The VLDB Journal (1997) 6: 73–96

A database model for object dynamics

M.P. Papazoglou[1], **B.J. Krämer**[2]

[1]Tilburg University, INFOLAB, P.O. Box 90153, 5000 LE Tilburg, The Netherlands; e-mail: mikep@kub.nl
[2]FernUniversität Hagen, D-58084 Hagen, Germany; e-mail: bernd.kraemer@fernuni-hagen.de

Edited by D. McLeod / Received March 1994 / Accepted January 1996

Abstract. To effectively model complex applications in which constantly changing situations can be represented, a database system must be able to support the runtime specification of structural and behavioral nuances for objects on an individual or group basis. This paper introduces the role mechanism as an extension of object-oriented databases to support unanticipated behavioral oscillations for objects that may attain many types and share a single object identity. A role refers to the ability to represent object dynamics by seamlessly integrating idiosyncratic behavior, possibly in response to external events, with pre-existing object behavior specified at instance creation time. In this manner, the same object can simultaneously be an instance of different classes which symbolize the different roles that this object assumes. The role concept and its underlying linguistic scheme simplify the design requirements of complex applications that need to create and manipulate dynamic objects.

Key words: Object-oriented database systems – Dynamic object re-classification – Object role model – Dynamic class hierarchy – Object migration

1 Background

Object-oriented data models possess the ability to represent many different complex types of data and their relationships with depth and precision. As a result, existing object-oriented database systems are employed successfully in areas which require performing manipulations on large collections of complex objects.

To model objects in a particular application domain, object-oriented database systems rely on the class concept. All domain objects are pre-classified and assigned to a single class as its instances. All objects of a certain type have exactly the same set of state variables and methods capturing their structure and behavior, respectively, and are treated strictly uniformly. Once an object is instantiated and populates a class, the only changes permissible are changes to its state variables. This preserves the uniformity of the entire set of objects contained in that specific class. Should the need arise for schema changes, these are applied to the schema classes and have to be propagated to all the objects contained in the classes under update. The restructuring of objects in consequence of a schema change is necessary to preserve consistency between the type associated with each class and the structure and behavior of the class member objects.

This traditional class-instance relationship requires distinguishing statically between the schema elements that are intended to describe a common structure and behavior, namely classes, and those that are expected to be idiosyncratic, viz. the objects. During the development phase of a database application the designer can often foresee commonalities between different parts of the application, leading to a desire to share structure and behavior between those similar parts. In several situations it is, however, highly beneficial for a system to have the ability to attach idiosyncratic behavior to an individual object or a set of objects within one or more classes at a later stage. For instance, consider Fred an `Engineer` object. Fred may be promoted to the level of a principal engineer; hence this object should dynamically acquire the properties of class `PrincipalEngineer` (and become an instance of this class), while also retaining the properties of an `Engineer`. At some point in his life Fred is first classified as an `Engineer`. Later through some process, Fred is re-classified as a `PrincipalEngineer`. Yet at another point, Fred may become a `MemberOfTheBoard`. This behavior may continue until retirement is reached or Fred becomes unemployed! Figure 1 shows the linear succession of transitions for an object called Fred during its lifespan. This figure shows that Fred who started his professional career as an `Engineer` object at time t_1 was first transformed to a `PrincipalEngineer` object at time t_2, and then to a `MemberOfTheBoard` object at time t_3 until he became unemployed at time t_4.

Unfortunately, stating behavior at design time puts severe restrictions on the kinds of unanticipated structure and behavior that can be introduced in an object-oriented database system without modifying existing database schema

Correspondence to: B.J. Krämer

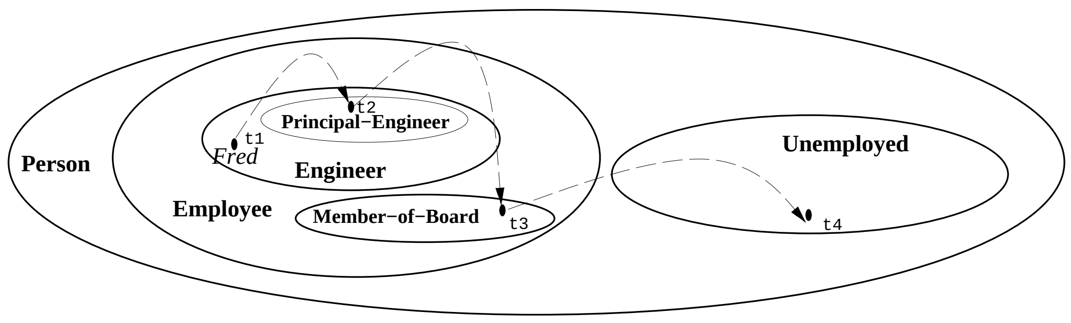

Fig. 1. Life cycle of the Fred object

classes and all their instances. There is currently no linguistic support to allow an object to alter its own behavior separately from the other members of the class to which it belongs. For example, every time that the Fred object needs to be re-classified (e.g., changes from an instance of class `Engineer` to `PrincipalEngineer`) it would have to first be removed from its original class and then be regenerated with the properties of its new class, thereby losing its original identity.

What is required is a linguistic framework which allows us to selectively seed new functionality to a distinguishable set of objects within a given class at runtime. In this way, it would be possible for members of a class to dynamically acquire different state variables and respond to different messages. This is not possible with conventional object-oriented database systems because it would involve changing the membership of an object from one class to another at runtime. This strictness of traditional object-oriented systems was first pointed out by proponents of prototype-based languages [LIE87] [StLU89].

1.1 The need for object dynamics

Because application and user needs are rarely stable, additional functionality needs to be constantly integrated into existing objects. To effectively model complex applications in which constantly changing situations can be represented, a system must be able to support the evolution and reconfiguration of individual objects. The strict uniformity of objects contained in a class is unreasonable: runtime structural and behavioral nuances should be specifiable for objects on an individual basis without restructuring the database schema or reorganizing the database contents.

An object that evolves by changing its type dynamically is able to represent changing situations as it can be an instance of different types from moment to moment. Such *dynamic objects* may fall into two broad categories.

1. Objects which need to *transform in a linear succession* from a beginning state to an end state. For example, consider the object Fred, who begins his professional life as an `Engineer` object and then becomes a `PrincipalEngineer` and finally a `MemberOfTheBoard` object (Fig. 1). Although the properties of Fred may vary in each of these phases they relate to the very same person (and, hence, need to relate to the same object) under different guises.
2. Another category of dynamic objects are those that *evolve in a pseudo-random fashion* depending on the occurrence of an external event. For instance, an academic may serve as a member of the university advisory committee, academic board and research advancement committee depending on years of service, performance and availability. These changes may be transient as they come and go with time (the lifetime of such committees is certainly short and their membership changes frequently).

When viewed externally, an object belonging to either of these two categories appears to oscillate among a set of different behaviors. Only some of these can be foreseen when the database schema is designed. It is thus highly desirable to adapt existing objects to new application requirements, while maintaining a single object identity. However, when designing an object system that enables objects to transit from one class to another, a number of issues have to be addressed. These include the following:

– How can the effects of dynamic changes to existing objects and classes be kept under control, so that they do not impact the structure of the database.
– Should the framework allow a member of some class to become a member of any other class or only of classes that it is related to, e.g., by subtyping.
– What restrictions need to be imposed on object transitions in order to balance expressiveness with the requirement of type safety?

In an object system that does not provide this kind of functionality, an inherent danger lies in the fact that programmers do not have the means to ensure that the object identifier of an evolving object is identical to the object identifier of the object from which it evolved. This problem is seriously compounded if other objects in the system contain references to an evolving object. The obvious solution is to create tables containing pointers to all potentially changeable objects and access them only through these table indices (usually called handles). Another solution is to create a new object every time an object changes class and then copy the appropriate properties of the old object to the new object and finally purge the old object. Yet another solution might be the Common Lisp approach, whereby every object identifier is represented by a pair of references: one pointing to the class and the other referring to the storage. However, such solutions are not only artificial but also introduce storage and performance overheads as well as adding a high degree of complexity and coupling. Moreover, they are error-prone and may result in corrupting already existing database objects.

From what has been stated above, it becomes obvious that we require linguistic mechanisms for object-oriented databases to support unanticipated behavioral oscillations for individual objects, or sets of objects, that have many types and share a single object identity. A language facility supports dynamic object properties best if new behavior can be introduced by stating to the system the differences between the existing behavior and the new desired behavior. Such language properties are known to support *object dynamics*.

1.2 Contributions

In this paper, we propose a model designed to extend the capabilities of object-oriented database systems so that they can support object dynamics. Central to our linguistic mechanisms is the concept of *role*. A role refers to the ability to change the classification of an object, so that the same object can simultaneously be an instance of different classes some of which are created dynamically. A role is an interface-based specification implemented on the basis of pre-existing objects in a way that allows a pre-existing object to gain (or shed) state and behavior dynamically while retaining its original identity. Roles designate significant, semantically meaningful shifts in object behavior (obtained dynamically) that are correlated with existing object properties and can be queried exactly as other conventional class objects. In summary, a role is determined on the basis of the collection of properties that are attached to the object in the center of interest and area responsible for bringing the role into existence.

This paper introduces the object role model (ORM), a model which integrates the concept of a role into object-oriented database technology in order to represent object dynamics. The linguistic facilities supported by the ORM introduce several special operators for creating roles and for allowing objects to be accessed in terms of particular roles that they may undertake. ORM is closely aligned with the ODMG-93 specifications for object databases. Thus, it offers the possibilities for a variety of object-oriented data models to provide the following features:

1. Support for objects with changing type: objects which dynamically change the current roles that they play – by gaining or retracting behavior.
2. Control of such forms of object evolution in accordance with application semantics by allowing objects to react to external events, in order to modify their behavior.
3. Respect of the structural and behavioral consistency of typed objects.

The research presented in this paper builds on preliminary work reported in [PAP91], [PaKB94] where we illustrated how roles may improve the versatility and modeling power of object-oriented database systems. In the remainder of this paper we develop our model in detail. The following section informally presents central concepts of the ORM and further motivates the approach through an illustrative example. The model is then formalized in Sect. 3 to provide a precise foundation for the specification of a handful of elementary operations to manipulate class DAGs. These are introduced in Sect. 4. High-level database operations, composed on the basis of these elementary operations, are then introduced in Sect. 5. Section 6 discusses related work, while Sect. 7 presents our summary and future work.

2 Basic concepts and definitions

The discussion that follows introduces basic concepts and terminology and focuses on objects which have the characteristics described below. We refer to these characteristics as the basic object model characteristics as they provide a sound basis for integrating the concept of a role into object-oriented databases. The object-oriented modeling concepts and terminology used in this paper are based on those found in [ATK89] and [ZdM89].

2.1 Basic object model characteristics

The basic object model constituents are *types*, *objects*, *classes*, and *relationships*.

Types: In a similar manner to abstract data types in programming languages, types define sets of structured data together with operations to modify such data in a controlled manner. A type consists of a unique type name, a collection of typed attributes and a set of operations (or methods). All types pertinent to a particular application are organized in a *directed acyclic type graph*, or *type DAG*. The nodes of the DAG are labeled with type names and are associated with a type specification, while the edges represent a partial ordering relationship among types that defines constraints on their type specifications (cf. Fig. 2).

Objects: All objects are instantiated from one type specification defining their structure and behavioral interface. Each object has a unique *object identifier* (oid) and a state. The oid serves as a unique handle to reference the object in order to access or modify its state. Object identity is implemented via system-generated logically unique identifiers for each object at the time of its creation [MZO89].

Classes: A class is based on a type specification and determines a set of objects. A class includes the runtime notions of *object creation* by cloning the prototype for the class, and the *extent*, which denotes the set of all objects that are instances of the class' type at a given point in time. Classes are organized into a class DAG, which is isomorphic to the corresponding type DAG. Whenever a new object is created as an instance of a type c, its object identifier is automatically added to the extent of the corresponding class c and to the extent of all superclasses of c (if any). Thus, an object can be a member of more than one classes at a time (multiple class membership). The top element class in the class DAG is called `Object` and all objects in the database are members of this class.

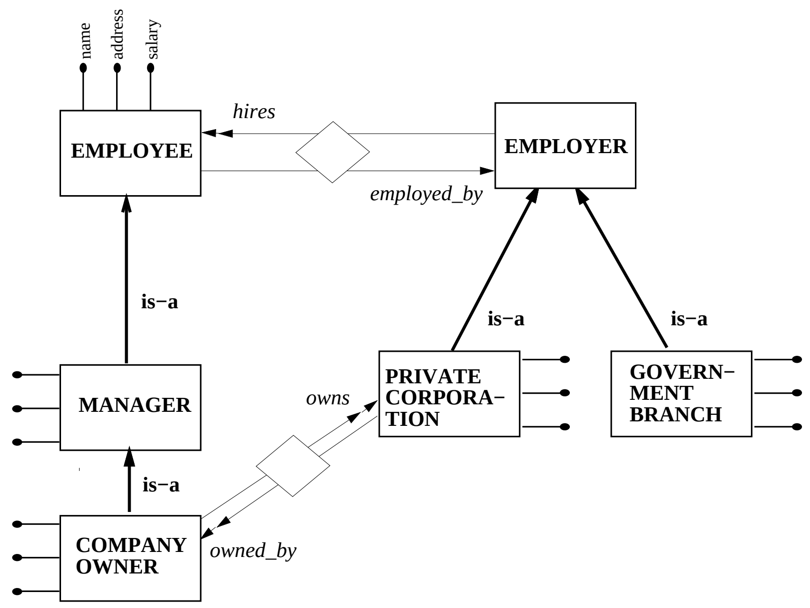

Fig. 2. A portion of the type DAG for an `employee-employer` object base

Relationships: An association in the object-oriented world can be modeled as a first-class object that has its own attributes and is existent dependent on its arguments. The types `hires` and `owns` in Fig. 2 are examples of relationships. Many object-oriented data models support an explicit relationship construct, such as, for instance, COCOON [SCH92]. Thus, the type DAG can be enriched by user-defined relationship types. The extent of a relationship class contains a set of pairs of object identifiers.

2.2 Example: type and class DAG

Figure 2 illustrates a schema portion of a sample employer-employee object base in the form of a type DAG. The graphic illustration is based on a variant of an ER diagram where diamonds represent binary relationship types and boxes represent conventional types. This figure shows that type `Employer` has as subtypes the two types `Private Corporation` and `GovernmentBranch`, whereas type `Employee` has as subtypes the types `Manager` and `CompanyOwner`. Type `Employee` is seen to be related to type `Employer` via a relationship type `employed_by`, whereas the inverse relationship type `hires` associates `Employer` with `Employee` types. *Relationships* between types can be constrained as usual to a 1-1 association (e.g., an `Employee employedBy` a single `Employer` is indicated by a single arrowhead from type `Employee` to type `Employer`); 1-N (e.g., an `Employer hires` a set of `Employee`s is indicated by a double arrowhead); or M-N (e.g., a set of `CompanyOwner`s own multiple `PrivateCorporations`).

Figure 3 depicts the class hierarchy derived from the type DAG in Fig. 2. Ovals in Fig. 3 denote class extents, while dashed rectangles denote relationship extents. Ovals and dashed rectangles are shown to contain the oids of the instances associated with the types introduced in Fig. 2. To fully understand the context of Fig. 3, consider an object of type `PrivateCorporation` with the oid `pc1`. When this object is created as an instance of that class, its oid is not only included in the extent of its corresponding class `PrivateCorporation` but also in the extent of its superclass `Employer`. Formal definitions of the class and type DAG are also given in Sect. 3.

2.3 Extending the basic model with roles

A role may be thought of as a typed *abstract channel* providing an alternative perspective on an existing object. A role ascribes properties that possibly vary over time and is implemented as an extension of existing objects. The purpose of a role is to model different "active" (application-specific) representation alternatives for the same object in terms of both structure and behavior. A particular object may concurrently exhibit many roles which are obtained dynamically throughout its lifespan. This type of object dynamism can be achieved by subdividing and grouping together distinguishable (and related) objects contained in the DAG classes and by defining subclasses or super-classes dynamically to encompass these object groupings. Each of the new classes created in this manner is a *role-defining class*. The purpose of role-defining classes is to partition an object into different forms which are specific to the application in which the object occurs.

The example depicted in Figs. 2 and 3 has been chosen only for reasons of simplicity. It is not characteristic of the usage of the role model. Typical complex systems where roles can be of benefit may be, for example, design, product development and knowledge applications. With such applications there is a need for designers and knowledge workers to experiment with their environments (by using role objects) and they also require effective database support to store useful stable and tested role objects as part of an object base shared between many applications.

Figure 4 extends the context of Fig. 3 with dynamic objects to satisfy the needs of a particular application. The bottom half of Fig. 4 shows how the class DAG can be privately extended to support role objects. A different application may

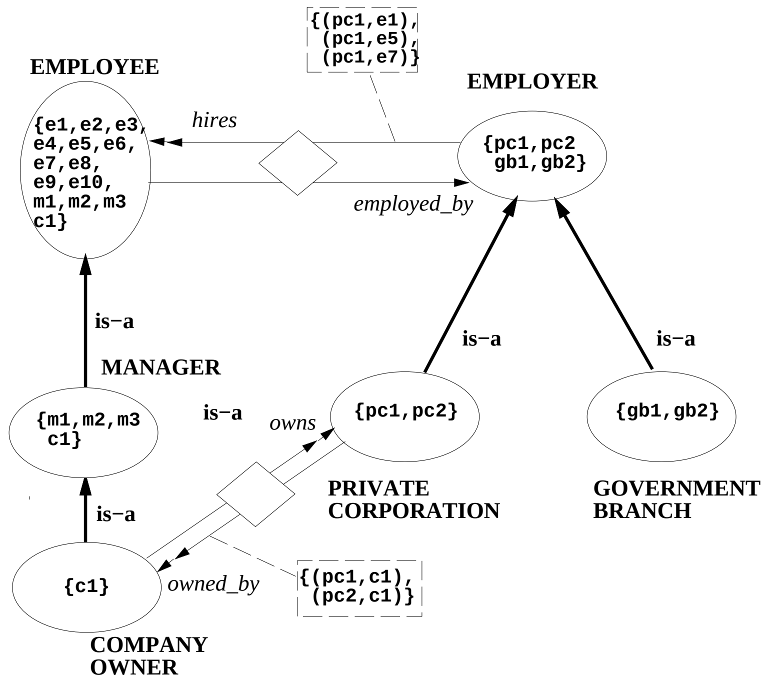

Fig. 3. The class hierarchy for the type schema in Fig. 2

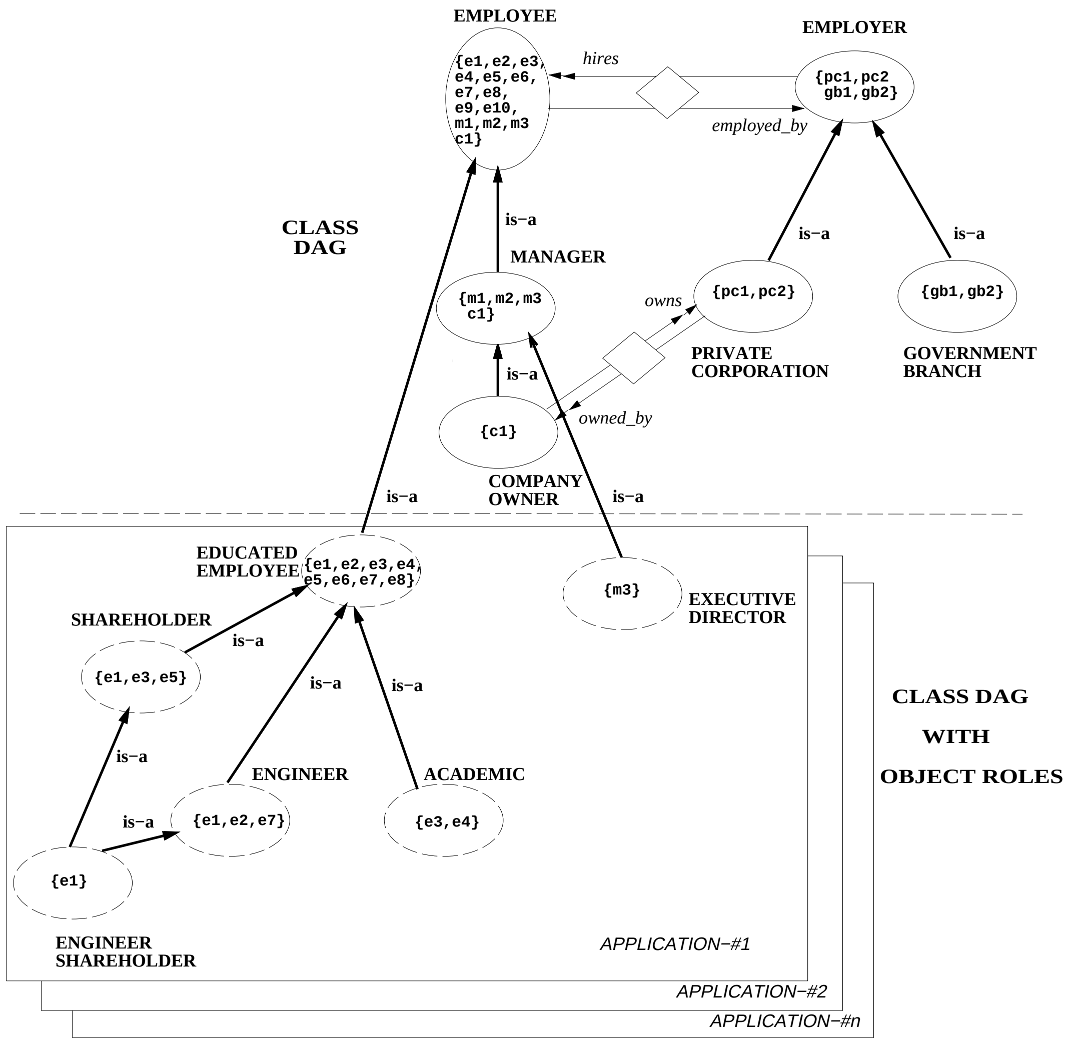

Fig. 4. The class hierarchy for the type schema in Fig. 2 evolved with roles

use the same class DAG and extend it with different kinds of roles meaningful to its context. Therefore, it is useful to perceive a role as a customizable slant into an object which constitutes a form of abstraction that configures this object in a way that is dictated by the needs of the application within which the role occurs.

By default, a role is *visible* only within the scope of the specific application that created it. Only in special circumstances can a role be shared between applications or become permanent, i.e., become part of the type DAG. This will be explained in detail in Sect. 5.3.4.

To comprehend how roles relate to object dynamics and how they may impact upon the behavior of objects, consider the case of an Employee object with oid $\mathtt{e}_2$ which dynamically obtains the role of an EducatedEmployee and an Engineer through a role defining operation (as explained in Sect. 5.1). The dashed ovals in Fig. 4 indicate the existence of roles which an object may assume or relinquish during its lifespan in addition to the properties acquired upon creation. Such roles include, for instance, the roles EducatedEmployee and Engineer for the Employee object with oid $\mathtt{e}_2$. Thus, to describe this situation, the DAG is extended by including two new (role-defining) classes, namely EducatedEmployee and Engineer that do not correspond directly to the schema types in Fig. 2. As roles are used to facilitate migration of objects in the class DAG, both roles EducatedEmployee and Engineer contain in their extent the Employee object with oid $\mathtt{e}_2$. Accordingly, the object with oid $\mathtt{e}_2$ co-exists now in the extent of the classes Employee, EducatedEmployee and Engineer. Roles allow the introduction of new classes into the class DAG without having to modify the definitions of existing classes. This implies that their effects are localized to the context of the application that created and uses them.

Roles are transient in nature. They have a finite lifespan which is defined by the application program that has created them. Roles are created by individual application programs, are stored in an object repository local to the programs that created them, and they have no global scope and effects, i.e., they cannot be “seen” outside the context of the application that created them. They normally do not become persistent unless there is an explicit requirement for this to happen (cf. Sect. 5.3). Each role introduces additional attributes and methods to existing objects – through a set of role-specific operations – thereby permitting the representation of behavioral shifts and increments. As roles re-define behavior defined in their classes of origin, the system may give different answers depending on how a particular object is viewed. For example, assume that we defined a method $income$ for Employee objects which gives their annual salary. This method might be re-defined when we consider the role of Employee object as a Shareholder to give us a combined Employee and Shareholder income.

The set of roles played by an object is obviously determined by its position in the class DAG. The existence of all the roles of interest for a given object, its *role set*, fully characterizes this object. The term role set is used here to aggregate information about how an object is evolving, and is determined by the set of classes in whose extent the object identifier occurs. These classes form a connected subgraph of a given class DAG extended with roles. For instance, the role set $\rho(\mathtt{e}_1) = \{$Employee, EducatedEmployee, Shareholder, Engineer, EngineerShareholder$\}$ includes all the roles that objects of the type Employee can perform. We use the term *ancestor role(s)* to denote all the roles above a given role in the class DAG. The term *parent role* is reserved for the role(s) immediately above a given role, whereas the term *descendant role(s)* is used to denote all the roles below that role in the class DAG. For example, the ancestral roles for Engineer are EducatedEmployee and Employee, its parent role is EducatedEmployee and its set of descendant roles consists of EngineerShareholder. Users can thus access and query objects from a particular perspective.

The main objective of roles in the ORM is to customize objects – according to application needs – so that they become equipped with their own idiosyncratic behavior. In this respect, roles present some similarity with views, however, unlike views their objective is to cater for dynamic object migration and automatic re-classification – without affecting the database schema[1]. This implies that the semantics of the ORM operations are *object-preserving* in the sense that they return part of the extents of their input classes. More importantly, the extension of the class DAG – due to the introduction of role-classes – does not change the set of objects contained in the class DAGs. These and other virtues of object-preserving operations and transformations have been addressed by [BER91] and [SCH92]. The emphasis is on preserving the consistency of (existing) evolving objects rather than creating new objects. New objects are created only through pre-existing DAG classes and are re-classified into roles either eagerly or lazily, depending on the case. If the role operations resulted in the generation of new objects, there will be a necessity for the system to maintain and cross-correlate multiple snapshot object configurations from diverse application programs.

3 Formalization of the ORM

In this section we formalize the ORM. First we introduce elementary concepts such as data types, objects, values, and method signatures. These form basic constituents of our definition of type and class DAGs. Our notion of well-formed and type-safe class DAGs is then derived in several steps from the definition of type DAGs. It relies on typing concepts which are also introduced in this section. In our definitions we adopt and extend concepts of the O_2 data model as defined in [KaLR92].

3.1 Data types, values and methods

In the definition of the syntax and semantics of schema types, classes, objects and roles below, we assume the following pairwise disjoint sets serving as *basic syntactic* and *semantic domains* in subsequent definitions:

A: a set of *attribute symbols* for naming object attributes; we use variables $a, a', a_1, a_2, \ldots$ as typical elements of **A**.

[1] The differences between roles and views are covered in some detail in Sect. 6 which describes related research work.

ST: a set of *schema-type names*.
RT: a set of *role-type names*.
R: a set of *relationship names* with variables $r, r_1, r_2, \ldots$ denoting arbitrary elements of **R**.
ID: denotes the set of all oids and variables, $i, i', i_1, i_2, \ldots$ are assumed to range over **ID**.

From **ST** and **RT**, we form the set **C** := **ST** ∪ **RT** of *class-type names*, and we use variables $c, c', c_1, c_2, \ldots$ to range over **C**. From **C** and **R**, we form the set **T** := **C** ∪ **R**; it allows us to refer collectively to class and relationship names whenever the distinction is irrelevant or can be deduced from the context. *Object identifiers* are modeled as pairs (c, n) with n a natural number in **N**.

The structure of objects is defined by means of two type constructors which allow us to form record and pair types[2] to describe the structure of class and relationship instances.

Definition 1. For a subset C of **C**, the set of *data types over* C, written $types(C)$, is defined as follows:

1. class names in C are in $types(C)$;
2. every $(a_1 : c_1, \ldots, a_n : c_n)$ is in $types(C)$ and is called a *record type*, provided that the c_i occur in C and the attribute names are distinct, i.e., $a_i \neq a_j$ for $1 \leq i < j \leq n, 0 \leq n$;
3. if c_1, c_2 are elements of C, every (binary) *relationship type* of the form (c_1, c_2) is in $types(C)$.

For reasons of simplicity, this definition does not admit recursive record or relationship types. *Record types* as ATtribute names must be unique. Each attribute name a_i for $i = 1, \ldots, n$ of a record type $(a_1 : c_1, \ldots, a_n : c_n)$ is viewed as an *attribute function* $a_i : c_1 \times c_2 \times \cdots \times c_n \rightarrow c_i$ mapping each instance of the record type into its i-th component. To access the source and destination class c_1 and c_2 of any relationship type (c_1, c_2), we also use the generic *projection functions* **src** and **dest**, respectively. Henceforth we use the variables $t, t', t_1, t_2 \ldots$ to denote data types.

Values are instances of data types. They are used to define the state of objects.

Definition 2. For a subset I of **ID**, we inductively define the set of *values over* I, written $val(I)$, as follows:

1. every element in I is in $val(I)$;
2. the special symbol $\perp$ is a distinct element in $val(I)$; it denotes the *undefined* value;
3. if $v_1, \ldots, v_n$ are in $val(I)$, so is the labeled record $(a_1 = v_1, \ldots, a_n = v_n)$ for $n \geq 0$, provided that all attributes a_i are distinct;
4. if $v_1, v_2 \neq \perp$ are in $val(I)$, then all pairs (v_1, v_2) are in $val(I)$.

In the sequel, the variables $v, v', v_1, \ldots$ are used to denote values.

Operations, often called methods in object-oriented settings, capture the behavior common to all instances of certain types. *Method signatures* are of the form

$$c.m : t_1 \rightarrow t_2 \,.$$

They provide a method name m, a class name $c \in C$ the method is associated with, an argument and a result type $t_1, t_2 \in types(C)$, respectively.

3.2 Schema-type DAGs

A schema-type DAG (or simply type DAG) defines the static part of a database. It organizes a database in the form of a directed acyclic graph. The nodes of the graph are labeled with schema-type and relationship names and are associated with data types and method signatures which determine the structure and behavior of instances of these types, respectively.

Definition 3. A *type DAG* is a quadruple $(T, \prec, \tau, \mu)$, where T is a subset of class and relationship types in **T** from which we derive the following two disjoint subsets:

$C := T \cap$ **ST**, a finite set of schema-type names and
$R := T \cap$ **R**, a finite set of binary relationship names;
$(T, \prec)$: is a partially ordered set[3]; $\prec$ denotes an *is_a relationship* and is disjoint on C and R;
τ : is a mapping from T to $types(C)$ such that, for all c in C, $\tau(c)$ is a record type and, for all r in R, $\tau(r)$ is a relationship type, respectively; τ is called a *type assignment*;
μ : is a mapping from C into a set of method signatures of the form $c.m : t_1 \rightarrow t_2$ such that $c \in C$ and $t_1, t_2 \in types(C)$.

For any two classes c, c' with $c \prec c'$ in C, c is called a *subclass* of c' and c' is called a *super-class* of c.

By $c^\bullet$ we refer to the set $\{c' | c \prec c'\}$ of super-classes and ${}^\bullet c$ denotes the set $\{c' | c' \prec c\}$ of subclasses of the class c.

According to this definition, the type DAG presented in Fig. 2 reads as follows:

$C = \{$`EMPLOYEE`, `MANAGER`, `COMPANY_OWNER`, `EMPLOYER`, `PRIVATE_CORPORATION`, `GOVERNMENT_BRANCH`$\}$
$R = \{$`hires`, `employed_by`, `owns`, `owned_by`$\}$
src(`hires`) = **dest**(`employed_by`) = `EMPLOYER`,
dest(`hires`) = **src**(`employed_by`) = `EMPLOYEE`,
src(`owns`) = **dest**(`owned_by`) = `COMPANY_OWNER`,
dest(`owns`) = **src**(`owned_by`) = `PRIVATE_CORPORATION`,
$\prec = \{$(`MANAGER`, `EMPLOYEE`), (`COMPANY_OWNER`, `MANAGER`), (`COMPANY_OWNER`, `EMPLOYEE`), (`PRIVATE_CORPORATION`, `EMPLOYER`), (`GOVERNMENT_BRANCH`, `EMPLOYER`)$\}$
τ(`EMPLOYEE`) = (`name` : ..., `address` : ..., `salary` : ...)
...
τ(`hires`) = (τ(`EMPLOYER`), τ(`EMPLOYEE`))
...
μ(`EMPLOYEE`) = ...

[2] To simplify the formal definition, we omit standard atomic data types such as **Boolean**, **integer**, **string**, or **real**.

[3] That is, $\prec$ is a reflexive, antisymmetric and transitive relationship over T.

The is_a relationship of a type DAG may be used to induce a sub typing relationship on class names. Moreover, a record t can be used whenever another record t' is expected, but only if t has at least the same attributes as t' and the types of the attributes of t are subtypes of the types of the corresponding attributes of t'. Similarly, a relationship type r is a subtype of another relationship type r' if the source and destination types of r are subtypes of the source and destination types of r'.

Definition 4. The *subtyping relationship* over a type DAG $(T, \prec, \tau, \mu)$, denoted by $\leq$, is defined by the following minimal set of rules:

1. $c_1 \prec c_2$ implies $c_1 \leq c_2$;
2. $(a_1 : t_1, \ldots, a_m : t_m, a_{m+1} : t_{m+1}, \ldots, a_{m+k} : t_{m+k}) \leq (a_1 : t'_1, \ldots, a_m : t'_m,)$
   if $t_i \leq t'_i$ for $i = 1, \ldots, m$;
3. $(t_1, t_2) \leq (t'_1, t'_2)$ if $t_1 \leq t'_1$ and $t_2 \leq t'_2$.

The previous definition includes no integrity constraints that would prevent the specification of ill-formed type DAGs. We consider a type DAG **TH** to be ill-formed if its is_a relationships on classes and relationships do not conform with the subtyping relationship according to the type assignment τ of **TH**, or if method signatures do not conform with the type structure. Such deficiencies are excluded by the following definition of well-formed type DAGs.

Definition 5. A type DAG $(T, \prec, \tau, \mu)$ is *well-formed* if for all c_1, c_2 in $C := T \cap$ **ST** and, for all r_1 and r_2 in $R := T \cap$ **R**, the following conditions hold:

1. $r_1 \prec r_2$ implies **src**$(\tau(r_1)) \prec$ **src**$(\tau(r_2))$ and **dest**$(\tau(r_1)) \prec$ **dest**$(\tau(r_2))$;
2. $c_1 \prec c_2$ implies $\tau(c_1) \leq \tau(c_2)$;
3. for all c in C, we have that $\mu(c) = \{c.m : t_1 \rightarrow t_2\}$;
4. if $c \prec c'$ and method m is defined in c with signature $c.m : t_1 \rightarrow t_2$ and in c' with signature $c'.m : t'_1 \rightarrow t'_2$, then $t_1 \leq t'_1$ and $t'_2 \leq t_2$ must hold.

Further, we assume that multiple-inheritance conflicts may not occur. This can be excluded explicitly using a sufficient condition as given in [KaLR92]. Condition 3 of the above definition ensures that μ is prefix closed with respect to class names. Conditions 2 and 4 ensure *type safety* on attributes and methods by requiring *covariance* (restriction) for class names and result types and *contravariance* (expansion) for arguments of methods.

Informally, the use of argument contravariance and result covariance can be explained as follows. Assume we expect a function or method f to have type $t_1 \rightarrow t_2$ and therefore consider t_1 arguments as permissible when calling f. Now assume f actually has type $t'_1 \rightarrow t'_2$ with $t_1 \leq t'_1$. Then we can pass all the expected permissible arguments of type t_1 without type violation; f will return results of type t'_2 which is permissible if $t'_2 \leq t_2$ because the results will then also be of type t_2 and are therefore acceptable as they do not introduce any type violations.

In the remainder of this paper when speaking about a type DAG we always mean a well-formed type DAG .

3.3 Objects, classes and relationships

Classes are inhabited by objects that are simply viewed as pairs associating an object identifier with a value according to Definition 2.

Definition 6. An *object* is a pair (i, v).

In Definitions 1 and 2, we defined independently how data types and values are formed correctly. But as objects are instantiated from data types, we must ensure that the value associated with a particular oid is compatible with the type of that oid. This is made precise in the definition below which indicates to what set of values the data types associated with the type names in a type DAG can be instantiated. We call this the interpretation of a type.

Definition 7 For any type DAG $(T, \prec, \tau, \mu)$ with $c, c_1, \ldots, c_m \in C := T \cap$ **ST** the *interpretation* of a type t in $types(C)$ under τ, denoted by $[\![t]\!]_\tau$, is defined as follows:

1. $[\![c]\!]_\tau = \{i | i = (c', n), c' \in C, c' \prec c, n \in \mathbf{N}\} \cup \{\perp\}$, i.e., oids of objects that were instantiated as members of subclasses of class c are allowed as values of type c;
2. $[\![(a_1 : c_1, \ldots, a_m : c_m)]\!]_\tau = \{(a_1 = v_1, \ldots, a_m = v_m, a_{m+1} = v_{m+1}, \ldots, a_{m+k} = v_{m+k}) | v_i \in [\![c_i]\!]_\tau, i = 1, \ldots, m + k, k \geq 0\}$, i.e., only those records that have at least the set of attributes of the record type and whose attributes assume values of subtypes of the corresponding attribute types are accepted as values of the record type;
3. $[\![(c_1, c_2)]\!]_\tau = \{(v_1, v_2) | v_1 \in [\![c_1]\!]_\tau, v_2 \in [\![c_2]\!]_\tau\}$, i.e., only pairs of values whose first and second component assume values of subtypes of the first and second component of the relationship type are taken as values of the relationship type.

In the above definition, each class name c is interpreted by the set of oids naming objects of any subclass of c (including c because $\prec$ is reflexive), each record type by the set of records that have at least as many components as the record type and assign a value out of the interpretation of the attribute c_i to the corresponding attribute a_i and, finally, each relationship type is interpreted by the set of pairs of oids of the proper source and destination class type.

We allow the undefined value $\perp$ to be used as an attribute value in records to cope with situations where no well-defined value is known.

3.4 Class hierarchies and roles

As mentioned in the previous section, objects can play several roles at a time and they can change their roles during their lifetime. This dynamics is captured in the notion of a class DAG[4] which is derived from a type DAG by adding a set ω of instantiated objects and specifying an oid assignment π_o that maps each class type (and relationship) into the set of (pairs of) oids of objects that were created as instances of that class. Moreover, it maps each role class into a set of oids of role objects acquired from other classes by explicit object migration.

[4] Henceforth we will refer to a class DAG as the class hierarchy to avoid any confusion with the notion of a type DAG.

Note that this overloaded interpretation of π_o is only possible because *we do not allow role classes to have their own instances*.

Definition 8. Given a well-formed type DAG **TH** = $(T, \prec, \tau, \mu)$, then a class hierarchy derived from **TH**, *class hierarchy* for short, is a structure **CH** = $(T', \prec', \tau', \mu', \omega, \pi_o)$, where

$T \subseteq T'$: such that $(T' \backslash T) \cap$ **ST** $= \emptyset$, i.e., the additional elements in T' are role classes;

$(T', \prec')$: is a partially ordered set such that the restriction of $\prec'$ to elements in T is equivalent to $\prec$, i.e., $\prec' |_T = \prec$;

τ' : is a type assignment such that the restriction of τ' to the domain of τ is identical to τ, i.e., $\tau'|_T = \tau$;

μ' : is a method assignment identical to μ on the restriction of μ' to the domain of μ, i.e., $\mu'|_C = \mu$;

ω : is a set of objects;

π_o: is a function, called *oid assignment*, which maps each class name $c \in T'$ and each relationship name $r \in T'$ into a finite set of identifiers and a finite set of pairs of identifiers naming objects in ω, respectively; π_o is disjoint on schema types and relationships, i.e., for all classes $\alpha, \alpha' \in T'$ with $\alpha \neq \alpha'$, we require that $\pi_o(\alpha) \cap \pi_o(\alpha') = \emptyset$; $\pi_o(\alpha)$ denotes the set of *own instances* of α if $\alpha \in T$ and it denotes the set of *transient instances* if α is role class name in $T' \backslash T$; the *extent* of each class or relationship α in T' is given by the extension π of π_o defined by

$$\pi(\alpha) = \bigcup_{\alpha' \in^\bullet \alpha} \pi_o(\alpha') \, .$$

By $oids$(**CH**) we denote the set $\{i | (i, v) \in \omega\}$.

Like type hierarchies, class hierarchies may be ill-formed if, for instance, the value associated with some object identifier i in the extent of $\pi(c)$ of some class c is incompatible with the type $\tau(c)$ of c, or if the additional role classes and their associated methods do not satisfy the conditions in Definitions 5.2, 5.3, and 5.4. The following definition gives a number of integrity constraints that a well-formed class hierarchy must satisfy. The first condition in the following definition states that the values of objects must be *compatible with their type* under the given type assignment.

Definition 9. A class hierarchy $(T, \prec, \tau, \mu, \omega, \pi_o)$ is *well-formed* if conditions 2, 3, and 4 of Definition 5 and, additionally, the following conditions hold:

1. for all objects (i, v) in ω and all classes $c \in T$ with $i \in \pi(c)$, we require that $v \in [\![\tau(c)]\!]_\tau$;
2. $$\bigcup_{c \in T} \pi_o(c) = \{i | (i, v) \in \omega\} \, .$$

3.5 *Type safety*

One of the core features of the approach to roles is preserving an object's identity while allowing it to change behavior and structure. An object may participate in many roles, but it has a unique oid. The only reference to an object is through its object identifier in conjunction with the specification of a role-class name. Allowing objects to dynamically assume new (and relinquish old) behavior by introducing new classes in (and dropping old classes from) the class hierarchy presents a serious threat to the type safety of the system. To avoid these potential problems, the ORM bases its type-checking mechanisms on the notion of *type conformance*. Conformance is a relation between types which determines whether objects of one type can be used in lieu of objects of another as discussed in the context of Definition 5.

A number of object-based or object-oriented languages such as Emerald [BLA87], Trellis-Owl [SCH85], Eiffel [MEY88] and object models such as TEDM [MZO89] and FROOM [MaB90] have adopted the notion of conformance to determine whether an object is of the specified type by comparing its interfaces with the interface specified by the type in question.

Observation 1. *The signatures, i.e. type definitions and method signatures, of any two subclass-related classes in a well-formed class hierarchy conform.*

This observation holds due to the conditions of Definition 9. An elaborate discussion about method conformance in the presence of covariance for class names and result types and contravariance of argument types can be found in [ScZ94] and [ScC95].

3.6 *Type union and intersection*

To support dynamic specialization and generalization, we need to introduce two partial operators that define the union and intersection of record types.

Definition 10. Let t and t' be two record types formed over a given class hierarchy. Moreover, let

$$\begin{array}{ll} t & = (a_1 : t_1, \ldots, a_k : t_k, a_{k+1} : t_{k+1}, \ldots, a_{k+m} : t_{k+m}) \\ t' & = (a_1 : t'_1, \ldots, a_k : t'_k, a'_{k+1} : t'_{k+1}, \ldots, a'_{k+n} : t'_{k+n}) \end{array}$$

with $t_i \leq t'_i$ for $i = 1 \ldots k$

Then the *type union* $t \sqcup t'$ is the record type

$$(a_1 : t_1, \ldots, a_k : t_k, a_{k+1} : t_{k+1}, \ldots, a_{k+m} : t_{k+m}, a'_{k+1} : t'_{k+1}, \ldots, a'_{k+n} : t'_{k+n}) \, ,$$

with $a_i \neq a'_j$ for $i = k+1, \ldots, k+m$ and $j = k+1, \ldots, k+n$.

Dually, the *type intersection* $t \sqcap t'$ is the record type

$$(a_1 : t'_1, \ldots, a_k : t'_k) \, .$$

In all other cases, we set $t \sqcup t' = \top$ and $t \sqcap t' = \bot$. Note that $\sqcup$ and $\sqcap$ are only partially defined. If defined, they are associative, as the order of attributes in a record type is semantically irrelevant. Therefore, we omit parenthesis when forming the union or intersection of multiple types.

Observation 2. *If $t \sqcup t'$ and $t \sqcap t'$ are defined, then $t \leq t \sqcup t'$ and $t \sqcap t' \leq t$ hold.*

4 Elementary operations on class hierarchies

The ORM provides elementary operations to modify class hierarchies. These include operations to:

1. modify the class hierarchy by adding and deleting role classes,
2. migrate objects from existing classes to new role classes,
3. modify the type definition of a role class by adding attributes and possibly new methods.

Although these operations may appear to be similar to schema update operations, they introduce a new dimension when combined with object migration. We assume that all elementary operations introduced below are strict, i.e., they are undefined whenever a constituent operator such as the type union or intersection are undefined. Moreover, we assume that the operations are undefined whenever at least one of their preconditions fails to hold. In this case, they have no effect on the given well-formed class hierarchy.

In this paper, we consider roles that can be defined as sub- and super-classes of existing classes and objects that can migrate into sub- or super-classes. This allows us to provide a comprehensive set of operations that guarantee type safety and consistent class DAGs. The implications of relaxing these constraints to apply to classes other than sub- or super-classes are the subject of ongoing research. Some preliminary results have been reported in [PKS95].

Throughout this section, we use the symbol **CH** to denote the following well-formed class hierarchy $(T, \prec, \tau, \mu, \omega, \pi_o)$.

4.1 Modifying the structure of a class DAG

The following operation supports the *generalization* of existing classes $c_1, \ldots, c_k$ into a new class c acting as the direct super-class of the former such that the type associated with c is the intersection of the type associated with c's subclasses. Generalization is, for instance, useful for adding common behavior to unrelated classes in a class DAG, as needs arise, by forming a common super-class and associating the new behavior with that super-class from which it is automatically inherited.

Definition 11. The operation

addRoleClass c **asSuperclassOf** $c_1, \ldots, c_k$

maps **CH** into a new class hierarchy $(T \cup \{c\}, \prec', \tau', \mu', \omega, \pi'_o)$, where

1. c in **RT**,
2. $\prec' = \prec \cup \{(c_1, c), \ldots, (c_k, c)\}$,
3. $\tau'(c) = \tau(c_1) \sqcap \cdots \sqcap \tau(c_k)$ and $\tau'(\alpha) = \tau(\alpha)$ for all $\alpha \in T$ (i.e., τ' is the same as τ for all old classes and relationships in T),
4. $\mu'(c) = \emptyset$ and $\mu'(c') = \mu(c')$ for all c' in T
5. $\pi'_o(c) = \emptyset$, $\pi'(c) = \pi(c_1) \cup \cdots \cup \pi(c_k)$, and, for all α in T, we have that $\pi'_o(\alpha) = \pi_o(\alpha)$ and $\pi'(\alpha) = \pi(\alpha)$

if the following conditions hold:

1. c does not occur in T,
2. $c_1, \ldots, c_k$ are elements of T, and
3. $(T \cup \{c\}, \prec')$ is a partial order.

Similarly we define an operation that allows us to specialize several classes $c_1, \ldots, c_k$ dynamically into a more specialized role class c. Its type is the union of the types of the former.

Definition 12. The operation

addRoleClass c **asSubclassOf** $c_1, \ldots, c_k$

maps **CH** into a new class hierarchy $(T \cup \{c\}, \prec', \tau', \mu', \omega, \pi'_o)$, where

1. c in **RT**,
2. $$\prec' = \prec \cup \{(c, c_1), \ldots, (c, c_k)\} \cup \bigcup_{c'_1 \in^\bullet c_1} \{(c, c'_1)\} \cup \ldots \cup \bigcup_{c'_k \in^\bullet c_k} \{(c, c'_k)\},$$
3. $\tau'(c) = \tau(c_1) \sqcup \cdots \sqcup \tau(c_k)$ and $\tau'(\alpha) = \tau(\alpha)$ for all $\alpha \in T$,
4. $\mu'(c) = \emptyset$ and $\mu'(c') = \mu(c')$ for all c' in T
5. $\pi'_o(c) = \emptyset$, $\pi'(c) = \emptyset$, and, for all α in T, we have that $\pi'_o(\alpha) = \pi_o(\alpha)$ and $\pi'(\alpha) = \pi(\alpha)$,

provided that the same conditions as in Definition 11 hold.

Observation 3. *Both operations preserve the well-formed ness of* **CH** *and are thus type-safe.*

Definition 13. The operation

markDeleteRoleClass c

maps **CH** into a *shadow class DAG* $(T', \prec', \tau', \mu', \omega, \pi'_o)$, provided that c occurs in $T' \cap$ **RT**, where

1. class c, all subclasses of c, all relationships that depend on classes in $^\bullet c$ disappear, i.e., $T' = T \backslash (^\bullet c \cup \{r \in T |$ **src**$(\tau(r)) \vee$ **dest**$(\tau(r)) \in {}^\bullet c\})$;
2. the is_a-relationship *relationship* is reduced by all pairs whose first or second component is no longer a member of T', i.e., $\prec' = \{(\alpha_1, \alpha_2) \in \prec | \alpha_1, \alpha_2 \in T'\}$;
3. the type and method assignments of all remaining classes and relationships do not change, i.e., $\tau'(\alpha) = \tau(\alpha)$ and $\mu'(\alpha) = \mu(\alpha)$ for all $\alpha \in T'$;
4. $\pi'(\alpha) = (\pi(\alpha) \backslash \pi(c)) \cup \pi_o(\alpha)$ *and* $\pi'_o(\alpha) = \pi_o(\alpha) \forall\, \alpha \in T'$. We have that $\pi'(\alpha) = (\ldots) \cup \pi_o(\alpha)$.

The continuous addition of role classes to a given class DAG is likely to reduce data space and affect the performance of ORM implementations. From a system designer's point of view, it is tempting to provide an operation for purging role classes that become obsolete. However, as there may be references to role objects, methods and classes that we wish to delete, instead of providing a conventional delete operation, we rather rely on the existence of an efficient garbage collector and provide an operation to mark these roles that are designated as invalidated by the users. We then let the garbage collector perform the deletion when no further references to that role class or its instances exist. If no garbage collector is available, special provisions can be made at the systems level, e.g., by building up a cross-reference table, to keep track of the establishment and destruction of references and thus prohibit dangling references upon deletion of objects, methods or attributes. The above *mark-delete* operation extends this type of deletion marking to all subclasses of the selected class and to all relationships that have any of the marked classes as source or destination. As role objects are not destroyed, there are no dangling references. Also new references to an invalidated role, as well as the dispatching of messages to invalidated role objects, result in trappable errors.

The effect of this operation on a given class DAG is illustrated in Fig. 5. This figure shows that, when marking a class as deleted, all of its descendant classes as well as relationships, which have this class either as origin or destination, are also marked as deleted. All remaining classes are shaded to denote the existence of a shadow DAG as defined above.

4.2 Migrating objects

Two types of object migration are potentially useful in a class DAG: migration from a class c to a subclass or super-class of c or to an arbitrary class. The former supports the dynamic specialization or generalization of objects, while the latter models the case where an object discontinuously changes its structure and behavior. In the framework of this paper we consider object migration into subclasses and super-classes only.

Definition 14. The operation

migrateObject i **fromClass** c_1 **toSubclass** c_2

maps **CH** into $(T, \prec, \tau, \mu, \omega, \pi'_o)$, provided that

1. $c_1 \in T$ and $c_2 \in T \cap$ **RT**,
2. $c_2 \prec c_1$, and
3. $i \in \pi_o(c_1)$, i.e., identifiers in $\pi(c_1)$ that are not owned or transient to c_1 but stem from other subclasses of c_1 cannot be migrated.

Then

1. a) i becomes a new member of the set of transient objects of c_2, i.e., $\pi'_o(c_2) = \pi_o(c_2) \cup \{i\}$,
   b) while the sets of own and transient objects of all other class and relationship types $\alpha \neq c_2$ remain the same, i.e., $\pi'_o(\alpha) = \pi_o(\alpha)$,
   c) for all super-classes $c \in c_2^\bullet$ of c_2, their extents are augmented by i, i.e., $\pi'(c) = \pi(c) \cup \{i\}$,
   d) while the extents of all other classes α in $T \backslash c_2^\bullet$ remain unchanged, i.e., $\pi'(\alpha) = \pi(\alpha)$, and
2. the value of each attribute a in $\tau(c_2)$ that does not occur in $\tau(c_1)$ is treated as $\perp$ whenever object (i, v) is accessed as an object of class c_2.

Class c_1 is called the *parent role* of c with respect to i.

Observation 4. *Each object in a class DAG* **CH** *has a smallest class, i.e., for all* $(i, v) \in \omega$*, there is a* $c \in T$ *such that* $i \in \pi_o(c)$ *and, for all other classes* $c' \in {}^\bullet c \cup (T \backslash c^\bullet)$*, we have that* $i \notin \pi(c')$.

By definition of a well-formed class hierarchy each object in ω is created as an instance of exactly one schema type and occurs in the extent of all super-classes of its corresponding class. The class where objects in ω are created is called their *smallest class*. The extent of each role class c in the ORM is empty upon creation (cf. Def. 11). The only way to populate the extent $\pi_0(c)$ of class c with oids is through object migration. One way of achieving object migration in the ORM is by placing objects of a given class c_1 into its descendants extents. This class may include in its set $\pi_o(c)$ objects of its own (for which it is the smaller class) or transient oids (for some of which it is the smallest class). The effects of migrating an object into a subclass c_2 of the given class c_1 are that: (a) the oid of the migrated object becomes a member of the set of transient objects of c_2; (b) c_2 becomes the new smallest class of the migrated object; and (c) the extents of any super-classes of c_2 are extended by the oid of the migrated object.

4.3 Adding attributes and methods

Definition 15. The operation

addAttribute $a : t = v$ **to RoleClass** c

maps **CH** into $(T, \prec, \tau', \mu, \omega, \pi_o)$ where

1. $\tau'(c) = (a_1 : c_1, \ldots, a_k : c_k, a : t)$ if $\tau(c) = (a_1 : c_1, \ldots, a_k : c_k)$ and $\tau'(\alpha) = \tau(\alpha)$ for all $\alpha \neq c \in T$ and
2. for any oid $i \in \pi(c)$ occurring in the extent of the modified class, we have that $a(i) = v$,

provided that c is a role class, i.e., $c \in T \cap$ **RT**, c has no subclasses, i.e., ${}^\bullet c = \emptyset$, $a \neq a_j$ for $j = 1, \ldots, k$, and $v = (t', n)$ with $t' \leq t$.

Definition 16. The operation

addMethod $c.m : t_1 \rightarrow t_2$ **to RoleClass** c'

maps **CH** into a new class hierarchy $(T, \prec, \tau, \mu', \omega, \pi_o)$, with $\mu'(c'') = \mu(c'')$ for all $c'' \neq c$ in T and $\mu'(c') = \mu(c') \cup \{c.m : t_1 \rightarrow t_2\}$, provided that

1. $c = c'$, $c' \in T \cap$ **RT**, ${}^\bullet c = \emptyset$,
2. for all $c'' \in c^\bullet$ and for all $c''.m : t''_1 \rightarrow t''_2$ in $\mu(c'')$, we have that $t''_1 \leq t_1$ and $t_2 \leq t''_2$ holds, and
3. there is no other method with signature $c'.m : t'_1 \rightarrow t'_2$ in $\mu(c')$.

Condition 2 ensures method conformance. This condition guarantees that all methods defined in a class have a unique name. It can be verified in practice by checking just the sets of equally named methods in direct super-classes of c'.

Observation 5. *Object migration, method and attribute extension, and deletion marking can also be shown to preserve the well-formedness and thus the type safety of a class DAG. The only exception is when no reference is made to any of the deleted entities (role class, relationship, oid). However, this type of reference can be trapped at the system level.*

For a complete proof of this observation, we have to show that, under the premise that the operations are defined, none of the conditions presented in the definition of well-formed type and class DAGs (Definitions 5 and 9) and none of the constraints of oid assignments and method signature assignments are violated.

5 Role class operations

In this section, we introduce some higher level ORM operations for creating and manipulating roles. The semantics of these operations are introduced in terms of compositions

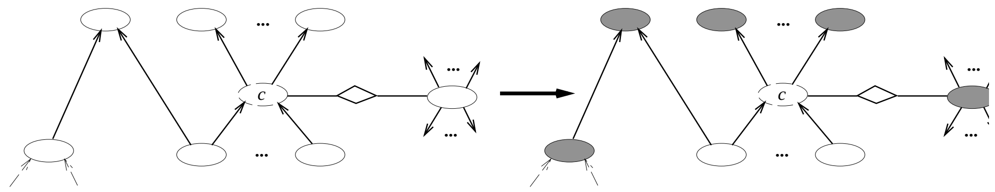

Fig. 5. Effect of deleting a role class c and the remaining shadow DAG

of the elementary role operations defined in Sect. 4. In the definitions below, we assume the existence of a well-formed class hierarchy $(T, \prec, \tau, \mu, \omega, \pi_o)$.

It is important to note that the role creation operations described in the following do not only physically create roles but also automatically instantiate their respective role classes and populate them with selected objects from their originating classes.

5.1 Role operations based on the grouping of objects

5.1.1 Creation of roles by enumeration

The first and simplest role-defining mechanism is *by enumeration*. Here, roles can be defined by identifying the role-creating objects by means of their object identifiers. The operation

```
createRoleClass $c$ asSubclassOf $c_1, \ldots, c_m$
    for $i_1, \ldots, i_{n_1}$ in $c_1$:
        $< roleClassBody >$
    for $j_1, \ldots, j_{n_2}$ in $c_2$:
        $< roleClassBody >$
    ...
    for $k_1, \ldots, k_{n_m}$ in $c_m$:
        $< roleClassBody >$
```

creates a new role class c into which oids $i_1, \ldots, i_{n_1}, j_1, \ldots j_{n_2}, \ldots, k_1, \ldots, k_{n_m}$ – from the super-classes $c_1, \ldots, c_m$ of class c – are migrated. This operation is the result of the following sequential composition of elementary role class operations defined in Sect. 4:

```
addRoleClass $c$ asSubclassOf $c_1, \ldots, c_m$;
migrateObject $i_1$ fromClass $c_1$ toSubclass $c$;
...;
migrateObject $i_{n_1}$ fromClass $c_1$ toSubclass $c$;
        $< roleClassBody >$

migrateObject $j_1$ fromClass $c_2$ toSubclass $c$;
...;
migrateObject $j_{n_2}$ fromClass $c_2$ toSubclass $c$;
...;
        $< roleClassBody >$

migrateObject $k_{n_m}$ fromClass $c_m$ toSubclass $c$
```

The operation is only defined if the constituent elementary operations are all defined. The statement $roleClassBody$ may include the addition of new attributes such as:

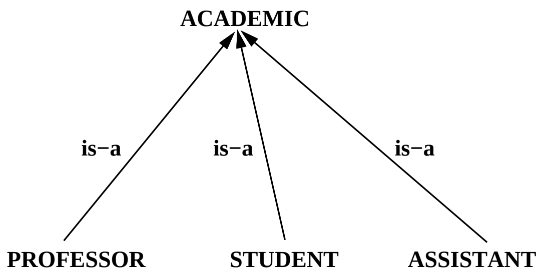

Fig. 6. A schema subportion for a university database

$$a_1 : t_1 = v_1, \ldots, a_l : t_l = v_l$$

to capture additional state information and new method signatures such as:

$$c.m_1 : t_1 \rightarrow t'_1, \ldots, c.m_n : t_n \rightarrow t'_n$$

and method implementations to capture new behavior for the identified object. The operations in the statement $roleClass\ Body$ extend the semantics of the high-level operation **createRoleClass** – as described above – by the following sequence of class hierarchy operations (described in Sect. 4.3).

```
addAttribute $a_1 : t_1 = v_1$ toRoleClass $c$;
...;
addMethod $c.m_1 : t_1 \rightarrow t'_1$ toRoleClass $c$;
...
```

This has as effect the creation of an additional facet for an object which retains its original object identifier.

In the ORM, we can generate role classes as generalizations of already existing classes. If we use generalization as a means to define a new role class, say c – common to a set of classes $c_1, \ldots, c_m$ – the extent of c would automatically contain the union of the extents of all classes $c_1, \ldots, c_m$. Consider the following statement in conjunction with Fig. 6 which describes a schema portion of a university database.

createRoleClass `Tutor` **asSuperclassOf** `Professor, Student, Assistant`.

The above statement has the semantics of the operation

addRoleClass `Tutor` **asSuperclassOf** `Professor, Student, Assistant`.

A new subclass relationship is introduced between the smallest common super-class of `Professor, Student` and `Assistant`, namely `Academic`, and the new class `Tutor`. The operation is undefined if there is no smallest common super-class of any *two or more* super-classes. The new role class factors out commonalities between existing classes. The semantics of the above operation result in the

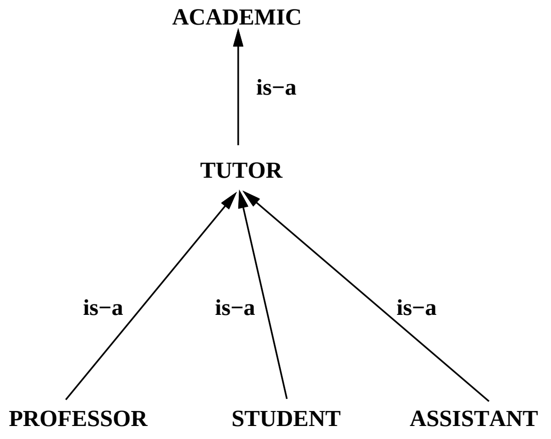

Fig. 7. Factoring out class commonalities and representing them as roles

properties of the class `Tutor` being the common properties of classes `Professor`, `Student` and `Assistant`. This situation is shown in Fig. 7. The extent of the new role class `Tutor` is formed by taking the *union* of the extents of the classes `Professor`, `Student` and `Assistant` according to Definition 11. The addition of the role class `Tutor` guarantees that all the re-arrangements in the class DAG result in a well-formed DAG (cf. Definition 9), as this operation is only defined if all its constituent elementary operations are well defined and, hence, the conjunction of their preconditions is satisfied.

The above operation is not flexible as it does not permit to exercise explicit control over subsets of objects in specialized classes which can migrate into the more generalized class. To selectively migrate objects from the extent of a specialized class to a generalized class we may use the following operation:

```
createRoleClass $c$ asGeneralizationOf $c_1, \ldots, c_m$
    for $i_1, \ldots, i_{n_1}$ in $c_1$:
    for $j_1, \ldots, j_{n_2}$ in $c_2$:
    ...
    for $k_1, \ldots, k_{n_m}$ in $c_m$:
with $< roleClassBody >$
```

This operation generates a subclass (role class) named c_i-c for each class of origin c_i (for $i = 1, \ldots, m$) and makes the role class c become their common parent class. The semantics of this operation are captured by the following sequential composition of elementary role class operations:

```
createRoleClass $c_1$-$c$ asSubclassOf $c_1$ for $i_1, \ldots, i_{n_1}$in $c_1$
...
createRoleClass $c_m$-$c$ asSubclassOf $c_m$ for $k_1, \ldots, k_{n_m}$
in $c_m$
addRoleClass $c$ asSuperclassOf $c_1$-$c, \ldots, c_m$-$c$
```

Moreover, if there exists a smallest common super-class c' of the classes $c_1, \ldots, c_m$, then c' becomes a parent class of c in the resulting class DAG.

To illustrate this concept, consider the following statement in ORM:

```
createRoleClass Tutor asGeneralizationOf Professor,
Student,                                   Assistant
    for $i_1, \ldots, i_{n_1}$ in Professor:
    for $j_1, \ldots, j_{n_2}$ in Student:
    for $k_1, \ldots, k_{n_m}$ in Assistant:
with $< roleClassBody >$
```

in conjunction with the schema subportion of a university database depicted in Fig. 6. This figure shows that `Academic`s comprise `Professor`s, `Student`s and `Assistant`s.

The above ORM statement creates a new role, namely `Tutor` (a kind of teaching assistant), for objects that belong to different classes, namely the classes `Professor`, `Student` and `Assistant`. Notice that after the execution of this statement the new role class `Tutor` is generated for the enumerated objects, contained in the role creation statement, as a direct subclass of `Academic`. This is due to the fact that `Academic` is the common (direct) superclass of all these three classes. The new role `Tutor` includes in its extent all the objects enumerated in the role creation statement. As `Tutor` is a role assumed by some and not all the objects in the classes `Professor`, `Student` and `Assistant`, further specializations of this new role class are also automatically generated by employing multiple inheritance to represent the roles `Professor-Tutor`s, `Student-Tutor`s and `Assistant-Tutor`s. This situation is depicted in Fig. 8.

5.1.2 Value-based creation of roles

Value-based roles may be defined using an expression based on attributes of the object in question. The semantics of value-based role class operations are defined in a similar manner as enumeration-based operations on the basis of the elementary operations introduced in Sect. 4. Value-based roles are defined according to the following syntax.

```
createRoleClass $c$ asSubclassOf | asGeneralizationOf $c'$ grouped by $e$:
    $< roleClassBody >$
```

where e denotes an expression referring to attribute values of particular attributes. Here a new role class c is created as subclass (super-class) of class c' and then all identifiers $i \in \pi_o(c')$ of objects (i, v) with $v = e$.

For example, if we wish to introduce a new role for educated employees (employees with a University degree), we would declare a role class as follows:

```
createRoleClass EducatedEmployee asSubclassOf Employee
  grouped by Employee.degree not null:
    $< roleClassBody >$
```

The statement $roleClassBody$ includes the definition of attributes and methods and is treated in the same manner as explained in the previous subsection.

5.1.3 Predicate-based creation of roles

Roles can be also created by means of predicates which must be satisfied by all the members of a role class. This distinguishing property of ORM is usually found in classification languages such as KL-ONE [BrS85]. This role creation mechanism forms predicate-based roles, which are defined according to some predicate P satisfied by all members of

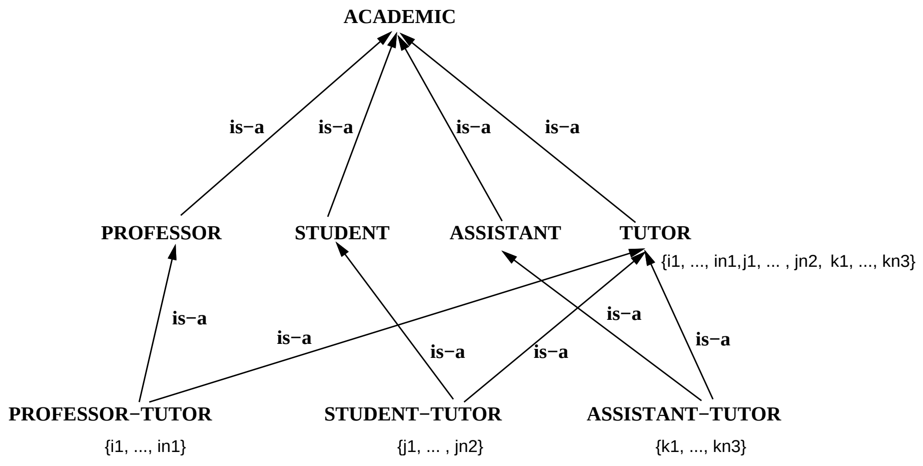

Fig. 8. Definition of roles for the schema in Fig. 6 involving multiple inheritance

any particular role class as specified by the following statement:

createRoleClass asSubclassOf | **asGeneralizationOf** c **grouped by** $case_1 \ldots, case_k$:
    $< roleClassBody >$

where each role is individually defined through a case-like statement $case_i$ using the following structure for each case:

c_i **is** P_i ,

where c_i are the new role classes and the predicates P_i refer to particular properties of the given class of origin c. The condition of the last case may be the keyword **other**, which applies only if all other cases failed. Note that, if the cases are not logically disjoint, the sequence of cases determines the role in which those objects matching multiple conditions are placed.

Again the semantics of this operation correspond to a sequence of **addRoleClass** c_i **asSubclassOf** c and **migrateObject** elementary operations such that only those objects that satisfy the condition P_i migrate to the new role class.

For example, in case that we wish to divide academics according to their pay rate, we would declare the following role classes:

createRoleClass asSubclassOf `Academic` **grouped by**
  `HighlyPaidAcademic` **is** `Academic.Salary` $> 100K$:
    $< roleClassBody >$
  `ModeratelyPaidAcademic` **is** `Academic.Salary` $> 50K$:
    $< roleClassBody >$
  `LowlyPaidAcademic` **is other** :
    $< roleClassBody >$

This facility introduces *parameterized role classes* which provide a way to define a template for a set of objects whose members behave in a similar manner. Different parameterizations of a parameterized class, e.g., `Academic`, produce different roles, e.g., `HighlyPaidAcademic`.

The above role-generating conditions are applied to and affect the extents of the classes mentioned in the role creation statement, e.g., `Academic`, at the time of execution of this statement. After the execution of this statement, the role-generating conditions act as demons on an *if-instantiated* basis and are evaluated "lazily" whenever a new object is instantiated and inserted into the extent of their associated role class, e.g., `Academic`. This leads to an automatic classification of newly created `Academic` objects into one of the three role classes `HighlyPaidAcademic`, `ModeratelyPaidAcademic`, and `LowlyPaidAcademic`.

5.2 Role operations based on inter-object relationships

The following role-creating operations allow one group of objects to be defined in terms of another in some other class in the DAG. The role-creating operations permit dynamic control over the patterns of inter-object linking and are also constructed as before using the elementary role operations defined in Sect. 4.

The semantics of role operations based on inter-object relationships correspond to a sequence of **addRoleClass** c_i **asSubclassOf** c and **migrateObject** elementary operations.

5.2.1 Reference-induced roles

Roles can be created by inter-relating object classes. The role operations described in this subsection exhibit the general form: $< object - set_1 > \ references \ < object - set_2 >$. The semantics of the reference-induced role creation operation are reminiscent of the division operation of the relational algebra and require that the operation returns a subset of objects from the $object\text{-}set_1$, where all the members of that subset are associated with all the members of $object\text{-}set_2$. The $object\text{-}set_1$ signifies a subset of the class extent of class c, whereas the $object\text{-}set_2$ corresponds to the oids $i_1, \ldots, i_k$ in the following operation:

createRoleClass c **as asSubclassOf** $c_1 < referenced - class >$
for $i_1, \ldots, i_k$ **in** c_2:
    $< roleClassBody >$

The reference can be in the form of a symbolic pointer such as an attribute of a particular class which may have its domain in another class, e.g., the class `PrivateCorporation` may have an attribute called `contracted-to` declared as "`contracted-to: setOf Government Branch`". Consider the following example where a new role is created for the class `PrivateCorporation` named

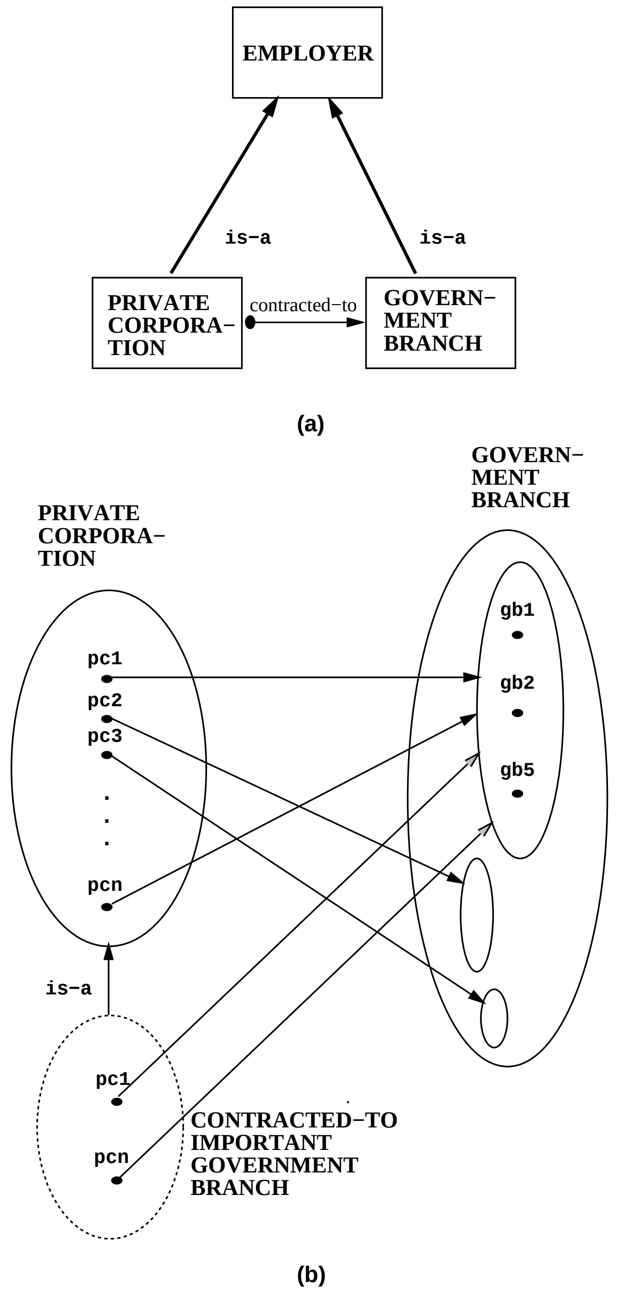

Fig. 9. Defining dynamic object roles via the use of references

`ContractedTo ImportantGvtBranch` in association with Fig. 9a.

createRoleClass `ContractedToImportantGvtBranch`
asSubclassOf `PrivateCorporation`
        `PrivateCorporation.Contracted-to`
for `gb1`, `gb2`, `gb5` **in** `GovernmentBranch`:
            $< roleClassBody >$

Figure 9b shows some sample data for the above situation. From the context of this figure it can be deduced that the `PrivateCorporation` objects $\texttt{pc}_1$ and $\texttt{pc}_n$ are `contracted-to` the `PrivateCorporation` specified in the above statement. As a result, the role `ContractedTo ImportantGvtBranch` is generated for the objects $\texttt{pc}_1$ and $\texttt{pc}_n$.

Alternatively, the reference may be substituted by a method in the body of c_1 which returns a set of objects of the type of objects belonging to the set $< object\text{-}set_2 >$. For example, instead of having a reference to objects of type `GovernmentBranch`, we may have a method which returns these objects. The method must obviously be declared in the body of the class which contains in its extent the object set $object\text{-}set_1$, i.e., `PrivateCorporation`. Consider the method signature[5] "`PrivateCorporation.Contractedto:()` $\rightarrow \mathscr{P}$`GovernmentBranch`" and the following statement:

createRoleClass `ContractedToGvtBranch`
asSubclassOf `PrivateCorporation`
        `PrivateCorporation.Contracted-to`():
            $< roleClassBody >$

The above statement creates a new role class, namely `ContractedToGvtBranch`, for *all* `PrivateCorporation` objects that are contracted to `GovernmentBranch`es.

We can also create a new role class of `PrivateCorporation` objects which are related to a particular `GovernmentBranch` by using the above statement in conjunction with a predicate:

createRoleClass `ContractedToGvtBranch`
asSubclassOf `PrivateCorporation`
        `PrivateCorporation.Contracted-to()`
grouped by `GovernmentBranch.name` = "Health":
        $< roleClassBody >$

5.2.2 Creation of roles through explicit linkages

There are roles which can be specified through explicit inter-object linkages which resemble dynamic role-relationships in KL-ONE [BrS85]. In the ORM, a relationship may be used to act as a predicate and capture the commonality among a set of individual role-playing objects. Therefore, relationships which associate two classes of objects may be used to generate a new role for a subset of the objects which are contained in the extent of the class at their point of destination. Thus, the operation

createRoleClass c_r **from** $r(i,$ **setOf** $c_d)$:
        $< roleClassBody >$

creates a new role class c_r as a subclass of class c_d and migrates to c_r all the objects in the extent $\pi_o(c_d)$ of c_d that are related to i in its relationship r, i.e., $\pi_o(c_r) = \{i' \in \pi_o(c_d) | (i, i') \in r\}$, provided that c_d is a subclass of the destination **dest**(r) of relationship r and i is an element in the extent $\pi_o(c)$ of some subclass c of the source class **src**(r). We call c_r the *role class generated by* r *wrt* c_d *and* i.

Additionally, predicates relating a particular object identifier in the source (or destination) class of a relationship to objects in its destination (or source class) are also admissible. This operation is exemplified by the following situation. Consider the relationship type `hires(Employer, setOf Employee)` between the types `Employer` and `Employee` (Fig. 10). This relationship type accepts the class of `Employer` objects as its source and `Employee` class

[5] $\mathscr{P}A$ denotes the powerset of A, i.e., the set of all possible subsets of A.

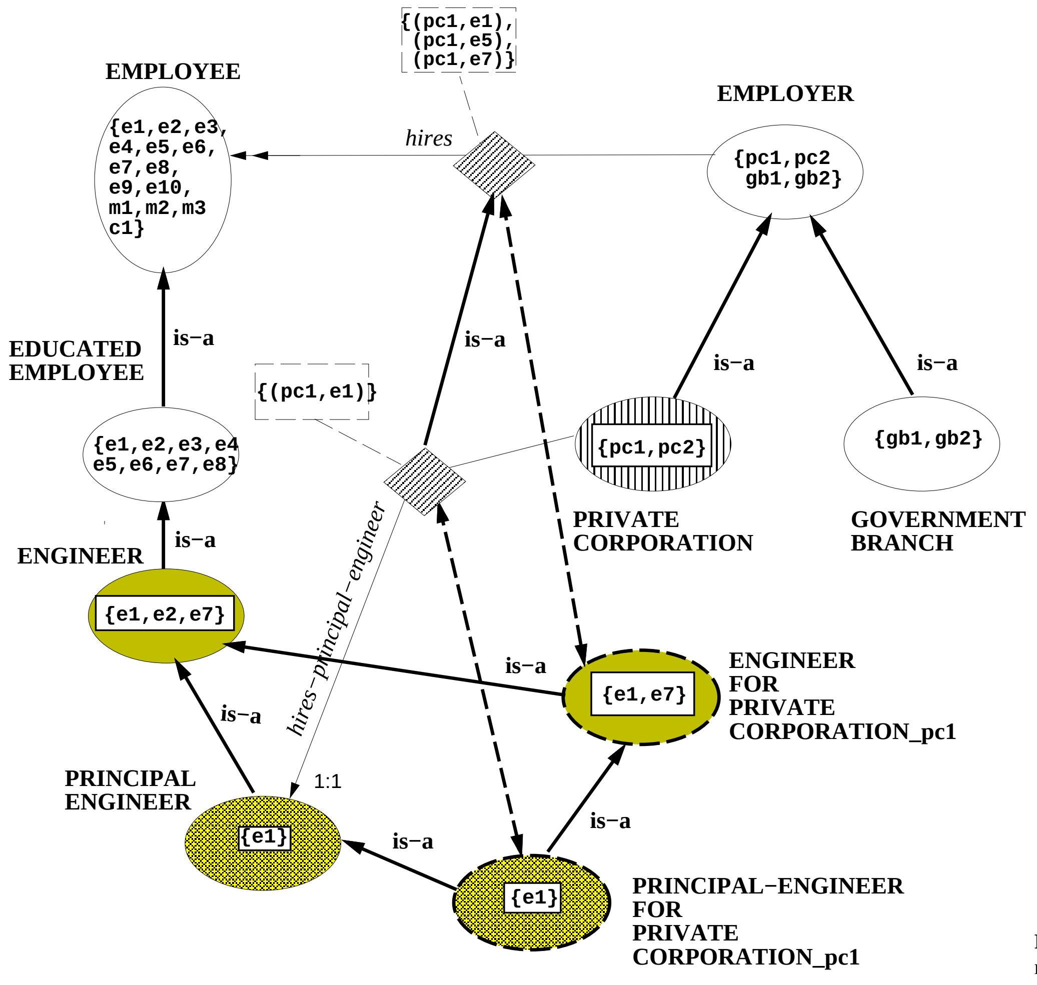

Fig. 10. Defining dynamic object roles via the use of relationships

objects as its destination (indicated by the presence of a double arrowhead) and imposes the constraint that a single `Employer` object may be related to a set of `Employee` objects[6].

The following statement:

createRoleClass `EngineerForPrivateCorporation_pc`$_1$
from `hires`(pc1, **setOf** `Engineer`):
$< roleClassBody >$

generates a new role called `EngineerForPrivateCorporation_pc`$_1$ and populates it with the `Engineer` objects that are associated with the particular `Private Corporation` identified by the object identifier `pc`$_1$. The new role is a subclass of the class `Engineer` which, in turn, is a subclass of the destination of the role-defining relationship `hires`. This situation is illustrated in Fig. 10, where oids `e1` and `e2` in the extent of class `Engineer`, as they are the only ones that also occur in the pairs of the extent of relationship exhires. The presence of the double-headed dashed arrow indicates the generation of a new role via the use of a role-defining relationship, e.g., `hires`. Although the relationships in this figure conform to the relationship types in Fig. 2, we have chosen to represent them as unidirectional for reasons of simplicity.

In Fig. 10, we have introduced the relationship type `hiresPrincipalEngineer` as destination class as a subtype of the relationship type `hires`. This subtype relationship associates `PrivateCorporation` with `PrincipalEngineer` objects. By employing the `hiresPrincipalEngineer` relationship, we may now generate a new role called `PrincipalEngineerFor-PrivateCorporation_pc`$_1$ for the `PrivateCorporation` identified by the object identifier `pc`$_1$. It is interesting to note that, since the role `EngineerForCorporation_pc`$_1$ was created by the relationship type `hires` and `hires PrincipalEngineer` is its subtype (by virtue of its definition in Fig. 10), the class associated with that role, namely `PrincipalEngineer ForPrivateCorporation_pc`$_1$, is a subclass of `EngineerForCorporation_pc`$_1$. This fact is checked by the system which applies the following invariant that is based on the *restriction* relation in KL-ONE.

Invariant 1. *If a class c_1 which is the (subclass of the) destination of a relationship type r_1 has a subclass c_2, and if a relationship type r_2, defined as having c_2 as (a subclass of) its destination, is a subtype of r_1, then every role class generated by r_2 wrt c_2 and some oid i in the extent of a subclass of the source of r_2 is a subclass of the role class generated*

[6] In fact, this relationship is a polymorphic one, since, according to the principle of argument contravariance, its argument domains may be expanded by subclasses of *either* its origin *and*/or its destination.

by r_1 wrt c_1 and i, provided that the source class of r_2 is either the same or a subclass of that of r_1, say c_3. Moreover, r_2 satisfies all the constraints imposed on both r_1 and r_2.

Observe that in Fig. 10 the lower and upper bounds which define the range cardinalities for the set of objects generated by the role-defining relationship are both set to 1, meaning that there is only a single `PrincipalEngineer ForPrivateCorporation`, indicated by a single-headed arrow in Fig. 10. The class DAG shown in this figure satisfies Invariant 1 with the following binding of variables: c_1 = `Engineer`, c_2 = `PrincipalEngineer`, and c_3 = `PrivateCorporation`, while r_1 = `hires` and r_2 = `hiresPrincipalEngineer`.

5.2.3 Role creation through reasoning

Finally, as an analogy to role-defining relationships, we may have roles generated through *reasoning*. We exemplify this situation by using an example relating to a loan-securing application, whereby a relatively "intelligent" object-oriented database system (employing production rules) helps a human intermediary with respect to the factors which must be satisfied by a bank customer to secure a loan of a certain type. For this purpose we shall use a simple rule-based sublanguage.

Rules can derive new patterns of associations among objects of selected classes. This situation is similar to the use of triggers, which is covered in Sect. 5.3. Consider, for instance, how the following two rules r_1 and r_2 operate in the context of Fig. 11. In this figure, we assume that the classes `Employee` and `Customer` are populated by schema type instances, whereas the classes `SteadyJobCustomer`, `QualifiesForHomeLoan` and `QualifiesForCar Loan` are roles created from these classes.

```
r1: if   employedBy(Customer, Employer) is
    GovernmentBranch
    or   Customer.LengthOfEmployment >= 5
    then createRoleClass SteadyJobCustomer(Customer)
         asSubclassOf Customer:
    < roleClassBody >
r2: if   SteadyJobCustomer(Customer)
    and  Customer.Disposable-Income >
         (2.5 * repaymentRate)
    then createRoleClass QualifiesForHomeLoan
         (Customer) asSubclassOf Customer:
    < roleClassBody >
    else createRoleClass QualifiesForCarLoan
         asSubclassOf SteadyJobCustomer:
    < roleClassBody >
```

These two rules describe a situation where `Customer`s (a subclass of `Employee` class) are characterized as `Steady JobCustomer`s on the basis of their employment. Subsequently, customers are assessed whether they satisfy the conditions required for securing a particular type of loan, e.g., home or car loan. The antecedent of rule r_1 is the polymorphic relationship type `employedBy` in Fig. 2, which associates objects of type `Employee` (and consequently `Customer` as a subclass of `Employee`) with objects of type `Employer` (or subtypes thereof).

Rule r_1 is a conditional statement that evaluates to true or false, following the computational semantics of standard rule-based systems, e.g., Prolog. If the rule evaluates to true, then the object `Customer` is effectively asserted (added) to the role-defining class `SteadyJobCustomer`. Thus, class `SteadyJobCustomer` is created dynamically after the execution of rule r_1 and contains a subset of the objects in the class `Customer` which satisfy the antecedents of the rule r_1. The entire situation is depicted in Fig. 11, which shows how the execution of the rules r_1 and r_2 leads to the generation of the three additional `Customer` roles `SteadyJobCustomer`, `QualifiesForHomeLoan` and `QualifiesForCarLoan`. Notice that the roles `QualifiesForHomeLoan` and `QualifiesForCar Loan` are mutually exclusive according to the definitions in rule r_2. Mutually exclusive roles is the subject of the following section.

5.3 Additional role operations

Additional ORM operations on role classes are defined below. In contrast to the operations covered in the previous, these operations accept already existing roles as input. The operations either operate on the extents of role classes or on an entire role class. In the former case, the role operations assume a role class as input and add/remove or migrate objects to/from it, whereas, in the latter case, they accept a role class as input and operate on its entire extent as a whole.

The set of role operations described in the following is representative of the possible operations on roles. There are other simpler operations which traverse the class DAG and compute role transitive closures such as *find-roleSet*, *find-Class-of-origin*, *find-parent*, *find-descendants* of a role and so on, which together with elementary operations covered in Sect. 4 help construct the operations that follow. We will not consider them any further, as their semantics and implications are easily understood.

5.3.1 Assuming a role

The following statement illustrates how an object may assume a new role.

assume $RoleClass$ **for**
$i_1, ..., i_n$ | $< ValueBasedExpression >$ |
$< PredicateBasedExpression >$;

An object may assume an already existing role by using this operation. The convention is that an object cannot assume a role unless a role-defining class for this role already exists. The statements $< ValueBasedExpression >$ and $< PredicateBasedExpression >$ have a syntax similar to that introduced in Sect. 5.1. Consider, for example, the following statements in the context of Fig. 4.

```
assume Engineer for
EducatedEmployee grouped by EducatedEmployee.Degree
= ''Engineering''
```

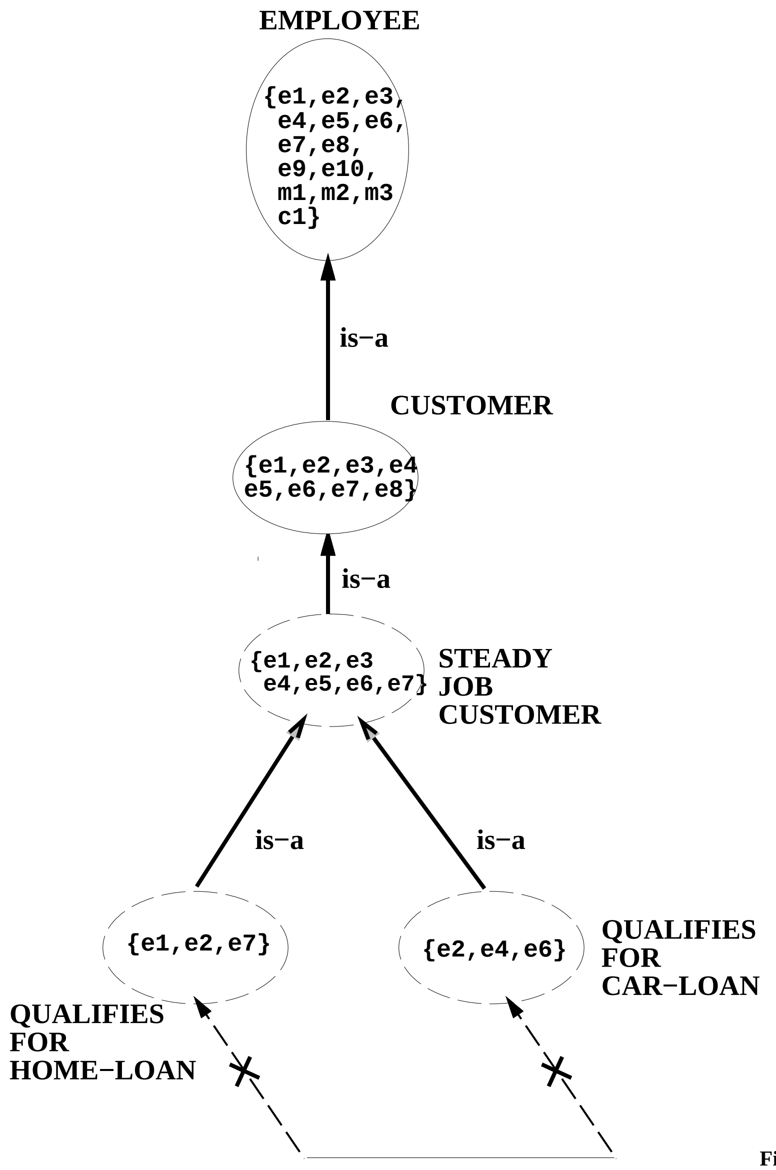

Fig. 11. Defining dynamic object roles via reasoning

5.3.2 Delaying operations and role transformation

5.3.2.1 Blocking roles. Role interaction is taken to mean how objects in one role class extent may interact with objects in another role class. Role interaction is mainly exemplified by the concept of mutual exclusion, which leads to role blocking. Two roles having a common ancestor are *mutually exclusive* if an object is prohibited from joining both of these roles and is forced to select either one.

Consider, for example, the `EducatedEmployee` objects in Fig. 12, which may wish to assume the additional roles of `Engineer`, `Academic` and `SocialWorker` objects. It might be desirable to block objects of type `Engineer` and `Academic` from being `SocialWorker` objects at the same time. Thus, we designate their respective role classes as being mutually exclusive, i.e., objects which appear in the extents of the classes `Engineer`, or `Academic`, are not allowed to appear in the extent of class `SocialWorker`, and vice versa.

constrainRoleClass `Engineer`, `Academic`, `SocialWorker` **for**
`EducatedEmployee`
with `Engineer Academic` **mutex** `SocialWorker`

The previous statement introduces mutually exclusive roles (and objects). This is indicated in Figs. 12 and 11 by arcs, intercepted by the symbol X, that are directed from the blocking towards the blocked roles. Obviously, nothing prevents an `Engineer` object from simultaneously being an `Academic` (although this is not shown in this figure). This type of role-blocking may be thought of as a set exclusion operation.

Invariant 2. *If two or more role-defining classes are mutually exclusive, then all of their subclasses are also mutually exclusive.*

This invariant guarantees that descendants of the `Engineer` role objects, e.g., `EngineerShareholder` objects, do not become members of the class `SocialWorker`, and vice versa.

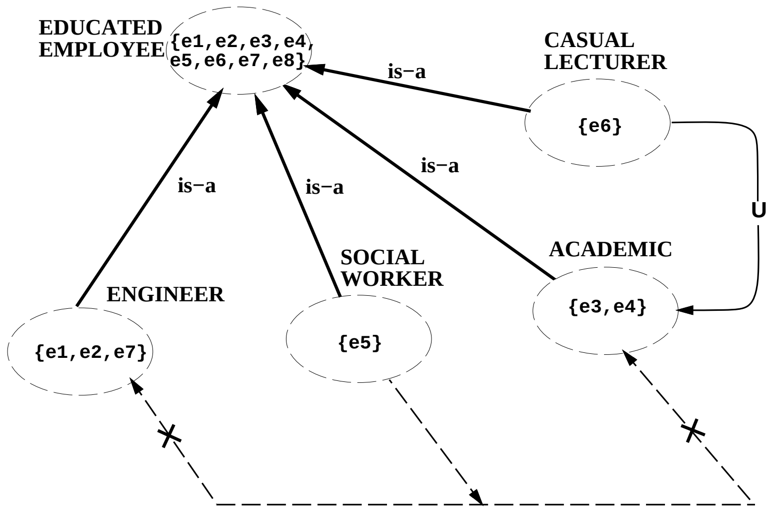

Fig. 12. Role delay and transformation

5.3.2.2 Automatic role transformation. The most common mechanism for the transformation of roles in the ORM is provided by means of triggers. A trigger in the ORM is thought of as a monitor on a data item (which may change value) or as a logical condition-action pair. When the condition is fulfilled, the action is executed. Consider the following example.

```
trigger TransformToAcademic when this
(CasualLecturer.Degree = ''Phd''
    and CasualLecturer.Appointment = ''FullTime'')
    this CasualLecturer becomes Academic
end trigger
```

The previous example shows how an object with `Casual Lecturer` role may become an `Academic`. This trigger, like a method, is defined in the body of the class `CasualLecturer`. The result of this operation is that the object in question, say identified by the oid e_6, is removed, by invoking operation **removeObject**, from the extent of class `CasualLecturer` and joins the extent of class `Academic`, by invoking the operation **addObject**. This relationship is signified by the horizontal arcs intercepted by the symbol $\supset$ and directed from the current role, i.e., `CasualLecturer` to the *target role* (the role class after the current object's role is changed; see Fig. 12). The ORM upgrades the object in question automatically to the structure and behavior of the target role class. This may involve deleting and adding properties and behavior. The type-safety invariants which govern this type of transformation have some resemblance with those used for the generic function *update-instance-for-different-class* in CLOS [KEE89].

The trigger conditions are not only applied to the current extents of the classes involved in the condition part of the trigger, they are also applied lazily to any objects joining the extents of these classes at a later stage.

5.3.3 Controlling role changes

The following operations control how objects may change the current role that they are playing. The simplest operation is to relinquish a current role, for an object or a set of objects, in favor of some other specified role. The following statement illustrates how an object may relinquish a role.

relinquish $RoleClass$ **for**
    $i_1, ..., i_n \mid < ValueBasedExpression > \mid$
    $< PredicateBasedExpression >$
    [**resume** $RoleClass$] | [**resume** $RoleClass$ **when** < *event* >];

An object may relinquish its current role and assume either: its immediate parent role (default case if the *resume* statement is missing) or a specified role in its role-set by means of the *resume* statement, or finally, revert to the class were it originated from. The operation relinquish is implemented by invoking the elementary operation **removeObject** for objects that satisfy the relinquish criterion. Consider the `Shareholder` object with oid e_3 in the context of Fig. 4 and the following statement:

relinquish $Shareholder$ **for** e_3;

this statement results in relinquishing the role `Shareholder` for the object e_3. This object then is deleted from the extent of this role class. This implies that the object with oid e_3 still keeps its role as `EducatedEmployee`.

In the following, we will explain the use of a simple *resume* statement in conjunction with the *relinquish* operation. The use of an event-triggered resumption of a role will be explained when we consider role suspension. The statement:

relinquish $Shareholder$ **for** e_3
    resume $Employee$;

results in the object with oid e_3 being removed from all class extents between `Employee` and `Shareholder`. This implies that this particular object abandons all of its roles and reverts to its class of origin. Role relinquishing (and suspension, see below) are governed by the following two invariants which apply automatic coercion of relinquished roles with existing role objects in the DAG and, in general, control how an object can *change* a role it currently plays. The above statement is implemented by multiple invocations of the operation **removeObject**. The operation *resume* utilizes, in general, the elementary operation **addObject**, **migrateObject**, provided that the selected oid does not already exist in the extent of the class specified by the **removeObject** operator.

Invariant 3. *If an object relinquishes (suspends) its current role, it also relinquishes (suspends) all of its current role's descendent roles, if any.*

Invariant 4. *An object that relinquishes its current role may assume its parent role, or any role in its current role set, provided that this role is an ancestral role of the role that the object released. We call these roles valid roles. Alternatively, it may assume any other role which is a direct or indirect descendant of a valid role in its role set provided that there exists a non-exclusive relationship between any of the roles in its valid role set and the one just assumed.*

Invariant 4 deserves some explanation. Consider, for example, the `EngineerShareholder` object with oid e_1. If the application semantics dictate that this objects should change its role from `EngineerShareholder` (i.e., $relinquish\ EngineerShareholder$) to `Academic` (i.e., $resume\ Academic$), then the sequence of role changes corresponding to these two operations is permissible. This object is removed from the extent of the role `Engineer Shareholder`, remains in the extent of the roles `Employee`, `EducatedEmployee`, `Engineer` and `Shareholder` and joins the extent of the role `Academic`. This new role is allowed, as its parent role is a valid role, i.e., `EducatedEmployee`, in the role set of the object e_1. However, object e_1 is not permitted to join the role `Social Worker` after dropping the role `EngineerShareholder`. This is because the role `Engineer` in its role set and the new role `SocialWorker` clash by definition, i.e., they have been defined as mutually exclusive.
The following operation is used for synchronization purposes, mainly in conjunction with a trigger-like event specification. It results in suspending further actions of an object (under a particular role) until a certain event occurs.

suspend $RoleClass$ **for**
$i_1, ..., i_n \mid ValueBasedExpression \mid$
$PredicateBasedExpression$
resume $RoleClass$ **when** $< event >$;

This operation is a further specialization of the operation *relinquish*. The main difference between these two operations is that objects specified by the operation suspend may remain suspended or "frozen" for an indefinite period of time, as the application demands, and then resume their previous role by means of the operator *resume* only when a pre-specified event has occurred. Such objects may be allowed to change role only in accordance with the invariants 3 and 4.
Consider the following example.

suspend `Academic` **for** `Academic.Degree` $\neq$ "Phd"
and `Academic.YrsOfService` $\geq$ 3
resume `Academic` **when this** (`EducatedEmployee.Degree = ''Phd''`
and `EducatedEmployee.Appointment = ''FullTime''`)

The above statement specifies that an object of type `Academic` may lose its academic status for an indefinite period of time and resume its parent, i.e., `EducatedEmployee`, role until an event occurs, i.e., a condition is fulfilled, which makes it possible for this object to revert to its suspended role.

5.3.4 Sharing and solidifying roles

Roles act in general like snapshot objects and cannot outsurvive the duration of the application program that created them. Normally, there is no need for all roles to become globally persistent and hence visible by other application programs and users. However, in several situations, there are some roles which might be useful for a large number of users and application programs. To provide for additional modeling flexibility, the ORM allows roles (and individual role objects in their extent) to be shared between applications or to be promoted to persistent types and objects, respectively.

To allow roles to be shared between applications, we use the following operation:

share $RoleClass$ **with** $ap_1, \dots, ap_m$
[**for** $i_1, \dots, i_n \mid ValueBasedExpression \mid$
$PredicateBasedExpression$]

This operation extends the visibility of $RoleClass$ from its local application context to other applications ap_i (for $i = 1 \dots m \geq 1$).

Role classes and selected objects in their extent may be made persistent by invoking the operator *solidify*. Solidified role classes have their definitions become automatically part of the type DAG and thus can no longer be distinguished from other database classes. In other words, this operation results in the evolution of the object base as it automatically adds new types and their respective instances. The syntax of this operation is as follows:

solidify $RoleClass$
[**for** $i_1, ..., i_n \mid ValueBasedExpression \mid$
$PredicateBasedExpression$]

When making a role class persistent, other role classes may also be solidified transparently. If a role is solidified, all objects included in its extent must also become permanent. This process is governed by the following invariant.

Invariant 5. *To solidify (share) a role, we must also solidify (share) all roles appearing in all reachable paths between the role's class of origin and the defining class for that role. Moreover, all role-defining classes referred to in the method signatures and in the role's definition statements must also be made permanent (sharable).*

6 Related work

In the database literature, the idea of role modeling was first exemplified by the seminal work of Bachman on the role data model [BAC77]. The definition of the role concept in Bachman's model is taken from the theatrical context and is used to mean a behavioral pattern which may be assumed by modeled entities in a problem domain. The role data model introduced a static part for modeled objects, called the *entity*, which establishes existence, and a dynamic type, called the *role-class*, establishing behavior for that entity.

In the following, we summarize current research activities which share some concern about the evolution of objects and outline their differences from roles. Of particular interest to us are research activities in connection with views, as they are also derived from already existing DAG classes.

6.1 Views and roles

Views are used in object-oriented systems to define logical partitioning of classes according to user authorization and access privileges. Most approaches for view definition suggest the use of query language expressions for specifying virtual classes, i.e., views [He90], [ScLT91], [ScST94], [RUN92], [KIM95] from stored classes in the class DAG. Alternative approaches use special object algebra operators for defining views [RUN92]. The definition of a view consists of all schema elements that can be included in a virtual class and a query (algebraic operation) that defines how the view is populated by selecting instances from one or more stored schema classes (and/or other views). The extent of these view classes is usually not stored explicitly but rather computed from the view-defining query upon request [ScLT91]. In contrast to views, roles have a different objective: their purpose is to support dynamic object migration and re-classification – in a way that does not affect the database schema. The observable differences relate to semantic preservation (views) vs. semantic upgrades (roles); object generation and updates (views) vs. strict object preservation (roles); differences in the treatment of object identity; and, finally view vs. role materialization techniques. These issues are addressed briefly in the following.

Views are in general semantics preserving as they introduce only new information as computed attribute values, e.g., by merging existing attributes, or by hiding attributes and importing existing schema classes [AbB91], [ScLT91], [RUN92]. In contrast to views, roles tend to refine semantics by attaching additional meaning – in the form of new (or unanticipated) behavior – to objects that have a special meaning for an application. Additional semantics are correlated with existing object semantics.

Views can be object-generating as they may generate new types of objects (not included originally in the schema) if view definition queries involve more than one stored classes, e.g., join operations. Some approaches escape this trend by adhering to the concept of object preservation [ScLT91], [ScST94], [RUN92]. Views are also used to create new objects and update already existing objects, provided that there exists a one-to-one correspondence between a materialized instance of a view and the stored class on which the view is defined [KiKS92]. In the ORM, new objects are created only through classes in the DAG and are automatically re-classified into role classes – provided that they meet the role conditions.

Another important difference is the treatment of oids. With views, the most common approach is to create a new oid for each materialized view [KIM95], [AbB91]. This is a direct consequence of the fact that views may be defined by joining two or more stored classes (and/or views), or alternatively by hiding attributes from stored classes. In both cases, it is not possible to identify the materialized view classes in terms of the oid(s) in the stored classes from which they were produced. Hence, mechanisms are introduced to map the oid assigned to each materialized view to the oid(s) of the instance of the stored class(es) from which the view originates. These concerns are not shared by roles, as their identifying instances share the same oid with objects in their classes of origin.

Finally, view objects are normally generated every time the view definition query is invoked, while the role mechanism allows the dynamic formation of groups of objects as a result of monitoring and reacting to conditions which apply to a class of objects as a whole. The role-generating conditions are evaluated lazily (incrementally) [FeMZ94] whenever a new object is inserted into a DAG class.

6.2 Roles and schema evolution

The management of objects that evolve dynamically over time has been of some concern to research activities in the context of schema evolution [SkZ87], [ZDO90] [BAN87], [ZIK91].

Schema evolution addresses the problem of schema updates applied to an object base due to changing application requirements. There are two approaches to schema evolution: *conversion* and *versioning*. The former restructures the affected instances to conform to classes which have been modified during the schema evolution process [BAN87], [BRE89]. These changes are introduced at the schema level and are propagated to all instances of a type in the database that is affected by the changes. The objective is not only to provide mechanisms for schema updates but also to make certain that the structural and behavioral consistency of types and objects is respected [ZIK91]. This requires writing transformation functions to allow compatibility with application programs that utilize the original classes and instances. To avoid the pitfalls of class redefinition and conversion, a class versioning approach can be employed, whereby the existing class definition is not changed, but rather a new version of the class definition is created which incorporates the required changes. Instances and applications are then associated with a particular version of a class and the runtime support is responsible for simulating the semantics of the new class interface on top of instances of the old, or vice versa [SkZ87], [BjH89]. These approaches guarantee minimal compatibility, as they rely on the existence of exception handlers to emulate instances and provide default values that are present in one version and not in another. A variant of this approach suggests the changing of the schema as a whole rather than the piecemeal changing of individual classes [LeH90], [MoS93]. The approach of [MoS93], in particular, provides facilities for allowing database schemas to evolve both forward and backwards for each class change. A serious problem with this approach is that users and application developers have to define their own update and backdate methods.

In contrast to roles, the emphasis with schema evolution is placed on mechanisms that facilitate the migration of an entire class population to a new (evolved) class by either dropping or re-adjusting the old class definition. In general, schema evolution requires human intervention, i.e., the involvement of a database administrator, to apply the schema changes and check the consistency of the database schema. Such issues do not affect roles. However, research activities relating to object-oriented schema type evolution [ZDO90], [SkZ87] and parametrized primitives for schema updates in O_2 [ZIK91], in particular, have influenced our work on dynamic objects.

6.3 Roles and other approaches

The notion of role has also been used in expressing potential object states and behavior in the context of office information systems [PER90]. Roles in this model are static: they are specified in their totality at the schema level and are not created dynamically on demand. In this model, the behavior of an object can be derived by means of an abstract state which defines all of its active roles and roles instantiations. This particular model places emphasis on defining mechanisms for coordinating multiple instantiations of a single role, on specifying rules for expressing valid role-state sequences and on placing constraints on the possible life cycle of objects.

Aspects [RiS91] is another approach which attempts to address dynamic object behavior and schema evolution in general. Aspects are used in a strongly typed object-oriented environment, which introduces sharp dichotomy between abstract data types and implementations to support multiple independent object views. The key difference between aspects and the role model proposed in this paper is that entities in the aspect model may have many different unrelated types, and unlike roles they do not simply evolve from one type into another by means of pre-specified conditions or inter-object relationships.

Remote similarity exists with the concept of multiple substitutability as defined in [MoZ92], where the principle of information hiding is used so that an object can either be addressed as any of its constituents or as a whole, depending on the situation. The part-of relationship is employed to enforce the different fashion constructs that an object may obtain. This allows an object to behave as any of its constituent fashions and to route a message directed to it to the appropriate fashion object. Fashions do not share the same concerns with views. They are specified statically (their binding occurs at runtime), have all different oids, and are not concerned with object migration.

Some similarities exist between the ORM and the Fibonacci object-oriented database programming language [ALB93]. The main concern of Fibonacci is to provide a sound programming environment, where objects may acquire new types and behavior while retaining their identity. Major concerns lie in designing an environment supporting strong typing and late binding in conjunction with message dispatching to resolve ambiguities due to multiplicity of types at runtime. Objects are not created or manipulated directly: they are always created and accessed through their roles. In this way, roles essentially become part of the database schema and are not transient as in the ORM. Fibonacci focuses mainly on implementation issues such as the use of delegation for implementing inheritance and message passing. No attention is given to linguistic facilities needed to support automatic migration of objects between classes; forming roles as groupings of objects and creating roles from inter-object relationships. Moreover, in Fibonacci, the issue of controlling role semantics in accordance with application-specific events is also not considered. All these concept are supported by the ORM and have influenced its design considerations.

6.4 Roles and programming languages

The Common Lisp Object System, CLOS [KEE89], has some similarities with the ORM. It offers a generic `change-class` method to facilitate switching an instance from one class to another. Methods can be added or deleted from existing classes and new classes can be added to an application. The meta object protocol of Common Lisp specifies clearly what happens if the target class supports additional or less attributes or methods. However, except for the semantics inherent in interpreters and compilers, no formal definition has been given for this framework and static type-checking is of no concern to this approach.

7 Summary and future work

The inability of contemporary object-oriented database systems to represent evolution and re-configuration of individual objects may lead to a loss of modeling assumptions and inter-object dependencies. This limitation makes the maintenance of consistency of dynamic objects almost impossible. In this paper, we have presented an extension to the object-oriented paradigm which supports a natural way of representing object dynamics and addresses such shortcomings. More specifically, we introduced the ORM as an extension of object-oriented databases to support unanticipated behavioral oscillations for individual objects, or groups of objects, that have many types and share a single object identity. Upgrowths of behavior in the ORM are known as roles that objects play which can be assumed and relinquished dynamically to reflect shifting modeling requirements.

The purpose of the ORM is to add more modeling power and flexibility to the object-oriented approach by capturing different kinds of object dynamics. They do so by being based either on conditions which apply to an individual object, groups of objects from a class, or on explicit/implicit inter-object relationships. The ORM allows dynamic object features to be fully synthesized with conventional object-oriented database characteristics.

The ORM linguistic facilities provide operations that support pre-existing objects to change their type, while retaining their original identity. They control such forms of object evolution in accordance with application semantics, while respecting the structural and behavioral consistency of the typed objects. Object groups may be constructed on the fly as a result of monitoring and reacting to conditions which apply to the scope of a class as a whole. Such conditions are evaluated “lazily” (incrementally) whenever a new object is inserted into a DAG class. The ORM linguistic features introduce modeling flexibility and give applications more organizational clarity by simplifying the design requirements of complex applications that need to create, relinquish and manipulate dynamic objects.

An initial prototype of the ORM was implemented in the object-oriented database system ONTOS 2.2 and the programming language C++. An extension of this early prototype based on an amalgamation of the ONTOS implementation and the expert system shell CLIPS has also been implemented. Its purpose is to provide more natural and powerful primitives for the ORM. In this way, reasoning facilities can be used for defining and manipulating role objects. For example, rules are used as a way to define role classes in terms of associations between selected objects from existing classes. Currently, a more flexible re-implementation of the ORM in ObjectStore, using its meta-object protocol, is underway.

The ORM can be extended by capturing another aspect of application semantics, namely the temporal aspect. The ORM was developed under the assumption that the object base contains only the current ”snapshot” of the role data we are interested in. This may prove to be too restrictive in many situations where it is desirable to maintain terminated roles, or old role versions, and associate them with current roles for reasoning purposes. We are currently investigating an adaptation of this scheme to the ORM.

Acknowledgements. We wish to thank the three anonymous reviewers for their valuable and constructive remarks and suggestions that helped improve the presentation and quality of the current version of this paper. We also wish to thank H.-W. Schmidt for his insightful remarks and fruitful discussions that resulted in improving this paper. The helpful comments and suggestions of Athman Bouguettaya and Dave Edmond are also greatly appreciated. This work was financially supported by the European Union under contract no. ECAUS003 and by the Department of Industry, Technology and Regional Development (DITARD), Australia under grant ref. no. 4556 as a European-Australian collaborative research project on Information Systems Interoperability.

References

[AbB91] Abiteboul S, Bonner A (1991) Objects and views. In: Clifford J, King R (eds) Proc SIGMOD-91, ACM, New York, pp 238-247

[ALB93] Albano A, et al. (1993) An object data model with roles. In: Agrawal R, Baker S, Bell D (eds) Procs. 19th VLDB Conf., Morgan Kaufmann, San Mateo, Calif., pp 39- 51

[ATK89] Atkinson M, et al. (1989) The object-oriented database system manifesto. In: Kim W, Nicolas JM, Nishio S (eds) Proc 1st Deductive Object-Oriented Database Conf., Elsevier, Amsterdam, pp 223-240

[BAC77] Bachman CW (1977) The role concept in data models. In: Procs. VLDB 77 Conf., ACM SIGMOD Record 9:464–476

[BAN87] Banerjee J, et al. (1987) Semantics and implementation of schema evolution. In: Dayal U, Traiger I (eds) Object- Oriented Databases, ACM, New York, pp 311-322

[BER91] Bergstein P (1991) Object-preserving class transformations. In: Paepcke A (ed) Proc OOPSLA ‘91 Conf., ACM, New York, pp 299-313.

[BLA87] Black A, et al. (1987) Distribution and abstract data types in Emerald. IEEE Trans. Software Eng 13:65-76

[BjH89] Björnerstedt A, Hulten C (1989) Version control in an object-oriented architecture. In: Kim W, Lochovsky F (eds) Object-Oriented, Concepts, Databases & Applications, Academic-Press, London, pp 451-486

[BrS85] Brachman R, Schmolze J (1985) An overview of the KL-ONE representation system. Cognitive Sci 9:171-216

[BRE89] Bretl R, et al. (1989) The GemStone data management system. In: Kim W, Lochovsky F (eds) Object-Oriented Concepts, Databases & Applications, Academic-Press, London, pp 283-308

[FeMZ94] Ferrandina F, Meyer T, Zikari R (1994) Implementing lazy database updates for an object database system. In: Bocca JB, Jarke M (eds) Proc 20th VLDB Conf., Morgan Kaufmann, San Mateo, Calif., pp 261-272

[He90] Helier S, Zdonik S (1990) Object views: extending the vision. In: Proc Int’l Conf. on Data Engineering, IEEE, Los Alamitos, Calif., pp 86-93.

[KaLR92] Kanellakis P, Lecluse C, Richard P (1992) Introduction to the data model. In: Banchilhon F, Delobel C, Kanellakis P (eds) Building an Object-Oriented Database System: The Story of O2, Morgan-Kaufmann, San Mateo, pp 61-76

[KEE89] Keene SE (19989) Object-oriented programming in Common Lisp. Addison-Wesley, Reading, Mass.

[KIM89] Kim W (19989) A model for queries in object-oriented databases. In: Apers PMG, Wiederhold G (eds) Proc Int’l Conf. on Very Large Databases, Morgan Kaufmann, San Mateo, Calif., pp 423-432.

[KIM95] Kim W, Kelley W (1995) On view support in object-oriented database systems. In: Kim W (ed) Modern Database Systems, Addison-Wesley. Reading, Mass., pp 108-129

[KiKS92] Kifer M, Kim W, Sagiv Y (1992) Querying object-oriented databases. In: Stonebraker M (ed) Proc SIGMOD-1992, ACM, New York, pp 393-402

[LeH90] Lerner B, Habermann AN (1990) Beyond schema evolution to database reorganization. SIGPLAN Notices 25:67-76

[LIE87] Lieberman H (1987) Using prototypical objects to implement shared behavior in object-oriented systems. In: Meyrowitz NK (ed) Proc OOPSLA’87 Conference, ACM, New York, pp 214-223.

[MZO89] Maier D, Zhu J, Ohkawa H (1989) Features of the TEDM object model. In: Kim W, Nicolas JM, Nishio S (eds) Proc First Int’l. Conf. on Deductive and Object-Oriented Databases, Elsevier, Amsterdam, pp 511-530

[MaB90] Manola F, Buchmann A (1990) A functional/relational object-oriented model for distributed object management. GTE Labs, technical memorandum, TM-03331-11-90-165

[MoZ92] Moerkotte G, Zachmann A (1992) Multiple substitutability without affecting taxonomy. In: Pirotte A, Delobel C, Gottlob D (eds) Proc Extending Database Technology Conference 1992, Springer, Berlin, LNCS 580, pp 120-135

[MEY88] Meyer B (19989) Object-oriented software construction. Prentice-Hall, Englewood Cliffs, N.J.

[MoS93] Monk S, Sommerville I (1993) Schema evolution in OODB using class versioning. SIGMOD Record 22:16-22

[PAP91] Papazoglou MP (1991) Roles: a methodology for representing multifaceted objects. In: Karagiannis D (ed) Proc DEXA-91: Database & Expert Systems Applications Conf., Springer, Berlin, pp 7-12

[PaKB94] Papazoglou MP, Krämer BJ, Bouguettaya A (1994) On the representation of objects with polymorphic shape and behavior. In: Loucopoulos P (ed) 13th Int’l Conf. on The Entity-Relationship Approach, Springer, Berlin, LNCS 881, pp 223-240

[PKS95] Papazoglou MP, Krämer BJ, Schmidt HW (1995) Safety criteria for modeling object type transitions. Technical Report TR95-29, Dept. Software Development, Monash University, Melbourne

[PER90] Pernici B (1990) Objects with roles. In: Proc ACM Conf. on Office Information Systems, ACM, New York, pp 205–215

[RiS91] Richardson J, Schwartz P (1991) Aspects: extending objects to support multiple, independent roles. In: Clifford J, King R (eds) Proc. 1991 ACM SIGMOD Int’l. Conf. on Management of Data, ACM, New York, pp 298-307

[RUN92] Rundensteiner E (1992) MultiView: a methodology for supporting multiple views in object-oriented databases. In: Yuan LY (ed) Proc Int’l Conf. on Very Large Databases, Morgan Kaufmann, San Mateo, Calif., pp 187-198.

[SCH85] Schaffert C, et al. (1985) Trellis object-based environment: language reference manual. DEC Techn. report DEC-TR-373

[ScC95] Schmidt HW, Chen J (1995) Reasoning about concurrent objects. In: Proc. Asia-Pacific Software Engineering Conf. (APSEC ’95), IEEE, Los Alamitos, Cal., pp 86-95

[ScZ94] Schmidt HW, Zimmermann W (1994) A calculus for reasoning about complexity of object-oriented programs. J Object-Oriented Syst 2:117–147

[ScLT91] Scholl M, Laasch C, Tresch M (1991) Updatable views in object-oriented databases. In: Delobel C, Kifer M, Masunaga Y (eds) Proc 2nd DOOD Conf., Springer, Berlin, LNCS 566, pp 189-207.

[SCH92] Scholl M, et al. (1992) The COCOON object model. Technical report ETH Zürich

[ScST94] Scholl M, Schek H, Tresch M (1994) Object algebra and views for multiobject bases. In: Ozu T, Dayal U, Valduriez P (eds) Distributed Object Management, Morgan-Kaufmann, San Mateo, Calif., pp 353-374

[SkZ87] Skarra A, Zdonik S (1987) The management of changing types in an object-oriented database. In: Shriver B, Wegner P (eds) Research Directions in Object-Oriented Systems, MIT Press, Cambridge, Mass., pp 393-416

[StLU89] Stein LA, Lieberman H, Ungar D (1989) A shared view of sharing. In: Kim W, Lochovsky F (eds) Object-Oriented Concepts, Databases & Applications, Academic-Press, New York, pp 31-48

[ZdM89] Zdonik D, Maier D (eds.) Readings in Object-Oriented Database Systems, Morgan Kaufmann, San Mateo, Calif.

[ZDO90] Zdonik S (1990) Object-oriented type evolution. Advances in Database Programming Languages, ACM Press, New York

[ZIK91] Zikari R (1991) A Framework for schema updates in an object-oriented database system. In: Proc 7th Int'l Conf. on Data Engineering, IEEE, Los Alamitos, Calif., pp 2-13