent path correspondences in Figure 15.18b cannot be made consistent, because there does not exist any sequence of augmentations that makes the two corresponding subschemas isomorphic. In other words, the correspondences corrupt the structure of the underlying schemas. This leads to anomalies when updating one of the underlying databases. For example, the deletion of a reference *takes* between an instance of *Student* and *Course* in Schema 1 would imply (via correspondence 1) a deletion of *takes* between corresponding instances of *Student* and *Course* in Schema 2, and consequently the deletion of the path *Student.takes.Course.offeredBy.Department*, together with the corresponding path (via correspondence 2) *Student.worksFor.Department*. Thus, the originally independent references *Student.takes.Course* and *Student.worksFor.Department* in Database 1 would become dependent on each other via the introduced correspondences.

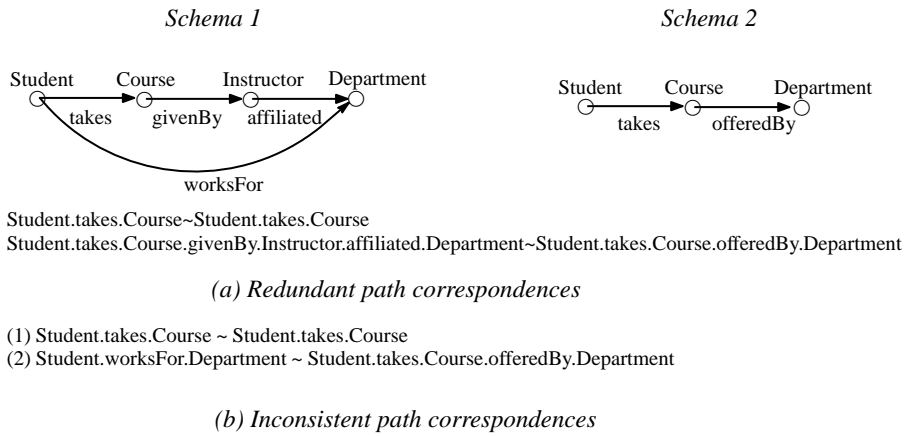


Figure 1.18. Violation of minimality.

From constraints 1 and 2, it suffices to investigate path correspondences of the form $v_1.p.v_2 \sim v_1.q.v_2$, where neither p nor q contains vertices from the set of corresponding vertices V ; consequently, p consists entirely of edges from E_1 , and q consists entirely of edges from E_2 .

(b) Equivalent, conflicting, and inconsistent paths within a schema :

The second group treats paths within a schema:

1. Equivalence of Multiple Role Specializations: Two paths $\odot \leftarrow x_{11} \leftarrow \dots \leftarrow x_{1m} \leftarrow x$, $\odot \leftarrow x_{21} \leftarrow \dots \leftarrow x_{2n} \leftarrow x$ are equivalent, even if some x_{1i} and x_{2j} are distinct (Figure 15.19).

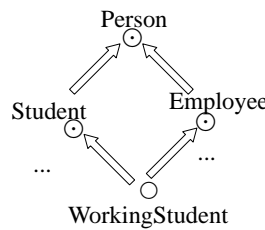


Figure 1.19. Equivalence of multiple role generalizations.

2. Acyclicity: No path contains a cycle, when we interpret the schema graph as an undirected graph. More specifically, no path of the form $v.p.v$ may correspond to a path from the other schema, and two paths of the form $v_1.p_1.v_2$ and $v_1.p_2.v_2$, where p_1 and p_2 have at least one distinct edge, constitute two candidates for alternative correspondences with one path $v_1.q.v_2$, unless they fall into 1. and thus are equivalent role specializations.

(c) Conflicting path correspondences: The third and most interesting group semantically constrains a pair of, or even a set of path correspondences

1. Association vs. Identity: There may not exist any path correspondence of the form $v_1(\Rightarrow | \Leftarrow) * v_2 \sim v_1(\Rightarrow | \Leftarrow | \cdot) * y_1.y_2(\Rightarrow | \Leftarrow | \cdot) * v_2$, where $(\Rightarrow | \Leftarrow)*$ denotes a path in E_1 consisting entirely of specializations or generalization. Likewise $(\Rightarrow | \Leftarrow | \cdot)*$ denotes a path in E_2 , and $y_1.y_2$ is not a specialization edge in E_2 (Figure 15.20).

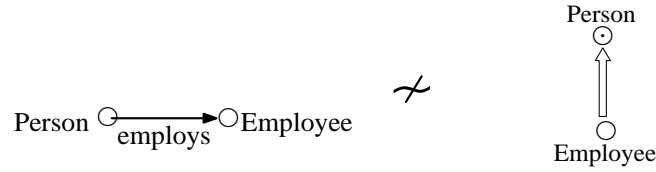


Figure 1.20. Violation of association vs. identity.

2. Disjointedness of Category Specialization (a): There may not exist any two path correspondences of the form

- $\oplus \Leftarrow *x_1.p_1.v_1 \sim \oplus.q_1.v_1$
- $\oplus \Leftarrow *x_1.p_2.v_2 \sim \oplus.q_2.v_2$

unless q_1 and q_2 share some common prefix and are then divided by category specialization or variant record themselves (Figure 15.21). Because x_1 and x_2 denote disjoint sets of objects in one schema, there cannot be any object in the instance of a class \oplus in the other schema that has the property v_1 as well as the property v_2 .

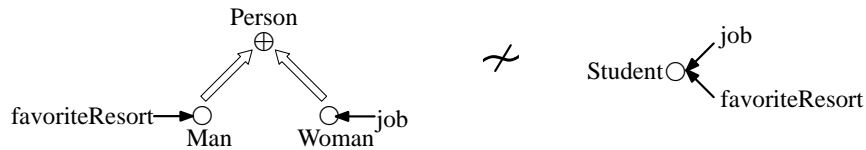


Figure 1.21. Violation of disjointedness of category specialization.

3. Disjointedness of Category Specialization (b): There may not exist any path-correspondence of the form $v_1.p_1 \Rightarrow \oplus \Leftarrow p_2.v_2 \sim v_1.q_1(\Rightarrow * | \Leftarrow * | \Rightarrow * \Leftarrow *)q_2.v_2$. This constraint corresponds closely to 2, with the difference that the category generalization is now not an element of the shared vertices.

4. Existence of a Sequence of Augmenting Transformations: There may not exist any three path correspondences of the form

- $v_1.x_1.p_1.v_2 \sim v_1.q_1.v_2$

- $v_1.x_1.p_2.v_2 \sim v_1.y_1.q_2.v_3$
- $v_1.p_3.v_4 \sim v_1.y_1.q_3.v_4$

For example, for the schemas in Figure 15.22,

- $Product.soldat.Sale.client.Person \sim Product.soldto.Person$
- $Company.sale.Sale.client.Person \sim Company.employs.Salesman.inChargeof.Person$

are consistent path correspondences. The partially integrated schema according to these correspondences is shown in Figure 15.23.

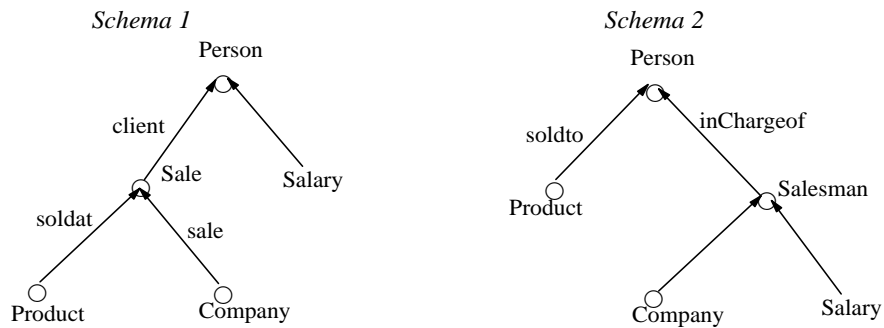


Figure 1.22. Two schemas that cannot be made completely isomorphic by augmentations.

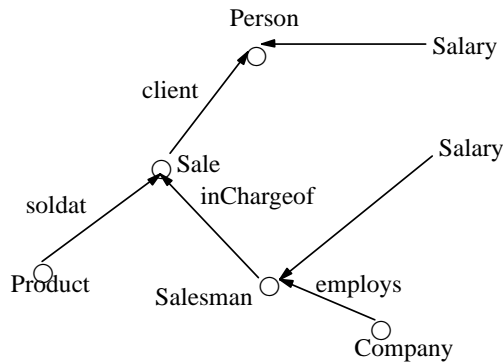


Figure 1.23. One possible partially integrated schema.

But the additional path correspondence

- $Salary.Person \sim Salary.Salesman.inChargeof.Person$

is inconsistent with the other two path correspondences, because, in this case, the schemas could not be augmented in such a way that they become isomorphic, giving rise to similar deletion anomalies as discussed in the framework of Constraint 2 in group (a) (minimality).

Unification via Path Correspondences

To actually integrate schemas or classes within a schema or to process a query against multiple schemas, the path correspondences have to be turned into a sequence of *augmentation transformations*, which make the corresponding subtrees isomorphic. Within this paper we restrict ourselves to *symmetric merging*: that is, in the case of a conflict, we always choose the representation with finer granularity. Thereby we avoid information loss, such that all instances from an external schema that have been transformed to the integrated schema can be transformed back (contracted) to the *same* external schema. On this basis, asymmetric merge, in which one of the schemas forms a normative goal to which all other schemas have to be transformed, can be achieved by contracting the integrated schema appropriately.

The symmetric merging algorithm guides the user recursively through a set of mutually consistent path correspondences and suggests (a) possible correspondence(s) between not yet identified vertices. A suggested correspondence can be either accepted, in which case the algorithm continues suggesting further correspondences, or it can be rejected, in which case the user can either discard the particular path-correspondence(s) completely, or discard only the not-yet-investigated parts of the path correspondence(s).

Algorithm for Symmetric Schema Merge:

Input:

Two schema graphs $G_1 = (V \uplus (V_1 - V), E_1)$ and $G_2 = (V \uplus (V_2 - V), E_2)$. A set C of mutually consistent path-correspondences of the form $v_i.p_i.v_j \sim v_i.q_i.v_j$; $v_i, v_j \in V$, where p_i consists of edges from E_1 , and q_i consists of edges from E_2 .

Output:

An integrated schema graph $G = (V \uplus V' \uplus (V_1 - V - V') \uplus (V_2 - V - V'), E \uplus E'_1 \uplus E'_2)$,
 $V' \subseteq V_1 \cup V_2$,
 $E \subseteq E_1 \cup E_2$,
 $E'_1 = E_1 - E$, $E'_2 = E - E_2$.

Algorithm:

1. $OpenVertices = V \cap Vertices$ in C .
2. For some $v \in OpenVertices$ try to apply one of the tree merging rules (see below), and suggest the result to the user..

3. if none of the tree merging rules can be applied, try to apply one, or more, if v takes part in more than one path-correspondence, without sharing the first edge, of the single path merging rules (see below), and suggest the result to the user.
4. if the user accepted the correspondence,
 - then
 - extend *OpenVertices* with the merged vertices;
 - delete v from *OpenVertices*;
 - extend E with the matched edges
 - else
 - reset E and *OpenVertices* to discard some path correspondences.
5. go to 2 as long as there are vertices in *OpenVertices*.

As already indicated above, we distinguish between two kinds of unification rules. (1) Tree unification rules, and (2) single path unification rules.

Tree Unification Rules Tree unification rules are applied if there exist at least two path correspondences that share a subpath in one schema. Figure 15.24 summarizes the tree merging rules, which can be derived from the augmentations introduced in Section 15.3.2. For better readability, the rules as specified here merge only binary (sub)trees with each other, but they can easily be generalized to merging n -ary trees also. Furthermore, the rules depict only single paths (possibly of zero length), denoted by dashed edges, for the recursive merge. This restriction can be easily overcome. The vertices in *bold/italic* in the merged (integrated) graph at the right-hand side are added to *OpenVertices* by the merging algorithm.

Merging rules (a) and (b) take care of nesting generalizations and references between classes in one of the graphs in such a way that they can be merged with the other (nested) graph. Merging rule (a) is also applicable to merge role generalizations.

Merging rule (c) merges a multivalued reference to a category generalization of classes, which corresponds to single-valued references to a class with multivalued references to corresponding classes.

Finally, merging rule (d) carries out the “substantiation” of categorial properties

Single Path Merging Rules As soon as there are no “true” subtrees to be merged, the rules for merging single paths (Figure 15.25) are used to merge the remaining correspondences.

Merging rule (a) merges single generalization hierarchies.

Merging rule (b) allows for intersecting instances of conflicting specialization paths. In this case, the integrated schema gets a class that is the intersection of instances of the corresponding classes.

Merging rule (c) allows for upward (downward) inheritance of properties and references to other classes.

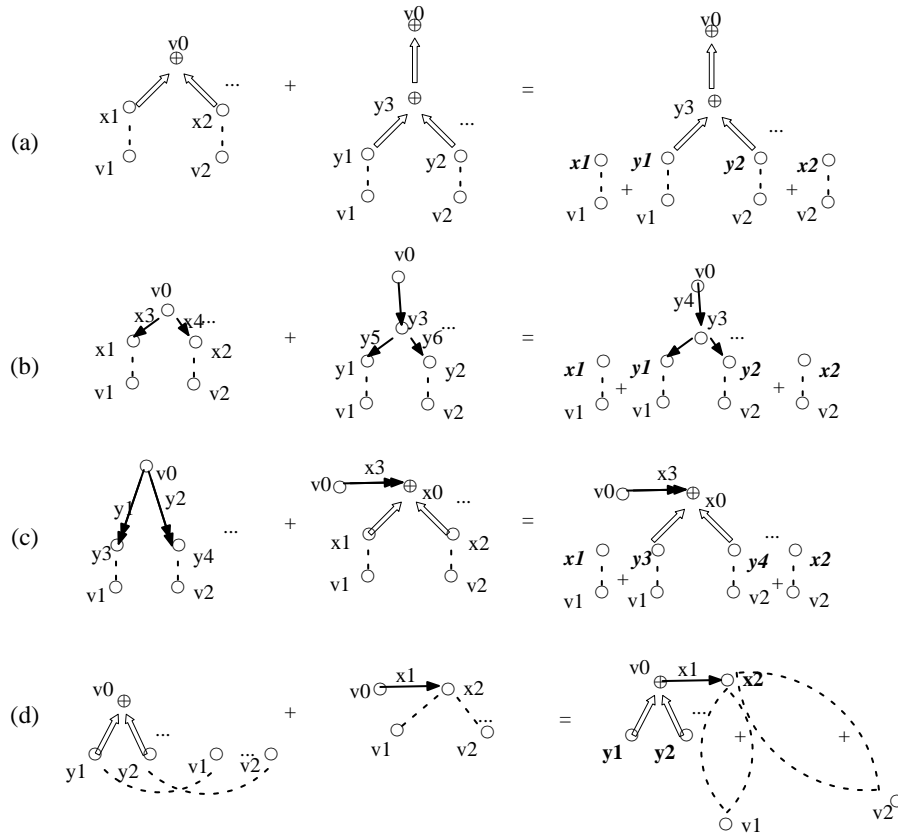


Figure 1.24. Tree merging rules.

Merging rule (d) introduces multivalued references, when they are corresponding to single valued references.

Finally, Merging Rule (e) is the only rule that involves further user interaction, because the actual interrelationship of vertices of two associative paths cannot be determined fully automatically.

1.3.4 Representation of Export and Integrated Schemas in VML

The specification of integrated schemas consists of three parts (recall Figure 15.1):

1. To overcome model heterogeneities, *export schemas* are generated.
2. To overcome structural heterogeneities between corresponding subschemas by *augmentation*, virtual classes are specified to form *import schemas*.

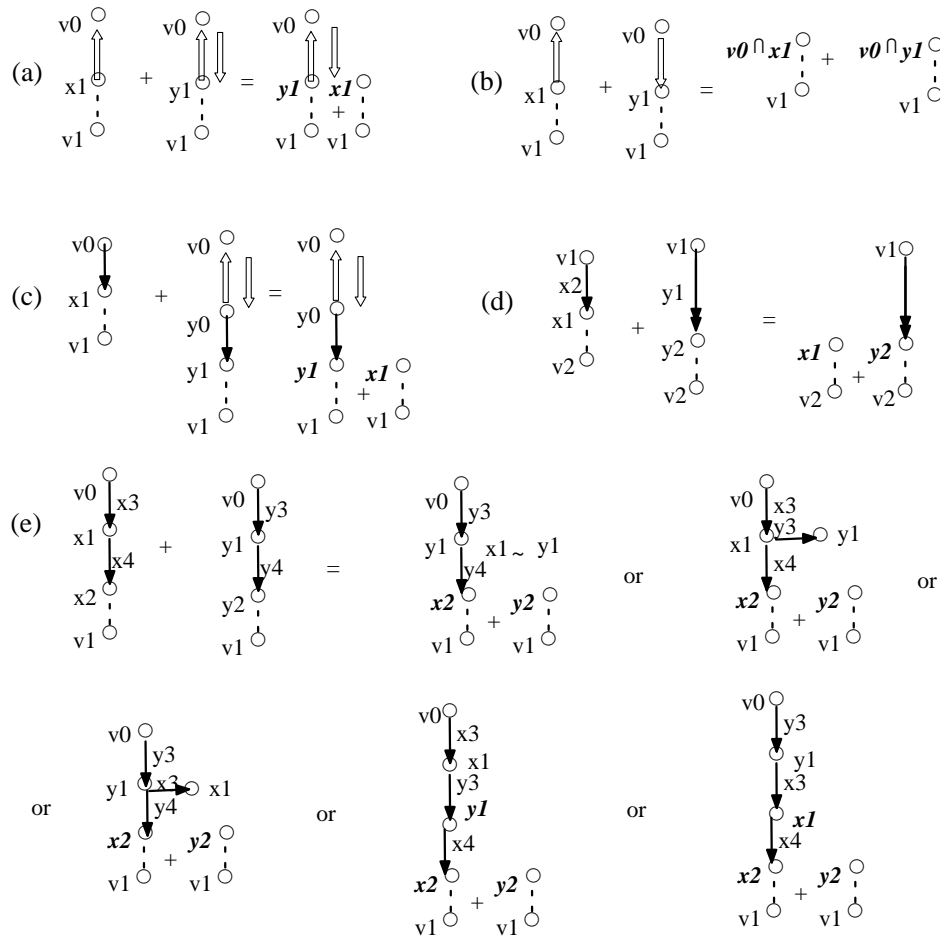


Figure 1.25. Single path merging rules.

- After this homogenization, corresponding classes are combined by *generalization* and corresponding properties are combined by user-defined methods to form the actual *integrated schema*.

On top of this integrated schema at maximum structural granularity, individualized application schemas can be built by *contracting* the integrated schema again. Note that these steps are optional. For example, if no augmentation is required in one of the schemas, then its export schema can be imported and generalized without any modification.

Generation of Export Schemas

Model heterogeneity is overcome by means of metaclasses, which implement interfaces to the modeling constructs of the underlying databases (recall Section 15.2.2). [KFA94] describes an implementation of a metaclass system for connecting Postgres databases. Similar techniques can be used to implement an interface to the Sybase databases. The following two schema specifications define import schemas on top of a Postgres database and a Sybase database.

```

Schema A   Export Schema for Postgres database
Schema A  IMPORT ExternalDatabaseConnectionPrimitives
//...definitions not relevant here
CLASS Person METACLASS PG_Metaclass
  INSTTYPE OBJECTTYPE
  INTERFACE
  METHODS
    SSN(): INTEGER;
    city(): STRING;
    street(): STRING;
  IMPLEMENTATION
  METHODS
    SSN(): INTEGER: {RETURN SELF->getvalue('ssn')};
    city(): STRING; {RETURN SELF->getvalue('city')};};
    street(): STRING; {RETURN SELF->getvalue('street')};};
  END
  INIT Person->init('Person')
  // attach to relation Person in the Postgres database
END

Schema B   Export Schema for Sybase database
Schema B  IMPORT ExternalDatabaseConnectionPrimitives
//...definitions not relevant here
CLASS Person METACLASS Sybase_Metaclass
  INSTTYPE OBJECTTYPE
  INTERFACE
  PROPERTIES addr(): Address;
  METHODS
    address(): Address;
    SSN(): INTEGER;
  IMPLEMENTATION
  METHODS
    address(): Address; {RETURN SELF->getvalue('addr')};};
    SSN(): INTEGER; {RETURN SELF->getvalue('ssn')};};
  END
  INIT Person->init('Person')

```

```

// attach to relation Person in the Sybase database
END

CLASS Address METACLASS Sybase_Metaclass
INSTTYPE OBJECTTYPE
INTERFACE
  PROPERTIES of(): Person;
  METHODS
    city(): STRING;
    street(): STRING;
IMPLEMENTATION
  METHODS
    address(): Address; {RETURN SELF->getValue('addr')};
    SSN(): INTEGER; {RETURN SELF->getValue('ssn')};
  END
  INIT Person->init('Addresses')
// attach to relation Addresses in the Sybase database
END
END_SCHEMA

```

Overcoming Heterogeneities by Introducing Virtual Classes

The export schemas depict a structural heterogeneity in their modeling of address. In Schema A *city* and *street* are associated directly to *Person*, whereas in Schema B *Person* refers to a class *Address* to which *city* and *street* are attached.

Based on the vertex correspondences,

- $A.Person \sim B.Person, A.city \sim B.city, A.street \sim B.street$

and the path correspondences,

- $A.(Person.city) \sim B.(Person.addr.Address.city)$
- $A.(Person.street) \sim B.(Person.addr.Address.street)$

the resolution of inconsistencies proceeds as follows:

Using tree merging rule (b) as specified in Figure 15.24, the class *A.Person* is split into two virtual classes *Person* and *Address* to achieve a representation that is homogeneous to the finer-grained representation in Schema B.

```

Schema A' Import Schema for Schema A
SCHEMA A_IMPORT:
  REFER SCHEMA A;
  IMPORT DatabaseIntegrationPrimitives;
  // provides VirtualClass metaclass
  // ... definitions not relevant here

```

```

CLASS Address METACLASS VirtualClass
INSTTYPE OBJECTTYPE
INTERFACE
METHODS
  of(): Person;
  city(): STRING;
  street() : STRING;
IMPLEMENTATION
METHODS
  of(): Person; {RETURN SELF->virtualOf()};
  city(): STRING;
    {RETURN SELF->virtualOf()->city()};
  street(): STRING;
    {RETURN SELF->virtualOf()->street()};
END
INIT virtualClassOf(A::Person);
END

```

```

CLASS Person METACLASS VirtualClass
INSTTYPE OBJECTTYPE
INTERFACE
METHODS
  address(): Address;
  SSN():INTEGER;
IMPLEMENTATION
METHODS
  address(): Address;
    {RETURN SELF->in(Address)};
  SSN():INTEGER;
    {RETURN SELF->virtualOf()->SSN()};
END
INIT virtualClassOf(A::Person);
END
END_SCHEMA

```

```

Schema B' Import Schema for Schema B
SCHEMA B_IMPORT:
  IMPORT B
  // no further definitions needed; because no augmentation required
END_SCHEMA

```

Both virtual classes *Address* and *Person* are realized as instances of the metaclass *VirtualClass*. In addition to the standard instance-behavior, this metaclass provides

a method *virtualOf*, which returns the object from which a virtual object has been created. This method uses a reference that has been generated by the initialization method *setVirtualOf* provided as well by the metaclass *VirtualClass*. The navigation to original objects thus allows for implementing the desired behavior of virtual classes.

Generation of Integrated Schemas

In the last step, the integrated schema is generated by generalization. For this purpose, the declared correspondences between classes must be differentiated to choose the appropriate metaclass, role generalization for overlapping instances, and category generalization for disjoint instances. For overlapping instances, correspondence predicates have to be specified. Furthermore, for the correspondences between properties, appropriate methods have to be generated, which treat data-conflicts for overlapping instances.

There are several strategies for overcoming data-conflicts: (a) prefer data from one database, (b) aggregate conflicting data, (c) ask the user to resolve the data-conflicts intellectually at query time. Appropriate method code for options (a) and (c) can be easily generated from user declarations by using code generation patterns. For option (b), the user has to provide the code for the desired aggregation method intellectually, because all the different possible cases cannot be anticipated with reasonable effort by a declarative methodology.

In our example, we have identified persons via their social security number, and addresses via the person they are attached to. Furthermore we have chosen strategy (a) for resolving data conflicts, preferring data from Database B, if available.

Schema A_and_B Integrated schema(view) of Databases A and B

```

SCHEMA A_and_B_integrated
  REFER SCHEMA A_IMPORT, B;
  IMPORT DatabaseIntegrationPrimitives;
  // provides RoleGeneralization metaclass
  // ... definitions not relevant here

CLASS Person METACLASS RoleGeneralization
  INSTTYPE OBJECTTYPE
  INTERFACE
    METHODS
      SSN():INTEGER
      address(): Address;
  IMPLEMENTATION
    METHODS
      SSN():INTEGER;
      {VAR localobj: OID;
      localobj:= SELF->in(B_IMPORT::Person);

```

```

        IF localobj != NULL THEN RETURN localobj->SSN()
        ELSE RETURN SELF->in(A_IMPORT::Person)->SSN();}
address(): Address;
    {VAR localobj: OID;
    localobj:= SELF->in(B_IMPORT::Person);
    IF localobj != NULL THEN RETURN localobj->address()
    ELSE RETURN SELF->in(A_IMPORT::Person)->address();}
END
INIT roleGeneralizationOf(A_IMPORT::Person,
                          B_IMPORT::Person,"SSN","SSN");
END

CLASS Address METACLASS RoleGeneralization
INSTTYPE OBJECTTYPE
INTERFACE
METHODS
    of(): Person;
    city(): STRING;
    street(): STRING;
IMPLEMENTATION
METHODS
    of(): Person;
    {VAR localobj: OID;
    localobj := SELF->in(B_IMPORT::Address);
    IF localobj != NULL THEN RETURN localobj->of();
    ELSE RETURN SELF->in(A_IMPORT::Address)->of();};
    city(): STRING;
    {VAR localobj: OID;
    localobj := SELF->in(B_IMPORT::Address);
    IF localobj != NULL THEN RETURN localobj->city();
    ELSE RETURN SELF->in(A_IMPORT::Address)->city();};
    street(): STRING;
    {VAR localobj: OID;
    localobj := SELF->in(B_IMPORT::Address);
    IF localobj != NULL THEN RETURN localobj->street();
    ELSE RETURN SELF->in(A_IMPORT::Address)->street();};
END
INIT roleGeneralizationOf(A_IMPORT::Address,
                          B_IMPORT::Address,"of", "of");
END
END_SCHEMA

```

1.4 Conclusion

In this paper we presented our approach on interoperability or integration of information bases. The Schema Integrator Workbench is a tool that supports a designer in the construction of integrated application schemas (individualized views onto existing heterogeneous databases). Such integrated application schemas are used by an interoperable object-oriented database system based on VODAK, which offers the run-time support for global access to existing heterogeneous database systems, for example, global transaction management.

The schema integration methodology implemented in the schema integrator workbench includes several integration steps: A syntactic transformation step provides for a syntactically uniform VODAK interface to the external information bases, describing their database schemas (including constraints), retrieval and manipulation capabilities, and their file formats.

A semantic integration step is needed to combine the several VODAK schemas. Structural and semantic differences in representation and conflicts in naming and scaling have to be resolved, correspondences between objects have to be identified, and appropriate user views have to be determined in order to specify an (or several) integrated view(s). This obviously includes semantic enrichment which makes implicit structure and semantics explicit and associates additional behavior, which is hidden in local application programs or, even worse, in informal local conventions. We discussed a declarative methodology that allows users to declare correspondences between schema constituents (e.g., classes, types, attributes) that contain overlapping data according to their world view. The declared correspondences are checked for consistency, and from consistent correspondences an integrated VODAK subschema is generated, together with mappings from the heterogeneous local corresponding subschemas. The novel feature of our methodology is that it is based on a flexible notion of schema unification, which allows for augmenting the structural granularity of corresponding subschemas in order to overcome their heterogeneity.

We want to extend our notion of schema unification via augmentation to partially assist also the integration of methods. The general problem of method integration is, of course, undecidable. However, we believe that special kinds of methods, like path methods that navigate along a schema path to retrieve or modify distant information [MGPF92a] can be broken down to their underlying schema constituents. Thus, their integration can be assisted by extending the structural unification presented in this paper. In addition, combining forms (program patterns), which do not touch the implementation of a method but just invoke one or more original methods, within the context of an integrated schema, can be used to integrate methods consistently with the integration of their signature (interface).

Furthermore, we want to improve our methodology in such a way that we can also cope with local schema evolution. For this purpose, we need to extend the concepts for testing the consistency of path correspondences in such a way that a partially integrated schema is taken into account and to include trigger mechanisms that identify those path correspondences that are invalidated by a local schema

evolution.

The concepts and prototypes developed for the schema integrator workbench, as well as the global transaction management to be offered by a database management system providing access to heterogeneous existing databases, are in the process of being transferred into the industrial cooperation project IRO-DB within the framework of ESPRIT-III.

Bibliography

- [AF95] K. Aberer and G. Fischer. Semantic Query Optimization for Methods in Object-Oriented Database Systems. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE 95)*, Taipei, Taiwan, 1995, pp. 70–79.
- [AK93] K. Aberer and W. Klas. The Impact of Multimedia Data on Database Management Systems. Technical Report No. 763, GMD, July 1993.
- [AKF94] K. Aberer, W. Klas, and A. Furtado. Designing a User-Oriented Query Modification Facility in Object-Oriented Database Systems. In *Proceedings of CAiSE*94*, Utrecht, The Netherlands, 1994, pp. 380–393.
- [AK94] K. Aberer and W. Klas. Supporting Temporal Multimedia Operations in Object-oriented Database Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Boston, USA*, May 1994, pp. 352–361.
- [BA94] K. Böhm and K. Aberer. An Object-Oriented Database Application for HyTime Document Storage. In *Proceedings of the Conference on Information and Knowledge Management (CIKM94)*, Gaithersburg, Md., 1994, pp. 26–33.
- [BAH94] K. Böhm, K. Aberer, and C. Hüser. Introducing D-STREAT—The Impact of Advanced Database Technology on SGML Document Storage. <TAG>, 7:2 (February 1994), pp. 1–4.
- [BBG⁺83] C. Beeri, P. Bernstein, N. Goodman, M. Lai, and D. Shasha. A Concurrency Control Theory for Nested Transactions. In *Proceedings 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1983, pp. 45–62.
- [BBG89] C. Beeri, P. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transaction Systems. *Journal of the ACM*, 36:2 (April 1989), pp. 230–269.

- [BGS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal*, 1:2 (October 1992), pp. 181–239.
- [BHP92] M. Bright, A. Hurson, and S. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *Computer*, 25:3 (March 1992), pp. 50–60.
- [Bre90] Y. Breitbart. Multidatabase Interoperability. *SIGMOD Record*, 19:3 (September 1990), pp. 53–60.
- [BST90] Y. Breitbart, A. Silberschatz, and G. Thompson. Reliable Transaction Management in a Multidatabase System. In *Proceedings ACM SIGMOD*, 1990, pp. 215–224.
- [BSW88] C. Beeri, H.-J. Schek, and G. Weikum. Multilevel Transaction Management, Theoretical Art or Practical Need? In *Proceedings EDBT'88*, Venice, 1988, pp. 134–154.
- [CHS91] C. Collet, M. Huhns, and W.-M. Shen. Resource Integration Using a Large Knowledge Base in Carnot. *Computer*, 24:12 (December 1991), pp. 55–62.
- [CRR91] P. Chrysanthis, S. Raghuram, and K. Ramamritham. Extracting Concurrency from Objects: A Methodology. In *Proceedings ACM SIGMOD Conference*, 1991, pp. 108–117.
- [CS91] M. Castellanos and F. Saltor. Semantic Enrichment of Database Schemas: An Object-Oriented Approach. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems, IMS 91*, Kyoto, Japan, 1991, pp. 71–78.
- [CT93] W. Chen and V. Turau. Code Generation for VML Based on Tree Pattern Matching and Dynamic Programming. Technical Report 755, GMD, 1993.
- [CT94] W. Chen and V. Turau. An Optimized Implementation for VML Based on Pattern Matching. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM94)*, Gaithersburg, Md., 1994, pp. 88–96.
- [CTK94] W. Chen, V. Turau, and W. Klas. Efficient Dynamic Look-up Strategy for Multimethods. In *Proceedings of the Eighth European Conference on Object-Oriented Programming (ECOOP'94)*, Bologna, Italy, 1994, pp. 408–431.
- [Dav78] C. Davies. Data Processing Spheres of Control. *IBM Systems Journal*, 17:2 (1978), pp.179–198.

- [DEL⁺89] W. Du, A. Elmagarmid, Y. Leu, S. Ostermann. Effects of Autonomy on Maintaining Global Serializability in Heterogeneous Database Systems. Technical Report, Purdue University, 1989.
- [DL87] P. Dwyer and J. Larson. Some Experiences with a Distributed Database Testbed System. In *Proceedings IEEE 75*, 1987, pp. 633–647.
- [Elm92] A. Elmagarmid (Ed.). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, Calif., 1992.
- [EN89] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Menlo Park, Calif., 1989.
- [FKN91] P. Fankhauser, M. Kracker, and E. Neuhold. Semantic vs. Structural Resemblance of Classes. *Special SIGMOD RECORD Issue on Semantic Issues in Multidatabase Systems*, 20:4 (December 1991), pp. 59–63.
- [FN92] P. Fankhauser and E. Neuhold. Knowledge-Based Integration of Heterogeneous Databases. In *Proceedings of IFIP DS-5 Conference—Semantics of Interoperable Database Systems*, Lorne, Victoria, Australia, 1992, pp. 150–170.
- [FX94] P. Fankhauser and Y. Xu. MarkItUp! An Incremental Approach to Document Structure Recognition. *Proceedings of the 1994 International Conference on Electronic Publishing, Document Manipulation and Typography*, 1994, pp. 447–456.
- [GF92] T. Goettke and P. Fankhauser. DREAM 2.0, User Manual. Technical Report, No. 660, GMD, 1992.
- [GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM TODS*, 8:2 (June 1983), pp. 186–213.
- [GMD95] GMD-IPSI. VODAK V4.0 User Manual. Technical Report No. 910, GMD, 1995.
- [Gra81] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th VLDB Conference*, 1981, pp. 144–154.
- [HGPK94] M. Halper, J. Geller, Y. Perl, and W. Klas. Integrating a Part Relationship into an Open OODB System using Metaclasses. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM94)*, Gaithersburg, Md., 1994, pp. 10–17.
- [HR83] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15:4 (December 1983), pp. 287–317.

- [HR90] S. Hayne and S. Ram. Multi-User View Integration System (MUVIS): An Expert System for View Integration. In *Proceedings of the Sixth International Conference on Data Engineering*, 1990, pp. 402–409.
- [HW91] C. Hasse and G. Weikum. A Performance Evaluation of Multi-Level Transaction Management. In *Proceedings VLDB*, 1991, pp. 55–66.
- [JPSL⁺88] G. Jacobsen, G. Piatetsky-Shapiro, C. Lafond, M. Rajinikanth, and J. Hernandez. CALIDA: A Knowledge-Based System for Integrating Multiple Heterogeneous Databases. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, Jerusalem, Israel, 1988, pp. 3–18.
- [KAN94] W. Klas, K. Aberer, and E. Neuhold. Object-Oriented Modeling for Hypermedia Systems Using the VODAK Modeling Language (VML). In A. Dogac, T. Ozsu, A. Biliris, and T. Sellis, eds., *Advances in Object-Oriented Database Systems*, NATO ASI Series, Springer Verlag, Berlin – Heidelberg, 1994, pp. 389–433.
- [KD90] M. Kaul and K. Drost. ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. In *Proceedings of the IEEE International Conference on Data Engineering*, 1990, pp. 2–10.
- [KFA94] W. Klas, G. Fischer, and K. Aberer. Integrating Relational and Object-Oriented Database Systems Using a Metaclass Concept. *Journal of Systems Integration*, 4:4 (December 1994), pp. 341–372.
- [Kla90] W. Klas. *A Metaclass System for Open Object-Oriented Data Models*. PhD thesis, Technical University of Vienna, Vienna, Austria, 1990.
- [KLS90] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the Sixteenth VLDB Conference*, 1990, pp. 95–106.
- [KN90] M. Kaul and E. Neuhold. Supporting Interoperability of Heterogeneous Information Bases by Complex View Hierarchies. In *Workshop on Perspektiven der Datenbank-Technik*, University of Bern, Switzerland, October 1990, pp. 23–31.
- [KNS90a] W. Klas, E. Neuhold, and M. Schrefl. Metaclasses in VODAK and their Application in Database Integration. Technical Report No. 462, GMD, 1990.
- [KNS90b] W. Klas, E. Neuhold, and M. Schrefl. Using an Object-Oriented Approach to Model Multimedia Data. *Computer Communications*, 13:4 (August 1990), pp. 204–216.

- [LA86] W. Litwin and A. Abdellatif. Database Interoperability. *IEEE-Computer*, 1986, pp. 10–18.
- [LBE⁺82] W. Litwin, J. Boudenat, C. Esculier, A. Ferrier, A. Glorieux, J. La Chimia, K. Kabbaj, C. Moulinoux, P. Rolin, and C. Stangret. SIRIUS Systems for Distributed Data Management. In H.-J. Schneider, ed., *Distributed Data Bases*, North-Holland, The Netherlands, 1982, pp. 311–366.
- [Lit85] W. Litwin. An Overview of the Multidatabase System MRDSM. In *Proceedings of the ACM National Conference*, Denver, Colo., USA, 1985, pp. 495–504.
- [LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22:3 (September 1990), pp. 267–293.
- [LMWF92] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, Calif., 1992.
- [LR82] T. Landers and R. Rosenberg. An Overview of Multibase. In H.-J. Schneider, ed., *Distributed Databases*, North-Holland, The Netherlands, 1982, pp. 153–184.
- [MGG86] J. Moss, N. Griffeth, and M. Graham. Abstraction in Recovery Management. In *ACM SIGMOD Conference*, 1986.
- [MGPF92a] A. Mehta, J. Geller, Y. Perl, and P. Fankhauser. Algorithms for Access Relevance to Support Path-Method Generation in OODBs. In *Proceedings of the Fourth International Hong Kong Computer Society Database Workshop*, Shatin, Hong Kong, 1992, pp. 183–200.
- [MGPF92b] A. Mehta, J. Geller, Y. Perl, and P. Fankhauser. Algorithms for Computing Access Relevance in Object-Oriented Databases. In *Proceedings of the First International Conference on Information and Knowledge Management*, Maryland, USA, 1992, pp. 657.
- [MKK95] F. Moser, A. Kraiß, and W. Klas. L/MRP: A Buffer Management Strategy for Interactive Continuous Data Flows in a Multimedia DBMS. To appear in *Proceedings VLDB Conference 1995*, USA, 1995. Morgan Kaufmann.
- [MNE88] M. Mannino, S. Navathe, and W. Effelsberg. A Rule-Based Approach for Merging Generalization Hierarchies. *Information Systems*, 13:3 (1988), pp. 257–272.
- [Mos85] J. Moss. *Nested Transactions—An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Mass., 1985.

- [Mot87] A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13:7 (July 1987), pp. 785–798.
- [MR91] P. Muth and T. Rakow. Atomic Commitment for Integrated Database Systems. In *Proceedings of IEEE Seventh International Conference on Data Engineering*, Kobe, Japan, 1991, pp. 296–304.
- [MR93] P. Muth and T. Rakow. VODAK Open Nested Transactions—Visualizing Database Internals. In *Proceedings SIGMOD*, 1993, pp. 558–559.
- [MRB⁺92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. Korth, and A. Silberschatz. Ensuring Transaction Atomicity in Multidatabase Systems. In *Proceedings SIGMOD*, 1992, pp. 288–297.
- [MRKN92] P. Muth, T. Rakow, W. Klas, and E. Neuhold. A Transaction Model for an Open Publication Environment. In Ahmed K. Elmagarmid, ed., *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, Calif., 1992, pp. 159–218.
- [MRW⁺93] P. Muth, T. Rakow, G. Weikum, C. Hasse, and P. Brössler. Semantic Concurrency Control in Object-Oriented Database Systems. In *Proceedings of IEEE Ninth International Conference on Data Engineering*, Vienna, Austria, 1993, pp. 233–242.
- [Mut94] P. Muth. *Transaktionsverwaltung in heterogenen und autonomen Datenbanksystemen*. Ph.D. Thesis, Technical University Darmstadt, Darmstadt, Germany, 1994.
- [NS88] E. Neuhold and M. Schrefl. Dynamic Derivation of Personalized Views. In *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, Calif., 1988, pp. 183–194.
- [O’N86] P. O’Neil. The Escrow Transactional Method. *ACM TODS*, 11:4 (December 1986), pp. 405–430.
- [REC⁺89] M. Rusinkiewicz, R. Elmasri, B. Czejdo, D. Georakopoulos, G. Karabatis, A. Jamoussi, L. Loa, and Y. Li. OMNIBASE: Design and implementation of a Multidatabase System. In *Proceedings of the First Annual Symposium in Parallel and Distributed Processing*, Dallas, Tex., 1989, pp. 162–169.
- [RGN90] T. Rakow, J. Gu, and E. Neuhold. Serializability in Object-Oriented Database Systems. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, Calif., 1990, pp. 112–120.

- [RLMN93] T. Rakow, M. Löhr, F. Moser, K. Süllow, and E. Neuhold. Using Object-Oriented Database Systems for Multimedia Applications. *it+ti, Oldenbourg*, (3), 1993, pp. 4–17.
- [RNL95] T. Rakow, E. Neuhold, and M. Loehr. Multimedia Database Systems - The Notions and the Issues. In Georg Lausen, editor, *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, Springer, Germany, March 1995, pp. 1–29.
- [SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22:3 (September 1990), pp. 183–236.
- [SLCN88] A. Sheth, J. Larson, A. Cornelio, and S. Navathe. A Tool for Integrating Conceptual Schemas and User Views. In *Proceedings of the Fourth International Conference on Data Engineering*, 1988, pp. 176–183.
- [SN90] M. Schrefl and E. Neuhold. A Knowledge-Based Approach to Overcome Structural Differences in Object-Oriented Database Integration. In M. Meersmann, Z. Shi, C. Kung (Eds.) *Artificial Intelligence in Databases and Information Systems*, Proceedings of the IFIP TC2/TC8 Working Conference, North Holland, 1990, pp. 265–304.
- [SN88] M. Schrefl and E. Neuhold. Object Class Definition by Generalization Using Upward Inheritance. In *Proceedings of the Fourth International Conference on Data Engineering*, 1988, pp. 4–13.
- [SSG⁺91] A. Savasere, A. Sheth, S. Gala, S. Navathe, and H. Marcus. On Applying Classification to Schema Integration. In *Proceedings of IEEE First International Workshop on Interoperability in Multibase Systems*, Kyoto, Japan, 1991, pp. 258–261.
- [TBC⁺87] M. Templeton, D. Brill, A. Chen, S. Dao, E. Lund, R. McGregor, and P. Ward. Mermaid: a front end to distributed heterogeneous databases. In *Special Issue on Distributed Database Systems, Proceedings of the IEEE 75*, May 1987, pp. 695–708.
- [TK95a] H. Thimm and W. Klas. Payout Management—An Integrated Service of a Multimedia Database Management Systems. To appear in *Proceedings of the First International Workshop on Multi-Media Database Management Systems, Blue Mountain Lake, NY, August 28-30, 1995*. IEEE Computer Society Press, 1995.
- [TK95b] H. Thimm and W. Klas. *Reactive Payout Management - Adapting Multimedia Presentations to Contradictory Constraints*. Technical Report No. 916, GMD, 1995.

- [TR93a] H. Thimm and T. Rakow. Upgrading Multimedia Data Handling Services of a Database Management System by an Interaction Manager. Technical Report No. 762, GMD, 1993.
- [TR93b] V. Turau and T. Rakow. A Schema Partition for Multimedia Database Management Systems. Technical Report No. 729, GMD, 1993.
- [Wei88] W. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37:12 (December 1988), pp. 1488–1505.
- [Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM TODS*, 16:1 (March 1991), pp. 132–180.
- [WHBM90] G. Weikum, C. Hasse, P. Brössler, and P. Muth. Multi-Level Recovery. In *Proceedings of Ninth Symposium on Principles of Database Systems (PODS)*, Nashville, Tenn., 1990, pp. 109–123.
- [WHW90] S. Widjojo, R. Hull, and D. Wile. A Specification Approach to Merging Persistent Object Bases. In *Fourth International Workshop on Persistent Object Systems—Implementing Persistent Object Bases: Principles and Practices*, Morgan Kaufmann, San Mateo, Calif., 1990, pp. 267–278.
- [WS92] G. Weikum and H.-J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A.K. Elmagarmid, ed., *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, Calif., 1992, pp. 515–553.