OMS/Java : Model Extensibility of OODBMS for Advanced Application Domains

Andreas Steiner, Adrian Kobler and Moira C. Norrie
{steiner, kobler, norrie}@inf.ethz.ch

Institute for Information Systems
ETH Zurich, CH-8092 Zurich, Switzerland

Abstract. We show how model extensibility of object-oriented data management systems can be achieved through the combination of a high-level core object data model and an architecture designed with model extensibility in mind. The resulting system, OMS/Java, is both a general data management system and a framework for the development of advanced database application systems. All aspects of the core model – constructs, query language and constraints – can easily be generalised to support, for example, the management of temporal, spatial and versioned data. Specifically, we show how the framework was used to extend the core system to a temporal object-oriented database management system.

1 Introduction

Extensibility has often been considered a purely architectural issue in database management systems (DBMS). In the 1980s, there was an increase in the various forms of DBMS that appeared — many of which were tailored to specific application domains such as Geographical Information Systems or Computer Aided Design systems. It was recognised that no single DBMS could hope to satisfy the requirements of all such applications and the idea of an extensible DBMS emerged. Thereafter, a number of extensible systems appeared – including kernel e.g. [CDKK85], customisable e.g. [HCL+90], toolkit e.g. [CDG+89] and generator e.g. [MBH+] systems. In these systems, the emphasis tends towards providing architectural support in terms of how to engineer new DBMS from existing frameworks and services such as storage management, transaction management, access methods and query optimisation and we therefore call this *architecture extensibility*. The idea evolved of a *database engineer* as someone whose task would be to construct an advanced DBMS based on such a system – and, frequently, this was a significant task. A discussion of general requirements of such systems is given in [GD94].

In contrast, there have been proposals for various forms of model extensibility in which new types and/or operators can be introduced in a DBMS to meet the needs of specific application domains, e.g. [BLW88, SC94, Cha96]. We aim for a higher-level of model extensibility in which *all* aspects of a data model – structures, query language and constraints – can be generalised. This ensures upward compatibility in that the core model, languages and mode of working

remain essentially the same. For example, generalisation to a temporal system enables temporal semantics to be associated with all constructs, operations and constraints of the core model. Further, non-temporal queries will be unaffected and temporal queries have the same form as non-temporal queries – with only prefixes required to specify that a temporal semantics is required. Similarly, the DBMS could be generalised to handle spatial data, versioned data, multimedia data and any combination of these.

OMS/Java is both a framework and an object data management system designed with such model extensibility in mind. The system is based on the OM model which, through a clear separation of model concepts, orthogonality of design and general modelling expressiveness has the properties required of the core model of such an extensible system. The development and architecture of OMS/Java exploited current software engineering principles such as application frameworks and design patterns to ensure that benefits of reusability and extensibility could be fully exploited.

In this paper, we describe the OMS/Java architecture and system and how its extensibility was exploited to implement a temporal object-oriented database management system with minimal effort. We begin in Sect. 2 and 3 with a discussion of the general requirements for systems with true model extensibility. This includes a discussion of the requirements of an appropriate core model along with a description of our core model OM. Section 4 then presents the general OMS/Java architecture, followed with a description of the OMS/Java system. To demonstrate the process of model extensibility, we describe the development of the temporal OMS/Java system, TOMS/Java, in Sect. 5. We conclude with a summary and outline of future work in Sect. 6.

2 Architecture

2.1 Reusable Parts of a DBMS

Modern DBMS have to support a variety of tasks and requirements [EN94, HS95]. A DBMS manages *persistent* data, meaning that data exists longer than tasks run by application programs. Data structures are defined to describe and store the data in a uniform way. Sophisticated *access methods* (for example, using indexing or hashing) provide for efficient data access. *(Declarative) languages* and *operations* need to be supported which allow the definition, modification and retrieval of data. The DBMS should also provide means and techniques to *optimise* data retrieval. *Transaction management* and *concurrency control* allow multiple users to access the same data simultaneously, avoiding any problems which might evolve. Additionally, a modern DBMS should support concepts to provide *data consistency*, *recovery* from system crashes and *database security* and *authorisation*.

The implementation of an *extensible* data model as a DBMS should still support all of these tasks while allowing the user to extend the model and system with new functionality. Currently available DBMS support the features listed

above in one way or another. Relational DBMS support a single data structure – the relation – and a non-extensible functionality. Object-oriented DBMS support extensibility of the data structures through the use of type constructors such as tuple, set, bag, list and so on and extensibility of the functionality through the possibility to store methods in the database.

It is difficult to efficiently and effectively support application domains which are based on specialised data models – for example, the management of temporal or spatial data or versioning – using such systems. The limitations of the relational data model with respect to data structures and functionality make it necessary to either build the additional constructs needed into the application itself or implement a specialised DBMS. With respect to object-relational or object-oriented DBMS, it is possible to extend the system to support such advanced application domains. We have investigated such an approach in the area of temporal databases [SN97b]. The major drawbacks of this approach, however, are that the DBMS cannot use the special semantics – for example, the temporal semantics – for query optimisation, that the resulting retrieval and update operations turn out to be very complicated and error prone and that the specification of specialised integrity constraints usually may only be supported through the implementation of methods.

Our idea of extensibility thus goes a step further. We would like to have a DBMS based on an extensible data model which allows the reuse of features such as persistence, query optimisation, efficient access methods, transaction management, crash recovery and authorisation, while supporting a generalised form of its data structures, algebra operations , query language and integrity constraints with special semantics. Such a system is upwards compatible in the sense that the basic data structures, operations and constraints can always be used, even in more advanced, generalised forms of the underlying data model, and that for each generalisation of the data model, the same concepts and manner of work can be used. The next section discusses in more detail the generalisation approach and introduces an architecture supporting this method of achieving more advanced data models.

2.2 The Generalisation Approach

A data model $\mathtt{M}$ can be considered to consist of three parts – the data structures $\mathtt{DS}$, the operations for data retrieval and update $\mathtt{OP}$ and the integrity constraints $\mathtt{C}$, $\mathtt{M} = (\mathtt{DS}, \mathtt{OP}, \mathtt{C})$. In [SN97c], we have introduced a temporal data model which – in contrast to other temporal data models – enhances all three parts of a non-temporal data model to support the management of temporal data. Figure 1 shows the three parts of a data model. Most of the temporal data models extend non-temporal data models in that they add special timestamp attributes to store temporal data and add special operations to query temporal data. Other ones extend the data structures but supply a special temporal algebra. Our generalisation approach, however, considers all three parts of the data model by supporting temporal object identifiers, temporal algebra operations and temporal constraints. The resulting temporal object model is independent of any

type system and orthogonal in the sense that anything modelled as an object, including collections of objects, constraints and even types, can be timestamped.

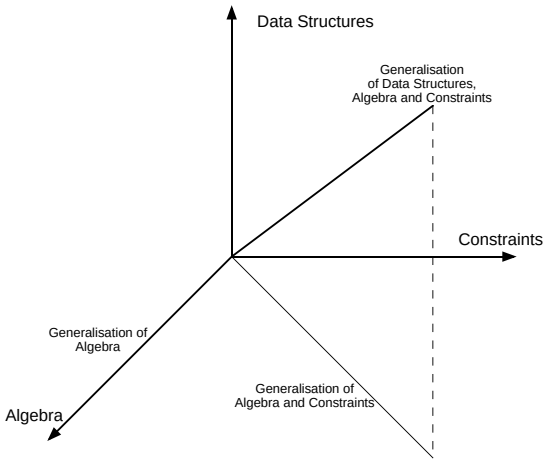

Fig. 1. The three parts of a data model

A DBMS should additionally provide for a language which can be used to specify the data structures and integrity constraints used for an application (the data definition language), update operations (the data manipulation language) and queries (the query language). This language thus consists of three sublanguages.

In order to come up with a DBMS which allows the reuse of the general tasks of query optimisation, efficient access methods, transaction management, crash recovery and so on, but supplies generalised data structures, algebra operations and integrity constraints, the notion of an object identifier, the algebra operations, the integrity constraints and the language supported must be exchangeable, respectively, extendible. Such an architecture is depicted in Fig. 2.

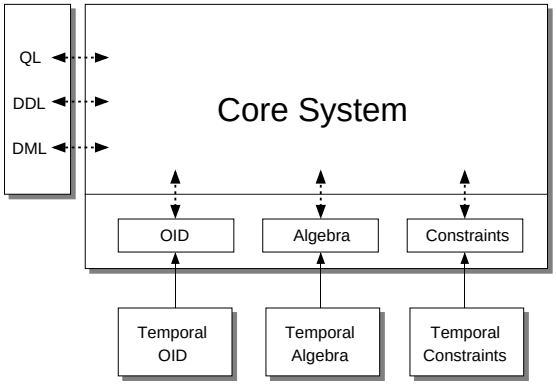

Fig. 2. Another form of extensibility in a DBMS

The core system supports the basic tasks of persistence, query optimisation, efficient access methods, transaction management, crash recovery, authorisation and so on. The object identifier, the algebra and constraints can be extended to support the needs of an advanced application domain, for example, temporal databases. In this case, the notion of an object identifier is generalised into a temporal object identifier which – besides a unique reference – contains a lifespan denoting when the object existed, for example with respect to the real world. To be able to query the temporal data stored in the form of temporal objects, the non-temporal algebra operations are overridden with algebra operations having the desired form of temporal semantics. Finally, the integrity constraints are replaced by temporal integrity constraints.

The same approach can be used to support data models supporting the management of spatial data or versioning. For spatial data, we exchange, respectively, extend the notion of an object identifier with a spatial identifier, which means that the position of an object or the space that it covers is attached to the object identifier. Algebra operations and integrity constraints are replaced with operations and constraints which are able to deal with the new semantics. For versioning, the version number is attached to the object identifier and again, operations and integrity constraints which are able to deal with the versioned objects are added to the system.

3 Requirements for Model Extensibility

The generalisation approach, as introduced in the last section, obviously deals with objects and algebra operations and integrity constraints referring to these objects. Usually, it is argued that advanced application domains such as the management of temporal, spatial and versioned data can be supported using extensible DBMS which support abstract data types. Methods have to be written to support queries and constraints dealing with the special semantics of the application domain. Clearly, such approaches are based simply on the type system of the DBMS used. The disadvantage of these approaches are, for example, that the special semantics usually cannot be used for query optimisation, and that the specification of queries and integrity constraints in the core model and the advanced model are not similar.

Our generalisation approach is based on the object level, which means, that we do not want to deal with instances of types and add additional attributes with special semantics. We want to deal with objects which inherently have special semantics, and a closer look at them reveals their property values. In other words, we advocate an approach which takes the special semantics of an advanced data model such as a temporal data model into account and does not treat temporal properties as just another, but somehow a bit more special, attribute value.

Our idea of an extensible data model is that it should support a rich set of algebra operations and constraints which work on collections of objects. The following subsections discuss the requirements for model extensibility in more detail and introduce a data model which supports the generalisation approach.

3.1 Separation of Typing and Classification

[Nor95] distinguishes between classifying entities into categories according to general concepts of the application domain and describing the representation of entities within the database in terms of types. For example, we can classify persons as being employees, students or tennis players. Each employee, student and tennis player is a person. Students can further be classified as studying either engineering or mathematics. This is depicted on the left hand side of Fig. 3. On the other hand, we represent persons as having a name and a birthdate, while employees additionally have an employment number and a salary, students have a student number and tennis players have a ranking and are member of a club. This type hierarchy is depicted on the right hand side of Fig. 3.

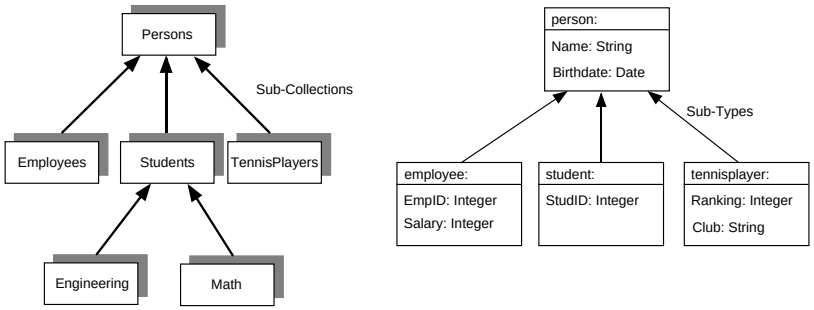

Fig. 3. Distinguishing typing and classification

In reality, a person may be classified as a student and a tennis player simultaneously. This means, that a person may play the role of an employee and a tennis player at the same time. In order to support the modelling of such facts, a data model must allow objects to have different types simultaneously. In our example, it must be possible that the same person object can be dressed with the types `employee` and `tennisplayer`. Most of the object-oriented data models do not support such role modelling. An object is considered to be an instance of a single type. The next subsection introduces a data model which separates typing from classification and supports role modelling.

3.2 The OM Model

Figure 4 shows a simple schema using the notation provided in the OM model. It models collections `Persons` and `Addresses` which are related with each other through association `have`. Cardinality constraints specify that a person has one or more addresses, while an address may be related with one or more persons. Collection `Persons` has three subcollections, namely `Employees`, `Students` and `TennisPlayers`. The integrity constraint `cover` demands that each person is also classified in at least one of the subcollections, or, in other words, that collections `Employees`, `Students` and `TennisPlayers` cover collection `Persons`.

In the shaded region of each box denoting a collection, the member type of the collection is given. Collection `Persons`, for example, has a member type `person`, which means that each object contained in collection `Persons` must be dressed with type `person`.

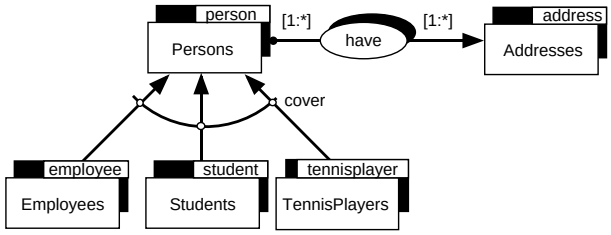

Fig. 4. A schema in the OM model

The Data Structures. The OM model supports collections of objects and associations. A collection of objects contains objects dressed with the same member type. A collection is itself an object and thus has an object identifier, and it is possible to have collections of collections. Figure 5 shows how role modelling is supported in the OM model. An object may be classified in two different collections simultaneously, for example in a collection `Employees` and a collection `TennisPlayers`. Depending on the collection through which we view this object, we see a different representation of it. Looking at an object through collection `Employees` shows attribute values such as a name, a birthdate and an employment number plus salary as defined in type `employee`. Viewing the same object through collection `TennisPlayers`, however, shows attribute values according to type `tennisplayer`.

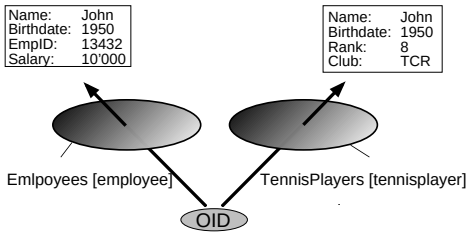

Fig. 5. Viewing an object in different roles

An association allows objects in collections to be related with each other. An association is a binary collection since it contains pairs of object identifiers where one object identifier refers to an object in a source collection while the other one refers to an object in a target collection. Associations are objects as well and thus have an object identifier.

Algebra. The operational model of OM is based on a generic *collection algebra*. A list of operations supported in OM is given in Table 1. The algebra includes operations such as *union*, *intersection*, *difference*, *cross product*, *flatten*, *selection*, *map* and *reduce* as well as special operations over binary collections such as *domain*, *range*, *inverse*, *compose* and *closure*.

Table 1. Algebra operations supported in the OM model

Algebra Operation	**Signature**
Union	$\cup : (coll[Type_1], coll[Type_2]) \rightarrow coll[Type_1 \sqcup Type_2]$
Intersection	$\cap : (coll[Type_1], coll[Type_2]) \rightarrow coll[Type_1 \sqcap Type_2]$
Difference	$- : (coll[Type_1], coll[Type_2]) \rightarrow coll[Type_1]$
Cross Product	$\times : (coll[Type_1], coll[Type_2]) \rightarrow coll[(Type_1, Type_2)]$
Flatten	$\texttt{flatten} : coll[coll[Type]] \rightarrow coll[Type]$
Selection	$\sigma : (coll[Type], Type \rightarrow \texttt{boolean}) \rightarrow coll[Type]$
Map	$\texttt{map} : (coll[Type_1], Type_1 \rightarrow Type_2) \rightarrow coll[Type_2]$
Reduce	$\texttt{reduce} : (coll[Type_1], (Type_1, Type) \rightarrow Type, Type) \rightarrow Type$
Domain	$\texttt{domain} : coll[(Type_1, Type_2)] \rightarrow coll[Type_1]$
Range	$\texttt{range} : coll[(Type_1, Type_2)] \rightarrow coll[Type_2]$
Inverse	$\texttt{inv} : coll[(Type_1, Type_2)] \rightarrow coll[(Type_2, Type_1)]$
Compose	$\circ : (coll[(Type_1, Type_2)], coll[(Type_3, Type_4)]) \rightarrow coll[(Type_1, Type_4)]$
Closure	$\texttt{closure} : coll[(Type_1, Type_2)] \rightarrow coll[(Type_1, Type_2)]$

The algebra operations are given specifying the input arguments and the results along with their types. Collections which list two types, for example, $coll[(Type_1, Type_2)]$ are binary collections, containing objects of the form $<\texttt{oid}_1, \texttt{oid}_2>$, where $\texttt{oid}_1$ refers to an object of type $Type_1$ and $\texttt{oid}_2$ to an object of type $Type_2$.

For set operations, the notions of *least common supertype* and *greatest common subtype* are used to determine the member type of the resulting collections. The *union* of two collections thus returns a collection whose type is the least common supertype of the two collections involved ($Type_1 \sqcup Type_2$). The resulting collection contains all the elements belonging to one (or both) of the argument collections. The *intersection* of two collections, on the other hand, returns a collection whose type is the greatest common subtype of the two collections involved ($Type_1 \sqcap Type_2$), containing the objects which are members of both argument collections. The type of the resulting collection of the *difference* of two collections corresponds to the type of the first argument in the operation. The resulting collection contains exactly those objects in the first argument collection which are not also members of the second one.

The *cross product* of two collections returns a binary collection $coll[(Type_1, Type_2)]$ containing all combinations of an object of the first collection with an object of the second one. The *selection operation* has as arguments a collection and a function which selects objects from the collection. The result is a subset of

the argument collection. The *map operator* applies a function with a signature $Type_1 \rightarrow Type_2$ to each object in the argument collection and returns a collection of objects of type $Type_2$. The *flatten operator* takes a collection of collections of the same member type and flattens them to a collection of type $Type$. The *reduce operator* allows the execution of aggregate functions over a collection of values. It has three arguments – the collection $coll[Type_1]$ over which the aggregate function is executed, the aggregate function itself with a signature $(Type_1, Type) \rightarrow Type$ and an initial value of type $Type$.

The OM model also supports special operations over binary collections. Some of them are listed in the second part of Table 1. The domain operation takes a binary collection and forms a collection of all the objects that appear as the first element of a pair of objects $<\texttt{oid}_1, \texttt{oid}_2>$ belonging to the binary collection, whereas the range operation returns a collection containing the second elements of such pairs. The inverse operation swaps the first and the second element of the pairs contained in the argument binary collection and returns them in a new binary collection. The composition operation combines those binary objects of the two argument collections where the second object in the first binary object $<\texttt{oid}_1, \texttt{oid}_2>$ appears as first object of the second binary object, $<\texttt{oid}_2, \texttt{oid}_3>$. The resulting collection contains binary objects, for example, $<\texttt{oid}_1, \texttt{oid}_3>$. The closure of a binary collection is the reflexive transitive closure of a relationship represented by the binary collection.

Union, intersection and difference of collections, cross product, compose, range, domain, flatten, inverse, closure simply manipulate object identifiers. Exceptions are the selection, map and reduce operations. A selection operation, for example, might access attribute values of the objects in discourse to select the desired ones.

Constraints. The OM model supports constraints such as the subcollection, cover, disjoint, intersection and cardinality constraints. The subcollection constraint demands that each object in a collection is also member in its supercollections. The disjoint, cover and intersection constraints over collections restrict the form of relationship between supercollections and their subcollections. The disjoint constraint demands that a set of subcollections do not share a single object of their common supercollection, whereas the cover constraint demands that each object of the supercollection appears in at least one of the subcollections involved in the constraint. The intersection constraint relates a subcollection with its supercollections such that all the common objects of the supercollections are also members of the subcollection. The cardinality constraints are used to restrict how often an object may be contained in an association.

These constraints can be evaluated again by simply manipulating object identifiers. The constraint themselves are also objects.

4 OMS/Java

OMS/Java (Object Model System for Java) [Mes97] can be considered as an object-oriented database management system, respectively, as an object-oriented framework for the Java environment supporting the OM generic data model presented in Sect. 3.2. There exist also other instantiations of the OM model for different environments such as OMS/Prolog [NW97] and OMS/Objectivity [Pro97]. OMS/Java can be characterised by three main features:

- The OMS/Java core system is extensible in the form described in Sect. 2.2
- OMS/Java can either serve as an *Object-Oriented Database Management System* (OODBMS) or as an *Application Framework*.
  Used as an OODBMS, OMS/Java can be, for example, a server component in a client/server environment. In the other case, a developer can use the OM model to analyse and design the application domain and then use the OMS/Java framework for implementing the applications. Hence, no mapping is necessary between the resulting design model and the implementation.
- Using OMS/Java for managing instances of Java classes is easy and does not cause any major redesign of existing class hierarchies.

Figure 6 shows a scenario where several client applications are connected to the same OMS/Java database server.

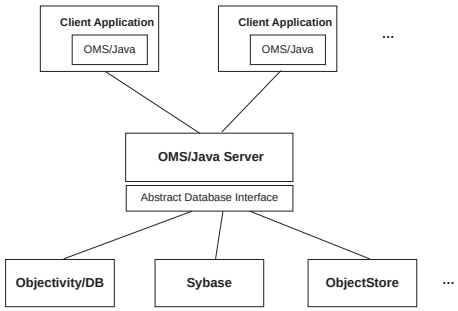

Fig. 6. OMS/Java: a multi-purpose system

Note that in Fig. 6 the OMS/Java application framework is part of all client applications, and that the storage management component of the OMS/Java server is exchangeable. This means that one can use as a storage management system either an existing database management system such as *Objectivity/DB* or a persistant Java system such as *ObjectStore PSE for Java*.

In the following sections, we describe those concepts of the OMS/Java system which are important to understand the basic ideas behind the framework as well as the decisions made to facilitate system extensibility.

4.1 Core Architecture

The OMS/Java system is divided into three main components:

- *The Configuration Component*
  The configuration component consists of all those parts of the system that are exchangeable and extensible, including:
  - Object Identifier Manager
  - Algebra Operations
  - Constraints
  - Query Language
  - Data Definition Language
  - Data Manipulation Language
  - Data Structures (such as Hashtables) and Data Access Algorithms
- *The Core System*
  The core system manages all the data structures needed for supporting the OM model. A description of some of these structures is given in Sect. 4.2. Further, special design patterns have been used to connect the core system to the other components (Sect. 4.3). Finally, all features of the system are available through an *Application Programming Interface* (API).
- *The Storage Management Component*
  The storage management component is responsible for not only the persistence of all data, but also for *Transaction Management*, *Concurrency Control*, *Recovery* and *Security* which includes *Access Control* and *Authentication*. Since this component is actually designed in such a way that it can be connected to other database management systems, most of these tasks are passed on those systems.

4.2 Basic Concepts

OMS/Java supports the core object model OM (Sect. 3.2) through the notion of *OM Objects*, *Collections* and *Constraints*.

OM Objects. An object in the OM model does not represent the same entity as a Java object which is an instance of a Java class. An *OM object* has a unique object identity during its whole lifetime. This differs from the way object references are handled in Java. A reference to an object in Java is only unique as long as the object is loaded in the Java Virtual Machine. Consider, for example, that you store a Java object in a file and then read it back again from that file. Reading from the file means that the Java Virtual Machine creates a new Java object with a new reference to it so it is not possible to decide whether the object you stored before is the same as the one you retrieved. To solve this problem, we introduced a special class *ObjectID*. Every instance of this class represents an unique persistent object identifier and is related to exactly one *OM object*.

Further, depending on the context in which an object is accessed (e.g. accessing an object through a specific collection), the object changes its role. This implies that more than one type can be associated with an *OM object*. This is not supported by the Java environment since it is not possible to change the type definition of a Java class instance during run-time.

As is shown in Fig. 7, we solved this problem by splitting up an *OM object* into the two Java classes `OMObject` and `OMInstance` (which is actually a Java interface class). Additionally, meta information about *OM objects* needed by the OMS/Java system are defined in instances of the Java class `OMType`.

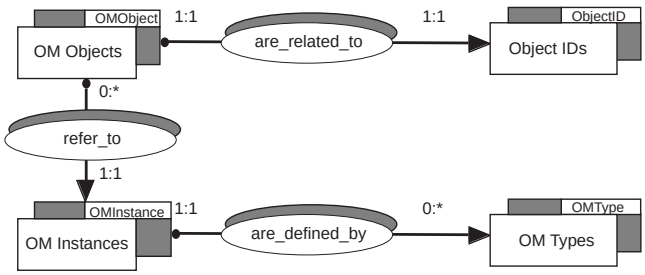

Fig. 7. OM Objects

Another approach would have been to implement one's own dynamic type system as has been done, for example, in the OMS/Prolog system [NW97]. But, since we wanted to be able to use standard Java classes in our system, this approach was not suitable.

Using Java classes involves two steps: first the meta information has to be defined in the schema (see also Sect. 5.2) such as

```
type person
(
  name: String;
);
```

Then, for all the types defined in the schema, a corresponding Java class has to be specified. This can be done in three ways:

- by extending the class `OMSimpleInstance` which provides all functionality needed by the system such as meta information.
- by implementing the interface `OMInstance`, or
- by using the adapter class `OMAdapter`.
  An adapter or wrapper "converts the interface of a class into another interface which clients expect. It lets classes work together that could not otherwise because of incompatible interfaces." [GHJV95]. Hence, the `OMAdapter` class makes it possible to use most of the Java classes without modifications.

The following class definition is an example for the first approach:

```
package demo;
import omJava.*;
public class Person extends OMSimpleInstance {
    public String name;
}
```

Collections. An *OM object* can be classified according to different criteria by inserting it into collections. Removing an object from a collection and inserting it into another one simply changes its role. So, if an *OM object* is stored in a collection then it gains automatically the member type of that collection. Further, a collection itself is an *OM object* and can therefore also be classified by inserting it into other collections. A rich set of algebra operations over collections (see Sect. 3.2) together with different kinds of collections such as sets, bags and sequences are also provided by the OMS/Java system.

Constraints. The various structural constraints such as *subcollection*, *cover*, *disjoint* and *intersection* between collections (Sect. 3.2), as well as the cardinality constraints for associations, are specified as *Global Constraints* in OMS/Java and will be checked at commit time of a transaction. *Local Constraints*, on the other hand, are related to a single *OM object*. For example, the constraint that all objects in a collection must be of the same type is defined as a *Local Constraint* and will be checked every time the collection is updated. *Local Constraints* are in fact specialisations of *Triggers*. A *Trigger* is related to one or more *OM objects* and is activated as soon as the trigger condition holds.

Note that *Global Constraints*, *Triggers* and *Local Constraints* are also treated as *OM objects* in OMS/Java.

4.3 Design Issues

To take full advantage of object-oriented concepts such as reuse and extensibility, it is important to use the appropriate design patterns.

> "Each pattern describes a problem which occurs over and over again in our Environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." [GHJV95]

Using design patterns not only increases the quality of the software product, but also makes it possible to build systems that are flexible enough to support requirements not known while implementing it. The following four design patterns were used to achieve extensibility of our system:

Abstract Factory. The main construct for system parameterisation, respectively, configuration is the *Abstract Factory*. It defines a set of abstract parameters needed at run-time by an application or, in our case, the OMS/Java core system. Such parameters include among others the algebra used for operations over collections, the object identifier generator, the query language and data definition language parsers and the various data structures such as hashtables (Sect. 4.4).

Abstract Classes. An *Abstract Class* defines a common interface for its subclasses. It defers some or all of the implementation to the subclasses. Hence, an abstract class cannot be instantiated, but forces the developer of the subclasses to at least implement important methods needed for the overall system functionality. For instance, there is a data structure `Container` which is used to store the elements of a collection. Inserting elements into such a container depends on whether a collection is a set, bag or sequence. For this purpose the *abstract* class `InsertionStrategy` defines a method `insert` which provides a common interface for inserting objects into containers and which must be specified by its subclasses:

```
abstract public class InsertionStrategy {
  ...
  abstract public void insert(Container container, Object object);
}
```

Strategy. The *Strategy* design pattern defines a family of algorithms. By encapsulating all these algorithms into a strategy object, they become interchangeable and independent from a specific client object. For example, the algorithms to insert objects into collections are defined as insertion strategies making it possible to vary the insertion algorithm depending on whether a collection is a set, a bag or a sequence.

```
public class SetStrategy extends InsertionStrategy {
  ...
  public void insert(Container container, Object object) { ... }
}

public class BagStrategy extends InsertionStrategy {
  ...
  public void insert(Container container, Object object) { ... }
}
       ...
```

Bridge.

> "A *Bridge* design pattern decouples an abstraction from its implementation so that the two can vary independently." [GHJV95]

For example, two bridges have been used to separate the algebra operations and the underlying data structures from the collection object. Therefore, it is possible to exchange the data structures and/or the algebra without having to modify the interface of the collection object. This implies that the collection object must be initialised by these two bridges. The following example illustrates the use of these bridge classes. Suppose we want to find all objects of which the attribute `name` is equal to the value `Smith` using the algebra operation `select`:

```
OMCollection result = collection.select("name", "Smith");
```

This invokes the method `select` of the `OMCollection` object which calls the corresponding method of the `Algebra` object. This method then performs the selection and inserts the matching objects into the result collection by invoking the `insert` method of this collection. That method again redirects the insertion by calling the `insert` method of the `InsertionStrategy` object which itself invokes the method `insert` of the `Container` object.

```
public class OMCollection extends OMSimpleInstance {
  protected Algebra    algebra;   // decouples algebra
  protected Container  container; // decouples data structure
  protected OMInsertionStrategy  strategy;
  ...
  public OMCollection select(String selection, Object value) {
    return algebra.select(this, selection, value);
  }

  public OMCollection insert(Object object) {
    strategy.insert(container, object);
  }
}

public class OMAlgebra implements Algebra {
  ...
  public OMCollection select(OMCollection collection,
                             String selection, Object value) {
    OMCollection result = ... // perform select operation over
                              // 'collection'
    result.insert(object.key()); // insert matching objects into
                                 //result collection
    return result;
  }
}

public class Container {
  ...
  public void insert(Object object) {...}
}
```

4.4 The Link between the Core System and the Configuration Component

The *Core System* is connected to the *Configuration Component* by an instance of the Java class `Factory` as shown in Fig. 8. This `Factory` class corresponds to the design pattern *Factory* and is an important concept for making a system extensible. The `Factory` class provides various methods such as `getNextObjectID()`, `getOMLparser()`, `getCollectionAlgebra()` and `getCollectionContainer()`. The method `getNextObjectID()`, for instance, returns an instance of class `ObjectID` which represents a unique object identifier, whereas `getCollection`

`Algebra()` returns an instance of `Algebra` which can be used for evaluating algebra operations on collections. All parts of the *Configuration Component* are obtained by calling the appropriate method of the `Factory` instance which, for its part, is given as a parameter while initialising the *Core System*. By subclassing the class `Factory` and overriding methods such as `getNextObjectID()`, it becomes possible to provide to the core system, for instance, different data structures such as object identifiers.

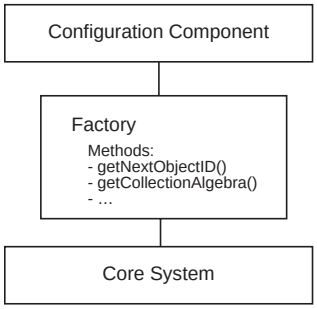

Fig. 8. The link between the core system and the configuration component

5 From OMS/Java to TOMS/Java

Extending OMS/Java with new features means in fact to extend, respectively, to replace parts of the *Configuration Component* which has been described in Sect. 4.1. The following list gives a summary of the main tasks that are involved when implementing extensions to OMS/Java:

- Extending the syntax of the *Data Definition Language*, the *Data Manipulation Language* and *Query Language*, and changing the corresponding parsers, i.e. implementing the new features in subclasses of the standard parsers.
- Subclassing the standard object identifier class
- Implementing new algebra operations in a subclass of the algebra class
- Extending the various constraint classes
- Defining a subclass of the class `Factory` which is used as a link between the *Core System* and the *Configuration Component*

The following sections describe in more detail how OMS/Java has been extended to support the *Temporal Object Data Model* TOM [SN97c, SN97a].

5.1 The Temporal Object Data Model TOM

TOM is based on the generic object data model OM [Nor93] and exhibits many of the features found in various temporal object-oriented models, e. g. [RS91,

WD93, KS92, BFG96], but in a more generalized form. For example, we timestamp not only data values, but also collections of objects and the associations containing the temporal relationships between objects. Anything considered to be an object in our model may be timestamped, even metadata such as constraints, types or databases. This allows the specification of a lifespan for each instance of such a construct. Hence, *TOM* is based on *object-timestamping*, i.e. the *object identifiers* are extended with a timestamp. A timestamp is a *temporal element* [Gad86] – a set of time intervals, closed at the lower bound and open at the upper bound. Additionally, TOM supports a temporally complete query lanaguage [BJS95] and temporal constraints. Temporally complete means that *all* operations found in the non-temporal data model have to be generalized into temporal operations, including, for example, the usually neglected *set difference* operation. Additionally, temporal comparison predicates have to be supported allowing expressions such as *a before b*. Temporal constraints generalise their non-temporal counterparts with temporal semantics. A constraint no longer has to hold over a single database state, but rather over the set of those database states contained in the constraint's lifespan.

5.2 Language Extensions

This section describes the extensions made to the syntax of the various languages (DDL, DML and QL) by means of an example:

Suppose we want to build an information system based on the schema given in Fig. 4. For the sake of simplicity, we model only the collections `Persons`, `Employees` and `Students`.

Defining the Schema. The following schema has been specified using *OML*, the data definition language of OMS/JAVA:

```
person = demo.Person; // links type 'person' to the Java class
                      // 'demo.Person'
employee = demo.Employee;
student  = demo.Student;

type person lifespan {[1980/1/1-inf)}
(
  name: String;
);

type employee subtype of person lifespan {[1980/1/1-inf)}
(
  empID: Integer;
  salary: Integer;
);
```

```
type student subtype of person lifespan {[1982/1/1-inf)}
(
  studID: Integer;
);

collection Persons    : set of person lifespan {[1980/1/1-inf)};
collection Employees  : set of employee lifespan {[1980/1/1-inf)};
collection Students   : set of student lifespan {[1982/1/1-inf)};

constraint: Employees subcollection of Persons lifespan
                                                {[1980/1/1-inf)};
constraint: Students subcollection of Persons lifespan
                                                {[1982/1/1-inf)};
```

Three categories of objects are defined in this schema: type, collection and constraint objects. Further, the *Lifespan* of *TOM* objects, specifying the period of time in which an object is valid, is set by the corresponding `lifespan` parameter.
For supporting the notion of *Lifespans*, we first had to extend the syntax of *OML* by the following definitions given in *EBNF* [Wir96]:

```
lifespan = "{" interval {"," interval} "}".
interval = "[" date "-" date ")".
date     = (year "/" month "/" day ["~" hour ":" minute ":"
                                                second]) | "inf".
```

Next, the *OML* parser had to be adapted to the new syntax. Since all parsers in our system are part of the *Configuartion Component* and not of the *Core System*, no changes are necessary in the *Core System* classes for supporting, for example, a new syntax. Only the corresponding parser has to be changed which can be achieved by either replacing the existing parser or by extending it. In most cases, especially if there are only minor changes of the syntax, extending an existing parser will be possible because we designed the standard parsers in such a way that they invoke special methods during the syntax analysis. For example, after the *OML* parser has parsed a type, collection or constraint definition such as `type person`, the method `parseRestOfLine(...)` is called. In the case of the standard parser, this method just returns without performing any action. Hence, for supporting the new *Lifespan* syntax we just extended the Java class `OMLparser` and overrode the `parseRestOfLine(...)` method.

Creating Objects. One possibility for creating objects and inserting them into collections is to import a file in which the data is described in terms of *Data Manipulation Language* statements:

```
create object paul lifespan {[1969/1/1-inf)}

dress object paul as demo.Person during {[1969/1/1-1975/6/1)}
values (name = "Paul Jones")
```

```
dress object paul as demo.Employee during {[1986-1996)}
values (empID = 23; salary = 2000)

dress object paul as demo.Student during {[1993-1995)}
values (studID = 2674)

insert object paul into collection Persons during
                                        {[1985/1/1-1996/1/1)}
```

The `create object` statement defines an *OM Object* the alias of which is `paul`. Its lifespan is set to the *Lifespan* parameter `lifespan [1969/1/1-inf)`. The `dress object` statements further specify that instances of the Java classes `demo.Person`, `demo.Employee` and `demo.Student` are associated to that object and that this association is valid as given by the *During* parameters. Remember that an *OM Object* can refer to more than one instance of a Java class (as shown in Fig. 7).

The `insert` statement states that the object `paul` is a member of the collection `Persons` during the time period specified by the *During* parameter `during [1985/1/1-1996/1/1)`. Note that the collection `Persons` has been already created after loading the schema.

To support the *Lifespan* as well as the *During* parameters, which are features of *TOM*, we had to extend the *DML Parser* taking the same approach as in the case of the *OML Parser*, i.e. we extended the *DML Parser* by making a subclass of the class `DMLparser` and implemented, respectively, overrode some methods.

Querying the Database. *AQL* (Algebraic Query Language) is the query language of OMS/Java. This language was originally developed for OMS/Prolog [NW97], and it is both a simple, and yet powerful language with operations over values, objects, collections and associations. As a simple example, consider that we want to know which objects are classified as both *Employees* and *Students*. This can be expressed in AQL as:

```
Employees union Students;
```

In the case of *TOM* the corresponding query contains the keyword `valid` in addition:

```
valid Employees union Students;
```

The `valid` keyword denotes that the query should be evaluated only on those objects which are valid *now*. The result of this query is presented by TOMS/Java in the form of a scrollable list. By double-clicking on one of the items in the list, the system dynamically generates a window showing all properties of the selected object. Figure 9 gives an example of such an object. As in the case of DDL and DML, we have extended the Java class `AQLparser`, in this case to support temporal queries.

Fig. 9. A person object

5.3 Changes Made in the Configuration Component

In this section, we outline the changes made in the Configuration Component that have been necessary to support *TOM*. Table 2 gives an overview of the classes which had to be extended. Instances of these classes can be obtained by calling the corresponding method of the *Factory*. Note that the term *Factory* is used in the text as a placeholder for an instance of the class `ValidTimeFactory`. This class is a subclass of the OMS/Java class `Factory` and overrides methods such as `getNextObjectID()` to support the semantic of *TOM* (Sect. 4.3).

Table 2. Java classes which had to be extended

Factory Method	OMS/Java class	TOMS/Java class
`getOMLparser()`	`OMLparser`	`TOMLparser`
`getDMLparser()`	`DMLparser`	`TDMLparser`
`getAQLparser()`	`AQLparser`	`TAQLparser`
`getNextObjectID()`	`ObjectID`	`ValidTimeObjectID`
`getInstanceList()`	`InstanceList`	`ValidTimeInstanceList`
`getCollectionAlgebra()`	`Algebra`	`ValidTimeAlgebra`
`getSubcollConstr()`	`SubcollConstr`	`ValidTimeSubcollConstr`

Object Identifiers. When a schema definition file is loaded, the core system first invokes the method `getOMLparser()` of the *Factory* to retrieve an instance of an OML parser, followed by a call to the parser method `generate(schema)` which parses the schema and generates the objects. In the case of *TOM*, this parser must be able to manage temporal information such as *Lifespans*. Hence, we made a subclass of `OMLparser` containing new methods for temporal information processing as described in Sect. 5.2.

For the schema given in Sect. 5.2, the parser creates the following Java instances: instances of `OMType` for each type definition, instances of `OMCollection` for each collection and instances of `OMSubcollConstr` for each subcollection constraint. For each of these instances, the parser also gets an `ObjectID` instance, respectively in the case of *TOM*, a `ValidTimeObjectID` instance by calling the *Factory* method `getNextObjectID()`. The class `ValidTimeObjectID` is a subclass of `ObjectID` which has been extended for storing *Lifespan* information. Finally, the parser sets the lifespans of these object identifiers to the ones given in the schema.

Object Creation. One way of creating objects is to import a file containing data defined by *Data Manipulation Language* statements as has been described in Sect. 5.2. For this purpose, the core system calls the *Factory* method `getDMLparser()` which returns an instance of the class `DMLparser`. Actually for *TOM*, the *Factory* returns a subclass of `DMLparser` which provides all methods needed for managing temporal information such as *Lifespans* (Sect.5.2).

The method `generate(inputFile)` of the parser is invoked by the core system which imports, respectively, creates the objects. For our example, the parser first creates an `OMObject` instance, the alias of which is `paul`, and then associates it to an instance of `ValidTimeObjectID` of which the lifespan is set to `[1969/1/1-inf)`. Further, the parser creates instances of the Java classes `demo.Person`, `demo.Employee` and `demo.Student` and sets the values of the different instance fields by calling the `setAttrib(...)` method of each instance. Finally, the parser relates these instances to the object `paul` by calling the method `dress(instance)` which inserts an instance into the *Instance List* of the object `paul`. The data structure *Instance List*, which is obtained by the *Factory* method `getInstanceList()` while creating an *OM Object*, is used to manage all *OM Instances* related to this object and provides methods such as `insert(OMInstance instance)`, `delete(OMInstance instance)`, `update(OMInstance instance)` and `replace(OMInstance instance, int index)`. For supporting *TOM*, we had to extend the `StdInstanceList` class and to override these methods. In our example, the *Validity* of the relationship between the object `paul` and the associated instances of `demo.Person`, `demo.Employee` and `demo.Student` is specified by the *During* parameters. These parameters are stored by the parser in the corresponding instance by calling the method `setParameter(parameter)` of the `OMInstance` class. Note that this parameter is of type `Object` making it possible to store any kind of information in an *OM Instance* object.

Collections and Algebra Operations. Collections are represented by instances of the Java class `OMCollection` and are created, for example, while loading a schema definition file by the *OML parser*. Creating such an object invokes the constructor of the `OMCollection` class. This constructor itself calls the *Factory* methods `getCollectionContainer()` which returns an instance of the class `Container` needed for managing the members of a collection, and `getCollectionAlgebra()` which returns, in the case of *TOM*, an instance of the class `ValidTimeAlgebra` providing algebra operations such as `union(...)`. Since the data structure as well as the algebra are returned by the *Factory* and are therefore not part of the `OMCollection` class which belongs to the core system, it is possible to exchange, for instance, the algebra without having to change the `OMCollection` class.

Inserting objects into collections can be done, for example, by importing a data file using the DML parser. In our example, the object `paul` is inserted into the collection `Persons`. In addition, the `during` parameter, which is a feature of *TOM*, specifies the time period during which `paul` is a member of `Persons`. No changes were necessary to support this feature because an `OMCollection`

class provides a method `setParameter(object, parameter)` which can be used to store additional information for each member object such as the *Validity* given by the `during` parameter. This information is needed by the algebra class and can be retrieved by calling the collection method `getParameter(Object object)`. Hence, the `OMCollection` class does not need to be able to manage this information and since the `parameter` is of type `Object`, the stored information is not restricted to specific semantics such as temporal information. So, for inserting the object `paul` into `Persons` the *DML parser* first invokes the method `insert(object)` of the collection to insert the object and then sets the *Validity* given by the `during` parameter by calling the collection method `setParameter(object, parameter)`.

Evaluating algebra operations simply means calling the appropriate methods of collection objects. For example, the AQL parser evaluates the *temporal* AQL query "`valid Employees union Students;`" by invoking the `union(...)` method of the `Employees` collection object:

```
employees.union(students, algebraParameter);
```

The method `union` takes two input parameters: the second collection object for the union operation and an algebra parameter which is in our case the keyword `valid`. Further, this method calls the method `union(employees, students, algebraParameter)` of the algebra instance which has been specified when the collection object has been created. Hence, only class `Algebra`, or subclasses of it such as `ValidTimeAlgebra`, must know how to handle the algebra parameter. Note that this parameter can also be of any type.

Constraints. *Global Constraints*, *Triggers* and *Local Constraints*, which have been defined in Sect. 4.2, are treated as *OM Objects* and are therefore associated to an unique object identifier which in the case of *TOM* is an instance of `ValidTimeObjectID`.

Global Constraints such as the subcollection constraint `Employees subcollection of Persons lifespan [1980/1/1-inf)` in our example are created by the *OML parser* after loading the schema definition file. The parser invokes for this purpose the *Factory* method `getSubcollConstr()` which returns, in the case of *TOM*, an instance of `OMValidTimeSubcollConstr`. The *Lifespan* parameters denote the time periods in which the constraints are valid. The `OMValidTimeSubcollConstr` class is a subclass of `OMSubcollConstr` of which various methods have been overridden for supporting *temporal* semantics. *Triggers* and *Local Constraints* are created at the same time when the objects to which they belong are initialised.

6 Conclusions

We presented a general approach for extending database management systems (DBMS) in terms of *model extensibility* and described the corresponding general

DBMS architecture. While our approach builds on ideas of *architecture extensibility*, i.e. providing architectural support in terms of services such as storage management, it also differs from it in terms of focussing on the generalisation of a core data model and system for advanced application domains.

In particular, we described the OMS/Java system which has been developed to support *model extensibilty*. As a proof of concept, we have extended OMS/Java to support the temporal object data model *TOM*. We are convinced that using the *model extensibility* approach leads to database management systems which are easily adaptable to various application domains. In the future, we want to investigate extending the system for a spatial data model and also for a version model. The latter is intended to provide an OMS/Java implementation platform for a document management system currently under development.

References

[BFG96] E. Bertino, E. Ferrari, and G. Guerrini. A Formal Temporal Object-Oriented Data Model. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology*, pages 342–356. Springer, 1996.

[BJS95] M. Böhlen, C. Jensen, and R. Snodgrass. Evaluating the completeness of TSQL2. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, pages 153–172, 1995.

[BLW88] D.S. Batory, T.Y. Leung, and T.E. Wise. Implementation Concepts for an Extensible Data Model and Data Language. *ACM TODS*, 13:231–262, September 1988.

[CDG+89] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. Vandenburg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan-Kaufmann, 1989.

[CDKK85] H.-T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug. Design and Implementation of the Wisconsin Storage System. *Software Practice and Experience*, 1985.

[Cha96] A. Chatterjee. A Framework for Object Matching in Federated Databases and its Implementation. *Journal of Intelligent and Cooperative Information Systems*, 1996.

[EN94] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Publishing Company, 1994.

[Gad86] S. K. Gadia. Toward a Multihomogeneous Model for a Temporal Database. In *Proceedings of the International Conference on Data Engineering*, pages 390–397, 1986.

[GD94] A. Geppert and K. Dittrich. Constructing the Next 100 Database Management Systems: Like the Handyman or Like the Engineer. *ACM SIGMOD Record*, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HCL+90] L.M. Haas, W. Chang, G.H. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Trans. on Knowledge and Data Engineering*, 1990.

[HS95] A. Heuer and G. Saake. *Datenbanken: Konzepte und Sprachen.* International Thomson Publishing, 1995.

[KS92] W. Käfer and H. Schöning. Realizing a Temporal Complex-Object Data Model. In *SIGMOD Conference 1992*, pages 266–275, 1992.

[MBH+] F. Maryanski, J. Bedell, S. Hoelscher, S. Hong, L. McDonald, J. Peckham, and D. Stock. The data model compiler: A tool for generating object-oirneted database systems. In *Proc. of the 1986 Intl. Workshop on Object-Oriented Database Systems.*

[Mes97] S. Messmer. The OM Model as a Framework for the Java Environment. Master's thesis, Department of Computer Science, ETH Zurich, 1997.

[Nor93] M. C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proceedings of the 12th International Conference on the ER Approach*, 1993.

[Nor95] M. C. Norrie. Distinguishing Typing and Classification in Object Data Models. In *Information Modelling and Knowledge Bases*, volume VI, chapter 25. IOS, 1995. (originally appeared in Proc. European-Japanese Seminar on Information and Knowledge Modelling, Stockholm, Sweden, June 1994).

[NW97] M. C. Norrie and A. Würgler. OMS Object-Oriented Data Management System. Technical report, Institute for Information Systems, ETH Zurich, CH-8092 Zurich, Switzerland, 1997.

[Pro97] R. Prodan. OMS/Objectivity. Master's thesis, Department of Computer Science, ETH Zurich, 1997.

[RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proceedings of the 10th International Conference on the ER Approach*, 1991.

[SC94] A. Segev and A. Chatterjee. Supporting Statistical Operations in Extensible Databases: A Case Study. In *Proc. IEEE Seventh International Working Conference on Scientific and Statistical Database Management*, Charlottesville, USA, sep 1994.

[SN97a] A. Steiner and M. C. Norrie. A Temporal Extension to a Generic Object Data Model. Technical Report 265, Institute for Information Systems, ETH Zürich, May 1997.

[SN97b] A. Steiner and M. C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *Database Systems for Advanced Applications (DASFAA)*, 1997.

[SN97c] A. Steiner and M. C. Norrie. Temporal Object Role Modelling. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, 1997.

[WD93] G.T.J. Wuu and U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 10, pages 230–247. Benjamin/Cummings Publishing Company, 1993.

[Wir96] N. Wirth. *Compiler Construction.* Addison-Wesley, 1 edition, 1996.

This article was processed using the LaTeX macro package with LLNCS style