

# Objects and Roles in the Stack-Based Approach<sup>\*</sup>

Andrzej Jodłowski<sup>1</sup>, Piotr Habela<sup>3</sup>,  
Jacek Płodzień<sup>1,2</sup>, and Kazimierz Subieta<sup>3,1</sup>

<sup>1</sup> Institute of Computer Science PAS, Warsaw, Poland

<sup>2</sup> Warsaw School of Economics, Warsaw, Poland

<sup>3</sup> Polish-Japanese Institute of Information Technology, Warsaw, Poland

andrzejj@ipipan.waw.pl

phabela@pjwstk.waw.pl

jpl@ipipan.waw.pl

subieta@ipipan.waw.pl

**Abstract.** In the paper we propose a new approach to the concept of dynamic object roles. The approach assumes, among others, that: a role is a distinguished subobject for an object; a role dynamically inherits attributes' values and methods of its parent object; objects can be accessed by their names as well as by the names of their roles. The paper focuses on implications of this concept for an object data model and for an object data store. We also explain how it fits with other modern notions from the conceptual modeling field. Finally, we discuss some issues concerning a query language for the object data model with roles.

## 1 Introduction

Since its very beginning, conceptual modeling has been evolving to answer the needs of information systems analysts and designers. A great number of concepts and ideas has been proposed in the literature and introduced into practice. Some of them have been abandoned; some are still in use. Nevertheless, the need for more and more modern, useful, effective and easy-to-use modeling concepts is growing more and more. One of the most promising of them is dynamic object roles.

Roles, like classes, may be used to classify objects, but unlike class-based classification, role-based classification may be multiple and dynamic [7]. Some of the most popular features of roles are as follows [6]: a role comes with its own properties and behavior; an object may acquire and abandon roles dynamically; an object may play different roles simultaneously; an object may play the same role several times. For example, a certain person can simultaneously be a student, a worker, a patient, a club member etc. Moreover, a person can be a student two or more times. Therefore it is more precise to say that a person *becomes* a student for some time and later he/she terminates the student role, than to

---

<sup>\*</sup> This work was partially supported by the European Commission project ICONS; project no. IST-2001-32429.

say that a student *is a* person. Similarly, a building can be an office, a house, a magazine etc.

For several years dynamic object roles have had the reputation of a notion on the brink of acceptance/rejection. On the one hand, there are many papers advocating the concept, e.g. [1,2,5,11]. On the other hand, many researchers consider applications of the concept not sufficiently wide to justify the extra complexity of conceptual modeling facilities. Moreover, the concept is totally neglected on the implementation side – as far as we know, none of the popular object-oriented programming languages or database management systems (DBMSs) introduces it explicitly. Hence, there are several special techniques of representing roles indirectly [6], for instance, as named places of relationships and as specialization/generalization.

In our opinion, despite the significant number of ideas and papers, or rather due to it, the concept of dynamic roles should be revised. The paper describes our proposal for composite objects with roles, which possess a special structure and semantics. In our research we use the *stack-based approach* (SBA) and its query language SBQL; see e.g. [4,8]. A version of roles was implemented in the prototype system Loqis [9]. Currently we are working on a prototype of an object-oriented DBMS for intelligent content management for Web applications, where we intend to implement the ideas presented here.

The rest of the paper is organized as follows. Section 2 introduces the general idea of our proposal. Section 3 discusses the differences between the concept of dynamic roles (including our proposal) and the traditional object models in programming languages and DBMSs. Section 4 presents assumptions for an object store and Section 5 discusses some issues concerning a query language in our model with roles. Section 6 summarizes the paper.

## 2 General Idea of our Proposal

The concept of dynamic object roles assumes that an object is associated with other objects (subobjects), which model its roles. In our terminology we distinguish between objects and roles: we assume that an object can contain many sub-objects called roles and a role belongs to a single object. These subobjects can be inserted and removed at run time. Object-roles cannot exist without their parent object. Deleting an object implies deleting all of its roles. Roles can exist simultaneously and independently.

A role can *dynamically inherit* from its object or from another role of this object (e.g. the specialization of a *Club\_Member* role can be a *Club\_President* role); inheritance between roles of different objects is forbidden. This kind of inheritance (known from prototype-based languages) is based on the same rule as the inheritance of class properties. A role, which inherits from a role, is sometimes called a *sub-role*; an inherited role is sometimes called a *super-role*.

A role can have its own additional properties (i.e. attributes, methods, associations etc). Two roles can contain properties with the same names, but this does not lead to any conflict, which is a fundamental difference in comparison to

the concept of multiple inheritance. For example, a person can play simultaneously the role of a research institute employee with a *Salary* attribute and also the role of a service company employee with another *Salary* attribute. These two attributes exist at the same time, but except for the name no other feature is shared, including types, semantics and business ontologies.

Associations can connect not only objects with objects, but also objects with roles and roles with roles. This makes the referential semantics clean in comparison to the traditional object models.

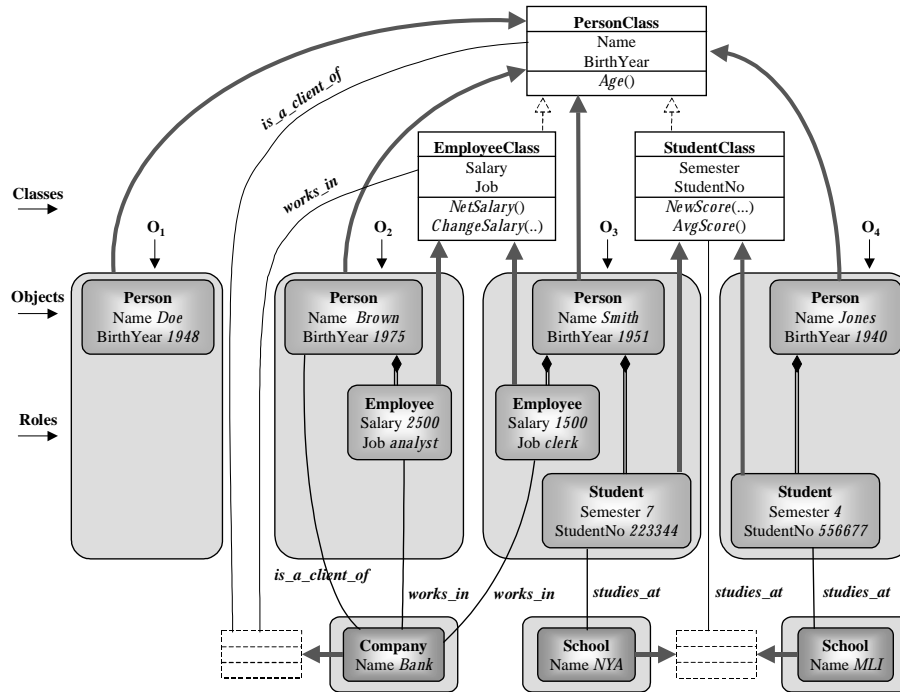


Fig. 1. Classes and objects with their roles

Fig. 1 presents an example of objects with dynamic roles and their corresponding classes in our proposal's data model:

- Each object/role is connected to its own class. The connection is shown as a gray thick solid arrow. Classes contain invariant properties of the corresponding roles, in particular, their names (the first section), attributes and their types (the second section; the attributes' types are not shown), methods (the third section), associations etc.
- A special structure for an object (shown as a gray rectangle with round corners) contains this object (*Person*) and an arbitrary number of its roles

(*Employee* and *Student*). We make no distinction between such a structure and its object.

- The *EmployeeClass* and *StudentClass* classes do not inherit statically from the *PersonClass* class. Instead, an *Employee* role and a *Student* role inherit dynamically all the properties of their *Person* super-roles, thus indirectly inherit the properties of *PersonClass*. We show this dynamic inheritance between *EmployeeClass/StudentClass* and *PersonClass* as a dashed arrow. However, if necessary, static inheritance between classes can still be applied in our model.
- Each object/role has its own name, which can be used to bind this object/role in a program or query. The presented objects can be bound through the name *Person* (each), through the name *Employee* (the objects  $O_2$  and  $O_3$ ) and through the name *Student* (the objects  $O_3$  and  $O_4$ ). The binding returns the identifier(s) of the appropriate object(s)/role(s).
- Each role is encapsulated, that is, its properties are not seen from other roles unless it is explicitly stated by a special link (shown as a double line with a black diamond end). In particular, an *Employee* role imports all the properties of its parent *Person* role. For example, if  $O_2$  is bound by the name *Person*, then the properties {*Name* “Brown”, *BirthYear* 1975} are available; however, if the same object is bound by the name *Employee*, then the properties {*Salary* 2500, *Job* “analyst”, *Name* “Brown”, *BirthYear* 1975} are available.

## 2.1 Links between Objects and Roles

As we have said, links can connect not only objects with objects, but also objects with roles and roles with roles. For example, in Fig. 1 a *works\_in* link connects a *Company* object with an *Employee* role (a similar link *studies\_at* connects *Student* roles with *School* objects). Fig. 1 shows also an *is\_a\_client\_of* link between *Person* and *Company* objects. Accessing *Person* through this link implies that any role of that object remains invisible.

Creating links between roles is an important quality for analysis and design methodologies and notations (such as e.g. UML [10]), because in several cases links should be able to lead to parts of objects, not only to entire objects. To model this situation, the methodologies suggest using aggregation/composition. Such an approach implicitly assumes that e.g. an *Employee* is a part of a *Person* similarly as an *Engine* is a part of a *Car*. Although the approach achieves the goal (e.g. we can connect the *works\_in* relationship directly to the *Employee* sub-object of *Person*), it obviously misuses the concept of aggregation, which normally is for modeling the “whole-part” situations. In our opinion, dynamic roles should be explicitly introduced into design methodologies and implementation environments.

### 3 Dynamic Object Roles vs. Classical Object-Oriented Models

Below we briefly discuss the most common features of roles advocated in the literature, which make the concept different in comparison to the classical object models:

- Multiple inheritance: Note that in Fig. 1 there is no *EmployeeStudentClass* class, which would inherit both from the *EmployeeClass* and *StudentClass* classes.
- Repeating inheritance: An object can have two or more roles with the same name; e.g. Brown can be an employee in two companies with a different *Salary* and *Job*. Such a feature cannot be expressed through the traditional inheritance or multi-inheritance concepts.
- Multiple-aspect inheritance: A class can be specialized for many aspects. Modeling tools, such as UML, cover this feature, but it is rather neglected in object-oriented programming and database models. One-aspect inheritance is not powerful enough and generally leads to multiple inheritance. Roles are a viable concept to avoid problems with this feature.
- Object migration: Roles may appear and disappear at run time without changing identifiers of other roles. In terms of the classical object models it means that an object can change its class(es) without changing its identity. This feature can hardly be available in the classical models, especially in models where the binding of objects is static.
- Referential consistency: In our model, relationships are connected to objects or roles, not to entire structures of objects and roles. Thus, for example, it is impossible to refer to Smith's *Salary* and *Job* when one navigates to its object from the *School* object. In the classical object models this consistency is enforced by strong typing, but it is problematic in untyped or weakly typed systems.
- Binding: An object can be bound by the name of any of its roles, but the binding returns the identifier of such a role rather than the identifier of the object. By definition, the binding is dynamic, because generally at compile time it is impossible to decide whether a particular object has a role with a given name.
- Typing: Because an object is seen through the names of its roles, it has as many types as it has different names for roles. Due to the dynamic nature of roles, typing must be (partly) dynamic.
- Subtyping: It can be defined in the standard manner; e.g. the *Employee* type is defined with the use of the *Person* type. However, it makes little sense to introduce a *StudentEmployee* type (cf.  $O_3$  in Fig. 1). Encapsulated roles make it unnecessary (and impossible) to mix up the properties of a *Student* object and the properties of an *Employee* object within the same single data structure.
- Aspects of objects and heterogeneous collections: A big problem with classical database object models, for instance ODMG [3], is that an object belongs

to at most one collection. This is contradictory with both multiple inheritance and substitutability. For instance, we can include an *StudentEmployee* object into the extent *Students*, but we cannot include it into the extent *Employees* (and vice versa). This violates substitutability and leads to inconsistent processing. Dynamic roles have a natural ability to model heterogeneous collections: an object is automatically included into as many collections as the types of roles it contains.

## 4 Object Store

In the following we present the formal SBA models of an object-oriented data store without and with roles and compare them. For simplicity and for making the semantics clean, we assume object relativism (i.e. each property of an object is an object too) and consider all the properties of the store (including classes) first-class citizens. The store models a program/database *state* thus does not involve types, which we consider a checking utility rather than a “materialized” property of the state.

|  |
|--|
| <p><b>Objects (O):</b><br/> <math>\langle i_1, \text{Person}, \{ \langle i_2, \text{Name}, \text{"Doe"} \rangle, \langle i_3, \text{BirthYear}, 1948 \rangle \} \rangle</math><br/> <math>\langle i_4, \text{Employee}, \{ \langle i_5, \text{Name}, \text{"Brown"} \rangle, \langle i_6, \text{BirthYear}, 1975 \rangle, \langle i_7, \text{Salary}, 2500 \rangle, \langle i_8, \text{works\_in}, i_{127} \rangle \} \rangle</math><br/> <math>\langle i_9, \text{StudentEmployee}, \{ \langle i_{10}, \text{Name}, \text{"Smith"} \rangle, \langle i_{11}, \text{BirthYear}, 1951 \rangle, \langle i_{12}, \text{StudentNo}, 223344 \rangle, \langle i_{13}, \text{Faculty}, \text{"Physics"} \rangle, \langle i_{14}, \text{Salary}, 1500 \rangle, \langle i_{15}, \text{works\_in}, i_{128} \rangle \} \rangle</math><br/> .....</p> <p><b>Classes (C):</b><br/> <math>\langle i_{40}, \text{PersonClass}, \{ \langle i_{41}, \text{Age}, (\dots \text{code of the Age method} \dots) \rangle, \dots \text{other properties of the PersonClass class} \dots \} \rangle</math><br/> <math>\langle i_{50}, \text{EmployeeClass}, \{ \langle i_{51}, \text{ChangeSalary}, (\dots \text{code of the ChangeSalary method} \dots) \rangle, \langle i_{52}, \text{NetSalary}, (\dots \text{code of the NetSalary method} \dots) \rangle, \dots \text{other properties of the EmployeeClass class} \dots \} \rangle</math><br/> <math>\langle i_{60}, \text{StudentClass}, \{ \langle i_{61}, \text{AvgScore}, (\dots \text{code of the AvgScore method} \dots) \rangle, \dots \text{other properties of the StudentClass class} \dots \} \rangle</math><br/> <math>\langle i_{70}, \text{StudentEmployeeClass}, \{ \} \rangle</math><br/> .....</p> <p><b>Root identifiers (R):</b> <math>i_1, i_4, i_9, \dots</math></p> <p><b>Inheritance relationships between classes (CC):</b> <math>\langle i_{50}, i_{40} \rangle, \langle i_{60}, i_{40} \rangle, \langle i_{70}, i_{50} \rangle, \langle i_{70}, i_{60} \rangle, \dots</math></p> <p><b>Membership of objects in classes (OC):</b> <math>\langle i_1, i_{40} \rangle, \langle i_4, i_{50} \rangle, \langle i_9, i_{70} \rangle, \dots</math></p> |
|--|

Fig. 2. An example state of the object store in the classical SBA object model

Formally, an object is a triple  $\langle i, n, v \rangle$ , where  $i$  is a unique internal object identifier,  $n$  is an external object’s name, and  $v$  is an object’s value. The value

can be atomic (e.g. “*Doe*”), can be a reference to another object, or can be a set of objects. Classes and methods are objects too. The classical SBA object store model (being some formal approximation of the object models of popular object-oriented systems) is a 5-tuple  $\langle O, C, R, CC, OC \rangle$ , where:

- $O$  is a collection of (nested) objects,
- $C$  is a collection of classes,
- $R$  is a collection of root identifiers (i.e. identifiers of objects being entries to the store),
- $CC$  is a binary relation determining the inheritance relationship between classes,
- $OC$  is a binary relation determining the membership of objects within classes.

In Fig. 2 (cf. Fig. 1) we present a simple example of an object store built in accordance with this definition (for simplicity we omit links).

Although the model seems to be natural and formally simple, there are several features which make it inconvenient, especially for defining a query language. The model results in anomalies with multiple, multiple-aspect and repeating inheritance. In the case of name conflicts it inevitably violates substitutability. Moreover, the model implies some problems with binding. For instance, suppose that a query contains the name *Person*, which has to be bound to the stored objects. According to the substitutability principle, it should be bound not only to  $i_1$ , but also to  $i_4$  and  $i_9$ . However, these objects have the names *Employee* and *StudentEmployee*, hence the binding is not straightforward. The binding rules must “know” that *PersonClass* defines objects named *Person*, is specialized by *EmployeeClass*, *StudentClass* and *StudentEmployeeClass*, and objects of these classes, independently of their current name, should be bound to the name *Person*. This information is not present in the given store model and must be introduced by some additional features.

The issue becomes even more problematic for weakly typed systems and/or dynamic binding. A similar problem concerns the semantics of references (in general, properties specific for a given subclass). For instance, if one accesses *Employee* objects through the name *Person*, then the *works\_in* references should not be accessible.

To remove these disadvantages and include roles, we propose a modification to this store model. Its new version is defined as a 6-tuple  $\langle O, C, R, CC, OC, OO \rangle$ , where:

- $C, CC$  are defined as before,
- $O, R, OC$  are defined both for objects and roles,
- $OO$  is a binary relation determining dynamic inheritance between objects and roles.

The  $OO$  relation defines two functional aspects. On the one hand, the relation determines which objects/roles are inherited by roles. On the other hand,  $OO$  fixes the semantics of manipulating objects with roles. In particular, copying

an object implies “isomorphic” copying of all of its roles, and deleting an object implies deleting all of its roles. Deleting a role implies recursive deleting all of its sub-roles, but not its super-role. *OO* is a pure hierarchy (each role has at most one super-role; no cycles).

In Fig. 3 we present an object store built in accordance with the new definition. Note that the component *CC* is empty in this case (cf. Section 2).

The model does not imply problems with multiple inheritance. Because each role is an independent encapsulated entity, no name conflict is possible. The model also clearly shows the reason for multiple inheritance anomalies in the classical object models: they are caused by the fact that properties of different classes (perhaps incompatible) are not encapsulated, but mixed up in a single environment.

|  |
|--|
| <p><b>Objects and roles (O):</b><br/> &lt; <math>i_1</math>, <i>Person</i>, { &lt; <math>i_2</math>, <i>Name</i>, "Doe" &gt;, &lt; <math>i_3</math>, <i>BirthYear</i>, 1948 &gt; } &gt;<br/> &lt; <math>i_4</math>, <i>Person</i>, { &lt; <math>i_5</math>, <i>Name</i>, "Brown" &gt;, &lt; <math>i_6</math>, <i>BirthYear</i>, 1975 &gt; } &gt;<br/> &lt; <math>i_7</math>, <i>Person</i>, { &lt; <math>i_8</math>, <i>Name</i>, "Smith" &gt;, &lt; <math>i_9</math>, <i>BirthYear</i>, 1951 &gt; } &gt;<br/> &lt; <math>i_{13}</math>, <i>Employee</i>, { &lt; <math>i_{14}</math>, <i>Salary</i>, 2500 &gt;, &lt; <math>i_{15}</math>, <i>works_in</i>, <math>i_{127}</math> &gt; } &gt;<br/> &lt; <math>i_{16}</math>, <i>Employee</i>, { &lt; <math>i_{17}</math>, <i>Salary</i>, 1500 &gt;, &lt; <math>i_{18}</math>, <i>works_in</i>, <math>i_{128}</math> &gt; } &gt;<br/> &lt; <math>i_{19}</math>, <i>Student</i>, { &lt; <math>i_{20}</math>, <i>StudentNo</i>, 223344 &gt;, &lt; <math>i_{21}</math>, <i>Faculty</i>, "Physics" &gt; } &gt;<br/> .....</p> <p><b>Classes (C):</b><br/> &lt; <math>i_{40}</math>, <i>PersonClass</i>, { &lt; <math>i_{41}</math>, <i>Age</i>, (...code of the Age method...) &gt;, ...other properties of the PersonClass class... } &gt;<br/> &lt; <math>i_{50}</math>, <i>EmployeeClass</i>, { &lt; <math>i_{51}</math>, <i>ChangeSalary</i>, (...code of the ChangeSalary method...) &gt;, &lt; <math>i_{52}</math>, <i>NetSalary</i>, (...code of the NetSalary method...) &gt;, ...other properties of the EmployeeClass class... } &gt;<br/> &lt; <math>i_{60}</math>, <i>StudentClass</i>, { &lt; <math>i_{61}</math>, <i>AvgScore</i>, (...code of the AvgScore method...) &gt;, ...other properties of the StudentClass class... } &gt;<br/> .....</p> <p><b>Root identifiers (R):</b> <math>i_1, i_4, i_7, i_{13}, i_{16}, i_{19}, \dots</math><br/> <b>Inheritance relationships between classes (CC):</b> <i>Empty.</i><br/> <b>Membership of objects/roles in classes (OC):</b> &lt; <math>i_1, i_{40}</math> &gt;, &lt; <math>i_4, i_{40}</math> &gt;, &lt; <math>i_7, i_{40}</math> &gt;, &lt; <math>i_{13}, i_{50}</math> &gt;, &lt; <math>i_{16}, i_{50}</math> &gt;, &lt; <math>i_{19}, i_{60}</math> &gt;, ...<br/> <b>Inheritance between objects and roles (OO):</b> &lt; <math>i_{13}, i_4</math> &gt;, &lt; <math>i_{16}, i_7</math> &gt;, &lt; <math>i_{19}, i_7</math> &gt;, ...</p> |
|--|

**Fig. 3.** An example state of the object store in the SBA object model with roles

Moreover, the model does not imply the mentioned before problems with binding. Note that the identifier of each role belongs to the root identifiers *R*. Hence the name *Person* will be bound to  $i_1, i_4, i_7$ . The binding concerns only the *Person* objects; other objects and roles are invisible. Similarly, the name *Employee* will be bound to  $i_{13}$  and  $i_{16}$ , but after this binding the corresponding *Person* objects ( $i_4$  for  $i_{13}$  and  $i_7$  for  $i_{16}$ ) become visible, according to the *OO*

relation. Thus, for instance, *Employee.Name* and *Employee.Age()* are correct expressions. Similarly when the name *Student* is being bound.

The model is also consistent concerning references. For example, when the name *Person* is being bound, the *works\_in* references are unavailable, because the *Employee* roles are invisible. This property holds independently of whether a system is strongly typed or untyped.

## 5 Query Language

The proposed model with roles requires modifying some elements of the “standard” SBA and the SBQL query language. Due to lack of space, the discussion concentrates only on some aspects of the issue.

Several mechanisms existing in the model without roles, for example binding and auxiliary names, remain the same for the new model. Among those that need modifying is an *environment stack*, which now can store binders for (root) roles. Moreover, new scopes pushed onto it by non-algebraic operators can contain sections opened for roles, sub-roles, sub-subroles etc.

After minor modifications, the well-known technique of casting has potential to be even more powerful than in the classical model. For instance, one may need to make an explicit conversion between a role and its object, between different roles of an object, or between an object and one of its roles. The syntax of such a cast operator is as follows:

*(name) query*

where *name* is the name of an object or a role, and *query* returns identifiers of objects or roles.

If *query* returns objects’ identifiers, then the operator returns the identifiers of *name* roles within those objects. If *query* returns roles’ identifiers, then the operator returns the identifiers of their objects (for *name* objects) or the identifiers of other roles within the same objects (for *name* roles). If an object has no role named *name*, then the result is empty.

The following examples illustrate it. Suppose that we want to receive *Employee* roles for Brown.

*(Employee) (Person where Name = “Brown”)*

The evaluation of the subquery *(Person where Name = “Brown”)* returns a collection of identifiers of *Person* objects for whom the value of the *Name* attribute is “Brown”. Then the *(Employee)* operator converts each of these identifiers into the identifier(s) of Brown’s *Employee* role(s); if an object has no *Employee* role, then the operator returns null. The result is a collection of the *Employee* roles’ identifiers; nulls are ignored.

In another example we get those of students who work (i.e. who are employees):

*(Student) Employee*

The subquery *Employee* returns the identifiers of *Employee* roles in all objects in a store. Then the (*Student*) operator converts each of these identifiers into the identifiers of the appropriate *Student* roles.

Similarly we can introduce a Boolean operator testing the presence of a given role within an object. Another operator could return the names of the roles that are currently in an object. Such operators support the generic programming technique.

## 6 Summary

In the paper we have presented an object model with dynamic roles, which we consider an alternative to the classical database object models, such as the ODMG object model. Dynamic roles can support conceptual models of many applications and, in comparison to the classical models, do not lead to anomalies and disadvantages of multiple, repeating and multi-aspect inheritance. An advantage of the model with dynamic roles is conceptual clarity concerning the level of an object store and the level of a query language, including among others data naming, scope control, binding names and casting. The model leads to some new concepts for a schema definition language and metadata management; this issue requires further research.

## References

1. Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An Object Data Model with Roles. Proc. of VLDB Conf. (1993) 39-51
2. Bertino, E., Guerrini, G.: Objects with Multiple Most Specific Classes. Proc. of ECOOP Conf. Springer LNCS 952 (1995) 102-126
3. Cattel, R.G.G., Barry, D.K. (Eds.): Object Data Management Group: The Object Database Standard ODMG, Release 3.0. Morgan Kaufmann Publishers (2000)
4. Płodzień, J., Kraken, A.: Object Query Optimization through Detecting Independent Subqueries. Information Systems. 25(8) (2000) 467-490
5. Richardson, J., Schwarz, P.: Aspects: Extending Objects to Support Multiple, Independent Roles. Proc. of SIGMOD Conf. (1991) 298-307
6. Steimann, F.: On the Representation of Roles in Object-Oriented and Conceptual Modeling. Data & Knowledge Engineering. 35(1) (2000) 83-106
7. Steimann, F.: Role = Interface: A Merger of Concepts. Journal of Object-Oriented Programming. 14(4) (2001) 23-32
8. Subieta, K., Kambayashi, Y., Leszczyłowski, J.: Procedures in Object-Oriented Query Languages. Proc. of VLDB Conf. (1995) 182-193
9. Subieta, K., Missala, M., Anacki, K.: The LOQIS System. Description and Programmer Manual. Institute of Computer Science, Polish Academy of Sciences. Report 695. Warsaw, Poland (1990)
10. OMG Unified Modeling Language Specification. Version 1.4. The Object Management Group (September 2001) <http://www.omg.org>
11. Wong, R.K.: Heterogeneous and Multifaceted Multimedia Objects in DOOR/MM: A Role-Based Approach with Views. Journal of Parallel and Distributed Computing. 56(3) (1999) 251-271