Roles and dynamic subclasses: a modal logic approach[1]

Roel Wieringa Wiebren de Jonge Paul Spruit

Faculty of Mathematics and Computer Science, Vrije Universiteit
De Boelelaan 1081a, 1081 HV, Amsterdam
Email: roelw@cs.vu.nl, wiebren@cs.vu.nl, pasprui@cs.vu.nl
Tel: +31 20 548 5568, Fax: +31 20 6427705

Abstract

In this paper, we argue that object-oriented models must be able to represent three kinds of taxonomic structures: static subclasses, dynamic subclasses and role classes. If CAR is a *static subclass* of $VEHICLE$, then a vehicle that is not a car can never migrate to the CAR subclass. If EMPloyee is a *dynamic subclass* of $PERSON$, then a $PERSON$ that is not an employee may migrate to EMP. In both cases, an instance of the subclass is identical to an instance of the superclass. Finally, if EMP is modeled as a *role class* of $PERSON$ every employee differs from every person, but a $PERSON$ instance can acquire one or more EMP instances as roles. We outline an approach to formalizing these taxonomic structures in order-sorted dynamic logic with equality.

Keywords: Theoretical foundations, formal methods, OO analysis and design

1 Introduction

Class migration is the phenomenon that an object can change classes during its lifetime. The topic of class migration has received attention in database modeling at least since Bachman and Daya wrote about it in the context of the network data modeling approach in 1977 [BD77]. Recently, there has been an upsurge of interest in class migration in object-oriented database specification languages [ABGO93, Cha93, EEAK91, EK92, EJD93, JSHS91, Per90, RS91, Wie89, Wie91b, WJ91]. In this paper, we argue that there are two ways to "'migrate" between classes. Consider a situation in which a $PERSON$ becomes an EMPloyee. This can be modeled in two ways. One way is to let EMP be a *dynamic subclass* of $PERSON$. Each EMP instance is then identical to a $PERSON$ instance, and in database terms, this means that it has the same *identifier* as a that $PERSON$ instance. As a consequence, if we count five employees in a set, then this set also contains five persons.

If, on the other hand, we want to model the situation that one $PERSON$ instance can be many different EMP instances (simultaneously or in sequence), then we have to give EMP instances their own identifiers, different from each other and from $PERSON$ identifiers. In this case, we call EMP a *role class* of $PERSON$ and instances of EMP are called *roles*. Since roles have their own identifiers, if we count five employees in a set, then this set may very well contain one person (playing five EMP roles).

[1] This research is partially supported by Esprit Basic Research Action IS-CORE (working group 6071).

In this paper we make the distinction between roles and dynamic subclasses precise and give a logic for reasoning about class migration and role playing. Most approaches to class migration[ABGO93, Cha93, EEAK91, Per90, RS91, Su91] incorporate what we call dynamic subclasses but ignore the possibility of multiple role playing. In addition, logical aspects of role-playing are ignored. Jungclaus et al. [JSHS91] use a role concept similar to our dynamic subclass concept, and give a modal logic for it based on temporal logic. Gottlob et al. [GSR93] argue for a role concept like ours and show how it can be implemented in a SmallTalk-like language. However, they ignore dynamic subclasses, which we think are just as important as roles; and their paper is more implementation-oriented, where we are more oriented towards methodological and logical issues. Sciore [Sci89] defines roles in a system that combines features of inheritance by delegation [Lie86] and class-based inheritance. There are however no explicit identifiers in this approach and there is no distinction between role classes, object classes, dynamic subclasses and static subclasses.

In section 2, we discuss the methodological aspects of class-migration and role playing. We look at the connection between object classification and identification, and define static subclasses, dynamic subclasses and role classes. We discuss two inheritance mechanisms, inheritance by identity, which is appropriate for (static and dynamic) subclasses, and inheritance by delegation, which is appropriate for role classes. Section 3 gives a logic for specifying updates in general, called DDL (Dynamic Database Logic), which is a variant of dynamic logic. Section 4 applies this logic to the specification of classes, static taxonomies, dynamic taxonomies, and role playing in a way that agrees with the methodological treatment of section 2. Section 5 concludes the paper.

2 Methodological aspects of role-playing and class migration

2.1 Object classification and identification

A **class** is a set of possible individuals, called **class instances**. If the instances are objects, the class is called an **object class**, if the instances are roles, then the class is called a **role class**. What the difference between objects and roles is, is explained later. For the time being, both objects and roles can be understood to be individual objects with a local state and a local behavior.

For each class, we can distinguish three important sets.

1. The **intension** of a class is the set of *all* properties shared by all class instances. The intension of class C is written as $int(C)$.
2. The **extension** of a class is the set of all possible objects in the class; i.e. this is just the class itself. The objects in the extension are called the **instances** of the class. To emphasize that we are talking about the extension of a class C, we write $ext(C)$.
3. In any state of the world, the **existence set** of a class is the set of class instances that exist in that state. The extension of the class is always the

same set of instances, independently from the state of the world, but the existence set varies with the state of the world. The existence set of class C in state σ is written as $ext_\sigma(C)$.

In each state σ of the world, there is a set of existing objects of any type, called $Exists_\sigma$. We have for any class C that

$$ext_\sigma(C) = ext(C) \cap Exists_\sigma.$$

Class C_1 is defined to be a subclass of C_2 if $ext(C_1) \subseteq ext(C_2)$. We write this as $C_1 \xrightarrow{is\text{-}a} C_2$. If $ext(C_1) \subseteq ext(C_2)$, there is an opposite subset relationship between the *intensions* of C_1 and C_2: $int(C_2) \subseteq int(C_1)$. We have

$$ext(C_1) \subseteq ext(C_2) \Leftrightarrow int(C_2) \subseteq int(C_1).$$

For example, CAR is a subclass of $VEHICLE$ because

$$ext(CAR) \subseteq ext(VEHICLE),$$

and this is equivalent to saying that

$$int(VEHICLE) \subseteq int(CAR).$$

The containment relationship between intensions represents the *inheritance* relation from superclass to subclass.

It is now well-accepted that in object-oriented database modeling, each object should have a unique identifier [Cod79, HOT76, KC86]. We define an **identifier** for an object as a string that is used as a proper name for the object such that there is a 1-1 relationship between objects and identifiers and, once an object has an identifier, then the connection between the two is never changed. The concept of identifier is analyzed in detail in a companion paper [WJ]. Here, we want to point out that there is a close relationship between object classification and object identification. This relationship can best be explained by considering the problem of *counting* objects.

Consider the problem of counting the number of passengers that traveled in a bus in one week. If we count persons, we may count 1000, but if we count passengers, we may count 4000. The reason for this difference is that if we count things, we must identify those things, so that we can say which things are the same and which are different. But in order to identify them, we must classify them. We may count one person where we count four passengers (at different times). Similarly, we may count one building where we count three shops, we may count three employees (at the same time) where we may count one person, etc.

This relationship between classification and identification is well-known in philosophical logic [GM73, LZ87, Lee91, Str59, Wig80]. There are two views concerning the relationship between classification and identification:

1. For each class C there is an equals sign $=_C$ that says whether individuals are identical or not if they are viewed as instances of C. In this view, if p_1 is a person and p_2 is a person, we can have $p_1 =_{PERSON} p_2$ and $p_1 \neq_{PASSENGER} p_2$.
2. There is only one equals sign, that is applicable to any pair of arguments from any pair of classes. However, we may have that different instances of different classes in some sense coincide in time and space. Thus, in this view, if p is a $PERSON$ and t_1 and t_2 are $PASSENGER$s, we can have $t_1 \neq t_2$ but that both "coincide" with p.

In this paper, we have chosen the second option, where "coincidence in space and time" is formalized as dynamic subclassification and as role-playing. Thus, in the logic introduced below, we have the choice of modeling the $PASSENGER$ as a dynamic subclass of $PERSON$, in which case we have $t_1 = t_2 = p$, or as a role class of $PERSON$, in which case the player of t_1 equals the player of t_2, because both players equal p, but t_1, t_2 and p are three different identifiers.

2.2 Static partitions

We now turn to the different ways to define subclasses for a given class. We require a subclass always to be element of a (full) partition of its immediate superclass. The reason for this is explained at the end of this subsection. We say that $C_1, \ldots, C_n$ is a **static partition** of C_0 if we have

$$\bigcup_{i=1,\ldots,n} ext(C_i) = ext(C_0),$$

$$ext(C_i) \cap ext(C_j) = \emptyset \text{ for } i \neq j.$$

It follows that for existence sets we have analogous properties: in each state σ of the world we have

$$\bigcup_{i=1,\ldots,n} ext_\sigma(C_i) = ext_\sigma(C_0),$$

$$ext_\sigma(C_i) \cap ext_\sigma(C_j) = \emptyset \text{ for } i \neq j.$$

Static partitions are represented graphically as in figure 1. (We assume that we are only talking about motor vehicles in all $VEHICLE$ examples.) Each class is represented in the Coad/Yourdon way [CY90] by a rectangle containing three boxes for, from top to bottom, the class name, the attributes local to the class, and the events (i.e. messages) local to the class. To save space, most examples drop the attributes and events. Each partition $\{C_1, \ldots, C_n\}$ of C_0 is represented by a set of is_a arrows from C_i $(i = 1, \ldots, n)$ to C_0 that merge before they reach C_0. There are two hidden cardinality constraints associated with each is_a arrow, viz. that each instance of C_i is related to exactly 1 instance of C_0 and that each instance of C_0 is related to exactly one instance of exactly one C_i in the partition; this is represented by the merging of is_a arrows. Note that each is_a arrow represents the identity function $is\_a : ext(C_i) \rightarrow ext(C_0)$.

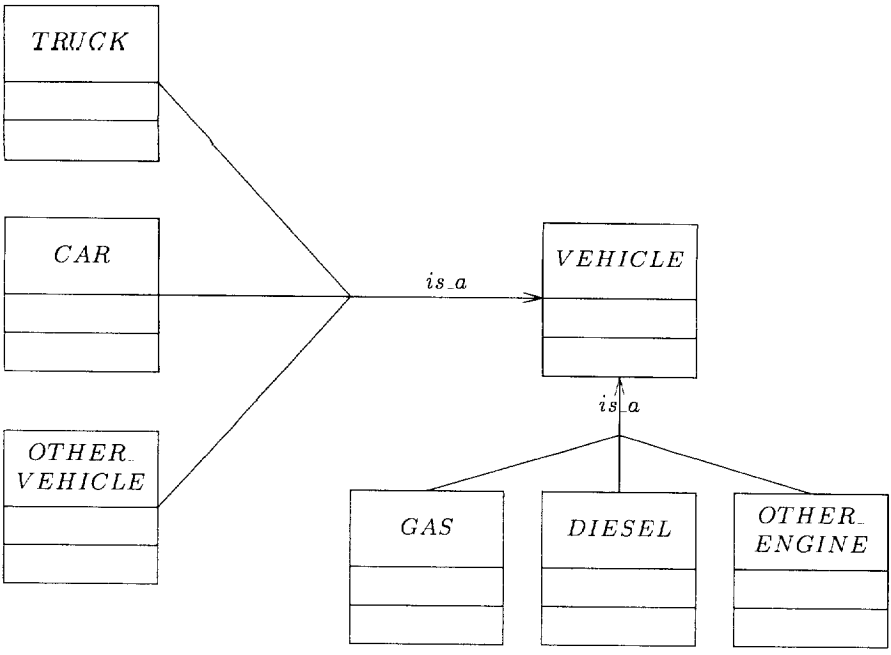

Fig. 1. Two *is_a* partitions of a class.

A partition of a class in subclasses can have any finite number of elements, as long as it contains at least two elements. We exclude the degenerate case where $n = 1$ (which would collapse into $C_1 = C_0$).

We allow any number of partitions per class. If o is an instance of C_0, it is an instance of exactly one subclass in every partition. Each pair of partitions of a class creates a number of intersection classes. For example, in figure 1 we have, among others, intersection classes

$CAR * DIESEL$,
$CAR * GAS$,
$TRUCK * DIESEL$, and
$TRUCK * GAS$.

These intersection classes are not shown in the diagram, but they are considered to be part of the model.

The number of subclasses rapidly grows bigger when we partition subclasses into smaller subclasses. All intersection classes that can be formed this way are considered to be part of the model. The number of intersection classes grows exponentially in the number of partitions per class.

A **static subclass** is a member of a static partition or a (finite) intersection of static subclasses. We can now state the reason why we require a subclass to always be an element of a partition: This way, we know that the universe of all possible objects is partitioned in a unique way into minimal static subclasses. By

taking all possible intersections of all static subclasses, we get a set of smallest static subclasses, that

- are pairwise disjoint,
- jointly cover the set of all possible objects, and
- have no static subclasses (other than the empty subclass).

We call these smallest static subclasses **species**. Species give methodological and logical advantages. First, a fundamental principle of classical taxonomies is to have, for each specialization, an unambiguous dividing principle and exhaustively list all subclasses that follow from this dividing principle, without leaving an unnamed restclass [Jos16, Res64]. This enhances the clarity of the model and reduces the chance that we miss important classes in the model.

Second, the dynamics of the model is easier to specify and understand if we specify creation events only for species and not for other classes. Suppose we subdivide a class C into subclasses but leave an unnamed restclass. We must then still be able to create instances of this unnamed restclass, and the creation event for the unnamed restclass must be declared somewhere. Now, it cannot be declared in the specification of C, because then it would be inherited by *all* subclasses of C, including all named ones. The only possibility is to declare it only for the restclass; but then this class must be specified, i.e. it must at least get a name.

2.3 Dynamic partitions

Let C_0 be a class and σ be a state of the world. We say that $C_1, \ldots, C_n$ is a **dynamic partition** of C_0 if we have for all states σ that

$$\bigcup_{i=1,\ldots,n} ext_\sigma(C_i) = ext_\sigma(C_0),$$

$$ext_\sigma(C_i) \cap ext_\sigma(C_j) = \emptyset \text{ for } i \neq j$$

and there are different states σ_1 and σ_2, for which there are i and j with $i \neq j$, such that

$$ext_{\sigma_1}(C_i) \cap ext_{\sigma_2}(C_j) \neq \emptyset.$$

The first two requirements also hold for static partitions. The third requirement means that there are at least two different states of the world such that if the world changes from one of these states into the other, at least one object moves from one subclass to another.

Note that in figure 1 we assumed that a truck cannot be converted into a car or vice versa, and that a vehicle with a gas engine cannot be rebuilt into a a vehicle with a diesel engine. The last assumption is not very realistic, and even the first one is questionable. In general, we found it very difficult to find examples of static partitions outside the realm of biology.

Just as for static partitions, we allow any finite number of elements in a dynamic partition, provided that the partition contains at least two elements. The

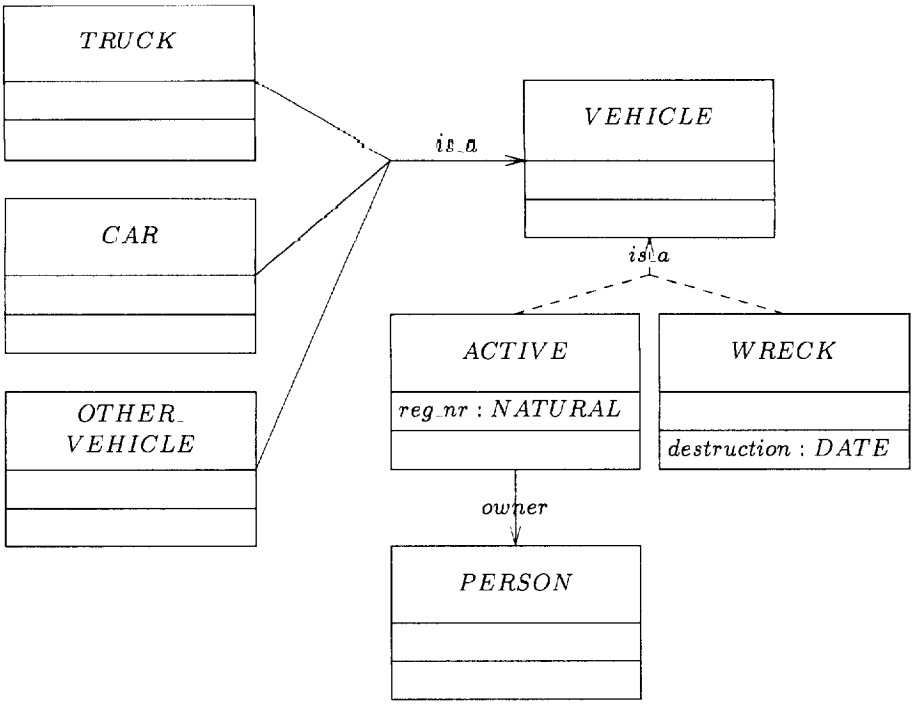

Fig. 2. Dynamic partitioning of a class.

dashed arrows in figure 2 mean that in each state of the world, the existence set of $VEHICLE$ is partitioned by the existence sets of $ACTIVE$ and $WRECK$, so that each $VEHICLE$ is either $ACTIVE$ or a $WRECK$. However, in a state transition, a $VEHICLE$ may move from one of these subclasses to the other. (If there is a constraint that it can only move from $ACTIVE$ to $WRECK$ and not from $WRECK$ to $ACTIVE$, then this must be represented in the life cycle of a $VEHICLE$ class.) Note that by inheritance, each subclass of $VEHICLE$ is also partitioned into an $ACTIVE$ and a $WRECK$ subclass. This gives us, among others, the following intersection classes:

$ACTIVE * CAR$,
$ACTIVE * TRUCK$,
$WRECK * CAR$, and
$WRECK * TRUCK$.

A **dynamic subclass** of C is an element of a dynamic partition of C or it is an intersection of a dynamic subclass of C with another class. If C_1 is a dynamic subclass of C_2, we write $C_1 \overset{is\_a}{\dashrightarrow} C_2$. All of the above intersection classes are dynamic subclasses.

We do not allow partitioning of a dynamic subclass with a *static* partition. Thus, a model in which $C_1 \overset{is\_a}{\longrightarrow} C_2 \overset{is\_a}{\dashrightarrow} C_3$ is ruled out. This considerably

simplifies the formalization as well as the intuitive structure of the models, and it seems not to exclude important modeling capabilities.

2.4 Class migration

What is often expresssed by saying that an object "migrates to a subclass" must be modeled in our approach as an object that *changes subclass* in a dynamic partition of a superclass. Figure 3 shows how we can model the situation that a person "moves to the student subclass". *NON_STUDENT*s have the same

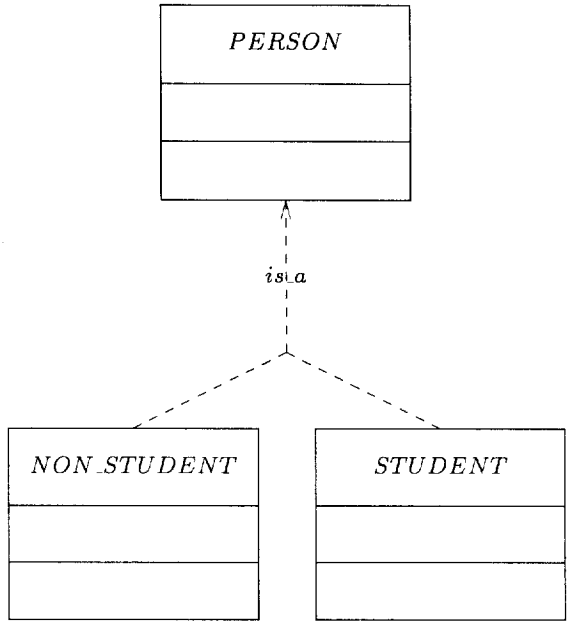

Fig. 3. A *PERSON* can move from one subclass to another in this dynamic partition.

properties as persons in general, except that *NON STUDENT*s have the additional property that they can move to *STUDENT*. (This property is not shared with *STUDENT*s.) *STUDENT*s have a number of additional properties, plus the property that they can move to *NON_STUDENT*. (This property is not shared with *NON_STUDENT*s.) Note that an object never changes its identifier when "migrating to a subclass"; this is because it must not change its identifier at all.

If dynamic subclasses would not be required to be a part of a dynamic partition, then we could have, for example, the dynamic subclass *STUDENT* as only subclass of *PERSON*. The event *become_student* occurs in the life of a *PERSON*, not of a *STUDENT*, and would therefore have to be allocated to *PERSON*. But then, intuitively, *STUDENT* would have to inherit this event,

which results in a paradox: a person in the state of being a student cannot *become* a student. This problem is avoided by requiring dynamic subclasses to be element of a dynamic partition that exhausts its immediate superclass.

From a methodological point of view, the distinguishing feature of between static and dynamic subclasses is the following property:

- A dynamic subclass can change its existence set without a change in existence set of its superclass, but when a static subclass changes its existence set, then its superclass also changes its existence set.

This is exemplified by the third property of dynamic partitions. For example, if $STUDENT \overset{is\text{-}a}{\dashrightarrow} PERSON$, it is possible that the existence set of $STUDENT$ changes but that the existence set of $PERSON$ does not change (because an existing person becomes a student or ceases to be a student). By contrast, if $CAR \overset{is\text{-}a}{\longrightarrow} VEHICLE$, then creation of a CAR is also creation of a $VEHICLE$.

2.5 Role-playing and delegation

Earlier, we promised to distinguish roles from objects. Intuitively, a **role** is just like an object, except that it has a special relationship to other objects (or roles), which are said to *play* the role. A role can be played by an object or by another role. More formally, we assume that there is a function *played by* in the model such that if R is a role classes, then there is an object- or role class such that in each state σ of the world we have

$$played\_by : ext_\sigma(R) \rightarrow ext_\sigma(P),$$

where P stands for a union of classes of player objects. For $r \in ext_\sigma(R)$, we call $played\_by(r)$ the **player** of r. This implies the following:

1. There is exactly one **player** of r.
2. r is **existence-dependent** upon its player, i.e. r cannot exist if its player does not exist.
3. There may be any number of roles played by a player, even if these roles are instances of the same role class.

It is possible to define **delegation** from roles to players. For example, suppose we model an employee e as a role of a person p, and *age* is an attribute of persons but not of employees. Then $age(e)$ would be a type error. We can recover from this error by *delegating* the evaluation of *age* to $played\_by(e)$ [Lie86]. This amounts to replacing $age(e)$ by $age(played\_by(e))$. Delegation can also be defined for events.

Role-playing is represented in a way similar to static subclasses (figure 4). Role-playing has the following characteristics:

- Each class (including a role class) can be specialized into one or more sets of role classes, called **role groups**. Each role group represents a set of mutually exclusive role classes, i.e. at any moment, any player can play roles from at most one role class in each role group simultaneously.

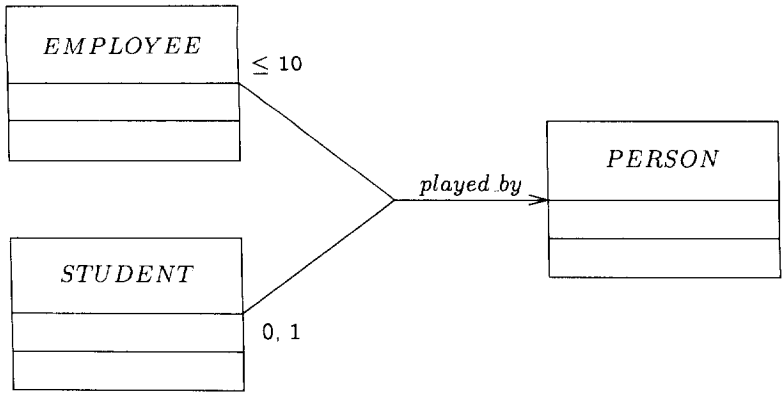

Fig. 4. Role classes with cardinality constraints. A *PERSON* can play at most 10 *EMPLOYEE* roles or, alternatively at most 1 *STUDENT* role. The two role classes are exclusive (a person cannot play roles of both classes simultaneously) but the cardinality constraints show that they are not exhaustive: Each role class allows cardinality 0, i.e. it is possible that a *PERSON* plays neither role.

- Unlike *is_a* partitions, a role group is not "exhaustive". There may be instances of the player class that do not play any role in a role group.
- For each role class, a cardinality constraint must be given, that says how many instances of the role class a player can play simultaneously. Absence of a visible constraint means unrestricted cardinality. This is different from the cardinality of an *is_a* partition, which is always 0, 1. In an *is_a* partition, each instance of the superclass is related to at most one instance of each class in the partition (and to precisely one instance of some class in its partition).
- Role groups may consist of only one role class, which is also a difference with *is_a* partitions.

To illustrate the last point, suppose we want to allow the possibility that a *PERSON* plays the roles of *EMPLOYEE* and *STUDENT* at the same time. Then we should put these in different role groups, as shown in figure 5 (both role groups have only one element in this figure).

2.6 Role playing and subclassification

We now have made two orthogonal distinctions: There are role classes and object classes, and each of these can be partitioned statically or dynamically. For example, let $EMPLOYEE \xrightarrow{played\_by} PERSON$. Then we can partition the role class as well as the object class (using an obvious notation):

$$\{MALE, FEMALE\} \xrightarrow{is\_a} PERSON$$

$$\{PERMANENT, TEMPORARY\} \xrightarrow{is\_a} EMPLOYEE.$$

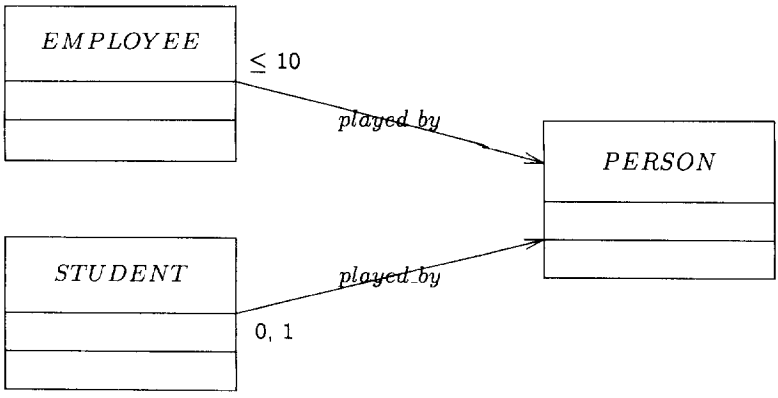

Fig. 5. A *PERSON* can play at most 10 *EMPLOYEE* roles simultaneously and at the same time at most 1 *STUDENT* role. The two role classes are *not* exclusive, because they are in different role "groups", nor are they exhaustive, for each allows cardinality 0. I.e. it is possible that a *PERSON* plays neither kind of role.

In addition, we can define a nested role for a subclass of *EMPLOYEE*:

$$PROJECT\_LEADER \overset{played\_by}{\longrightarrow} PERMANENT.$$

Thus, an employee can only be a project leader if he or she is permanent, and a permanent employee can be a project leader of any number of projects simultaneously.

The difference between role classes and (static and dynamic) subclasses lies in the fact that an instance of a subclass is identical to (i.e. has the same identifier as) an instance of its superclass but that an instance of a role class is different from any instance of its player class. This formalizes the difference with respect to counting, noted earlier.

- If C_1 *is a* C_2 (static or dynamic), then an existing instance of C_2 is related (by the identity relation) to at most one existing instance of C_1; i.e. it is identical to at most one existing instance of C_1.
- If $C_1 \overset{played\_by}{\longrightarrow} C_2$, then an existing instance of C_2 can be related to any number of existing instances of C_1 (subject to the cardinality constraint of the role class).

Consider the difference between modeling *EMPLOYEE* as a dynamic subclass and as a role class (figure 6). In both cases, a *PERSON* instance who is not an *EMPLOYEE* may become an employee. However, in the first case, an instance *e* of *EMPLOYEE* *is identical to* a *PERSON* in a certain state. A person is therefore related by the *is_a* arrow to at most one employee. In the second case, an instance *e* of *EMPLOYEE* is *different* from any *PERSON* instance. A person can be related by the *played by* arrow to any number of employees (including 0), but every existing employee is related by the *played_by* arrow to

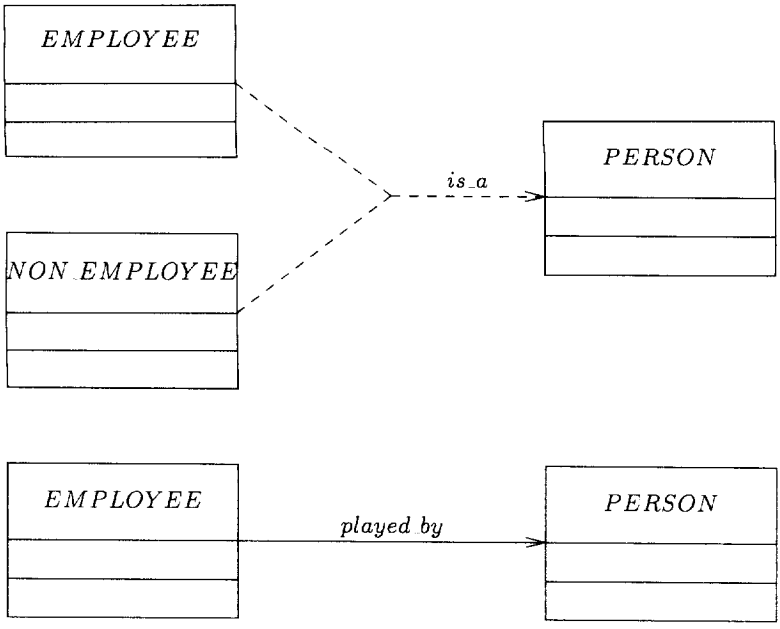

Fig. 6. In the upper diagram, $EMPLOYEE$ is a dynamic subclass of $PERSON$ and any person is at any moment either exactly one $EMPLOYEE$ or exactly one $NON\ EMPLOYEE$. In the lower diagram, $EMPLOYEE$ is a role class of $PERSON$ and any $PERSON$ can at any moment be 0, 1 or more employees.

exactly one existing person. Modeled as a role, e is a state of a person, but modeled as an instance of a dynamic subclas, it is a person in a certain state. Note that if the cardinality constraint on the *played by* arrow would be 0, 1, then one person could still play many different employee roles in sequence.

3 Order-sorted dynamic database logic

In this section, we present a version of dynamic logic [Har84, KT90] that can be used to reason about database updates. This logic is a generalization of Dynamic Database Logic (DDL) defined by Spruit [SWM92, SWM93, Spr93]. For reasons of space, we only present the syntax and axioms, and give only a brief impression of declarative semantics. Operational semantics is not discussed at all. Details will be given in the full paper.

3.1 Syntax of order-sorted logic

We start with an exposition of order-sorted logic, following the expositions by Goguen and Meseguer [GM82, GM87a, GM92]. An **order-sorted signature** $\Sigma = ((\mathbb{S}, \leq), \mathbb{F}, \mathbb{P})$ consists of the following sets:

- A partially ordered set $(\mathbb{S}, \leq)$ of **sort names**. We use w as a metavariable over $\mathbb{S}^*$ (strings of sort names) and extend $\leq$ pointwise to strings in $\mathbb{S}^*$ of equal length.
- A set $\mathbb{F}$ of **function declarations** of the form $f : w \to s$ for $w \in \mathbb{S}^*$ and $s \in \mathbb{S}$. s is called the **result sort** of f. If $w = \epsilon$, then f is called a **constant** and s is called the **sort** of f.
- A set $\mathbb{P}$ of **predicate declarations** of the form $P : w$ for $w \in \mathbb{S}^*$.

We only consider order-sorted signatures that are equational and that satisfy the covariance and regularity conditions:

- An order-sorted signature is **equational** if there is a distinguished universal sort $U \in \mathbb{S}$ such that $s \leq U$ for all $s \in U$, and there is a declaration $=: U \times U \in \mathbb{P}$. We write $x = y$ instead of $= (x, y)$.
- $\mathbb{F}$ satisfies the **covariance condition** if $f : w \to s$, $f : w' \to s'$ and $w \leq w'$, then $s \leq s'$.[2]
- $\mathbb{F}$ satisfies the **regularity condition** if for each $w \in \mathbb{S}^*$ for which there is a $(w', s') \in \mathbb{S}^*$ with $w \leq w'$ and $f : w' \to s' \in \mathbb{F}$, there is a *least arity* $(w'', s'') \in \mathbb{S}^*$ with $w \leq w''$ and $f : w'' \to s'' \in \mathbb{F}$. Regularity guarantees that each term always has a least sort.
- $\mathbb{P}$ satisfies the **regularity condition** if for each $w \in sort^*$ for which there is a $P : w' \in \mathbb{P}$ with $w \leq w'$, there is a *least arity* $w'' \in \mathbb{S}^*$ with $w \leq w''$ and $P : w'' \in \mathbb{P}$. This guarantees that different declarations of P are not mutually inconsistent.

We assume that for every sort, there is an infinite set X_s of variables and that the sets X_s are mutually disjoint for different $s \in \mathbb{S}$. In the following, we use a set $X = \{X_s \mid s \in \mathbb{S}\}$. The set $T_{\Sigma,s}(X)$ of terms of sort s over Σ and X is defined as the smallest set that satisfies the following conditions:

- $X_s \subseteq T_{\Sigma,s}(X)$.
- If $(f : s_1 \times \cdots \times s_n \to s) \in \mathbb{F}$, $t_1, \ldots, t_n$ are terms of sort $s_1, \ldots, s_n$ then $f(t_1, \ldots, t_n) \in T_{\Sigma,s'}(X)$ for all $s' \geq s$.

A term is **closed** if it does not contain variables, otherwise it is **open**. The smallest sort of a term t is called $sort(t)$. We define $sorts(t) = \{s \in \mathbb{S} \mid sort(t) \leq s\}$.

The language $L_\Sigma(X)$ of **formulas** over Σ and X is defined inductively as follows:

- If $P : s_1 \times \cdots \times s_n \in \mathbb{P}$ and $sort(t_i) \leq s_i$ for terms t_i, $i = 1, \ldots, n$, then $P(t_1, \ldots, t_n)$ is a formula.
- If ϕ and ψ are formulas, then $\phi \vee \psi$ and $\neg\phi$ are formulas.
- If ϕ is a formula and x a variable of sort s, then $\forall x : s :: \phi$ is a formula.

[2] Goguen and Meseguer call this the **monotonicity** condition. It is also possible to base order-sorted logic on Cardelli's [Car84] contravariant condition, e.g. see [EGL89].

The operators $\wedge$, $\leftrightarrow$ and $\exists$ can be introduced as usual by definition. A formula is called **closed** if all variables in it are bound by a quantifier.

For a treatment of the semantics of $L_\Sigma(X)$, the reader is refered to Goguen and Meseguer [GM87a, GM92]. In particular, it is shown there that specifications over the Horn clause fragment of $L_\Sigma(X)$ (that contains only positive conditinal formulas of the form $\phi \rightarrow \psi$ with ϕ and ψ conjunctions of positive atomic formulas) have an initial semantics.

3.2 Syntax of order-sorted dynamic database logic

We now add a requirement on signatures that will allow us to define the language of DDL. An **dynamic database logic** (DDL) signature $\Sigma_{DDL} = ((\mathbb{S}, \leq), \mathbb{F}, \mathbb{E}, \mathbb{P})$ satisfies the following requirements.

- Σ_{DDL} contains a covariant and regular order-sorted signature $\Sigma = ((\mathbb{S}, \leq), \mathbb{F}, \mathbb{P})$ with equality.
- $\mathbb{F}$ is partitioned into sets $\mathbb{F}_U$ and $\mathbb{F}_N$ of **updatable** and **nonupdatable** function symbols, and $\mathbb{P}$ is partitioned into sets $\mathbb{P}_U$ and $\mathbb{P}_N$ of **updatable** and **nonupdatable** predicate symbols, respectively.
- $=: U \times U \in \mathbb{P}_N$ (i.e. the equality sign is declared to be nonupdatable).
- There is a sort name $EVENT \in \mathbb{S}$ such that for all $s \in \mathbb{S}$ different from $EVENT$ we have $EVENT \not\leq s$ and $s \not\leq EVENT$.
- $\mathbb{E} \subseteq \mathbb{F}_\mathbb{N}$ is a set of **event declarations** of the form $e : w \rightarrow EVENT$, with $w \in \mathbb{S}^*$.
- $EVENT$ only occurs in declarations in $\mathbb{E}$, and occurs only as result sort.

Note that the equality sign and event symbols are nonupdatable. A term of sort $EVENT$ is called an **event term**. The sort $EVENT$ is reminiscent of the `msg` sort in Maude [Mes93, MQ93]. The logic of events in DOL is however diffent from the logic of event application in Maude (which is based on rewriting logic).

Other versions of DDL contain process operators to build process terms from event terms [SWM92, SWM93, Spr93]. We do not need process terms here and omit them from the language.

The language $L_{\Sigma_{DDL}}(X)$ of **formulas** over Σ_{DDL} is defined inductively as follows:

- $L_\Sigma(X) \subseteq L_{\Sigma_{DDL}}(X)$.
- If $\phi \in L_{\Sigma_{DDL}}(X)$ and e an event term, then $[e]\phi \in L_{\Sigma_{DDL}}(X)$.

$[e]\phi$ means, intuitively, that after every possible execution of e, ϕ is true. The dual $\langle e \rangle \phi$ is defined as $\neg[e]\neg\phi$ and means that there is a possible execution of e after which ϕ is true.

A **DDL specification** $Spec = (\Sigma_{DDL}, \Phi)$ is a DDL signature Σ_{DDL} and a finite set of closed equations $\Phi \subseteq L_{\Sigma_{DDL}}(X)$. An example of a specification is

sorts

$NATURAL, PERSON, FEMALE, MALE$

taxonomy
$MALE \leq PERSON$, $FEMALE \leq PERSON$
nonupdatable functions
— Declarations of operators on $NATURAL$ ...
updatable functions
$age : PERSON \rightarrow NATURAL$
nonupdatable predicates
— Declarations of Boolean operators on $NATURAL$ ...
updatable predicates
$Married : PERSON \times PERSON$
events
$inc\_age : PERSON \rightarrow EVENT$
$marry : MALE \times FEMALE \rightarrow EVENT$
axioms
$\forall m : MALE, f : FEMALE :: [marry(m, f)]Married(m, f)$
$\forall m : MALE, f : FEMALE :: \langle marry(m, f)\rangle true \rightarrow \neg Married(m, f)$
— Axioms for inc_age and for operators on $NATURAL$ ...

Comments in the above start with a —. The first axiom says that after an occurrence of event $marry(m, f)$, $Married(m, f)$ is true. The second axiom says that an execution of $marry(m, f)$ is only possible if (currently) m and f are not married.

The fragment of $L_{DDL}(X)$ that does not contain modal operators is interpreted in models which we call **possible worlds**. A formula containing modal operators is interpreted in models $\mathcal{M}$ that each consists of a set of possible worlds plus, for each closed event term e, a transition relation on possible worlds. The meaning of updatability and nonupdatability is that in each $\mathcal{M}$, the nonupdatable symbols have the same interpretation in all possible worlds, whereas the updatable symbols may have different interpretations in different possible worlds.

For example, in a model $\mathcal{M}$ of the above specification, there are sets $[\![NATURAL]\!]_{\mathcal{M}}$, $[\![MALE]\!]_{\mathcal{M}} \subseteq [\![PERSON]\!]_{\mathcal{M}}$, $[\![FEMALE]\!]_{\mathcal{M}} \subseteq [\![PERSON]\!]_{\mathcal{M}}$, and $[\![EVENT]\!]_{\mathcal{M}}$, together with functions on these sets that interpret age, inc_age, $marry$ and the operators on $NATURAL$. In each possible world of $\mathcal{M}$, $Marry$ is interpreted as a subset of $[\![MALE]\!]_{\mathcal{M}} \times [\![FEMALE]\!]_{\mathcal{M}}$. In addition, $\mathcal{M}$ contains a transition relation on possible worlds that says what the result of an event is. For example, if m and f are closed terms of sorts $MALE$ and $FEMALE$, respectively, the transition relation that interprets $marry(m, f)$ leads from worlds in which $\neg Marry(m, f)$ to worlds in which $Marry(m, f)$ is true.

The *intended* semantics of a $L_{\Sigma_{DDL}}(X)$ specification is that the nonupdatable part is interpreted as an initial algebra (this puts strong restrictions on the form of axioms for nonupdatable symbols). The nonupdatable part of a specification thus defines an abstract data type, which we then use as domain of the possible worlds of the model. All possible worlds have the same domain, but they may differ in the interpretation of the updatable symbols. There are several minimal-change semantics that can be given to events. Some of these have been

formalized for simple sets of atomic events, but further research is needed on this topic. More on the formal definition of the semantics of $L_{\Sigma_{DDL}}(X)$ appears elsewhere [SWM93, Spr93, Wie91a, WM93].

3.3 Axioms for order-sorted dynamic database logic

In the following, t stands for an arbitrary term, e stands for an arbitrary term of sort $EVENT$, $Var(t)$ is the set of variables in the term t, $sort(t)$ is the unique smallest sort of t (which we require to exist), $t[t'/x]$ is t with all occurrences of x replaced by t', $FV(\phi)$ is the set of free variables in ϕ, and $\phi[t/x]$ is ϕ with all free occurrences of x replaced by t. $D \subseteq X$ is a finite set of variable declarations. The axiom system DDL consists of the following axioms and inference rules:

(Prop) All substitution instances of propositional tautologies

(Inst) $\forall x : s :: \phi \rightarrow \phi[t/x]$ with x free for t in ϕ and $sort(t) \leq sort(x)$

(K) $[e](\phi \rightarrow \psi) \rightarrow ([e]\phi \rightarrow [e]\psi)$

(R) $$\frac{\forall D :: e_1 = e_2}{\forall D :: [e_1]\phi \leftrightarrow [e_2]\phi}$$

(Barcan) $\forall x : s :: [e]\phi \rightarrow [e]\forall x : s :: \phi$ for $x \notin Var(e)$

(PosFr) $\forall D :: P(t_1, \ldots, t_n) \rightarrow [e]P(t_1, \ldots, t_n)$ where $P \in \mathbb{P}_{\mathbb{N}}$ and t_i contains only nonupdatable function symbols $(i = 1, \ldots, n)$.

(NegFr) $\forall D :: \neg P(t_1, \ldots, t_n) \rightarrow [e]\neg P(t_1, \ldots, t_n)$ where $P \in \mathbb{P}_{\mathbb{N}}$ and t_i contains only nonupdatable function symbols $(i = 1, \ldots, n)$.

(Refl) $\forall D :: t = t$ where all variables in t are declared in D.

(Subst) $$\frac{\forall D :: t_1 = t_2}{\forall D :: t_1[t/x] = t_2[t/x]}$$ where $sort(t) \leq sort(x)$ and all variables in t_i are declared in D $(i = 1, 2)$.

(ConF) $$\frac{\forall D :: t_1 = t_2}{\forall D :: t[t_1/x] = t[t_2/x]}$$ provided that $sort(t_i) \leq sort(x)$ and all variables in t_i are declared in D $(i = 1, 2)$.

(ConP) $$\frac{\forall D :: t_1 = t_2}{P(t_1) \leftrightarrow P(t_2)}$$ provided that $P : C \in \mathbb{P}$, $sort(t_i) \leq C$ and all variables in t_i are declared in D $(i = 1, 2)$.

(Abs) $$\frac{\forall D :: t_1 = t_2}{\forall D \cup \{x : s\} :: t_1 = t_2}$$ provided that $x : s \notin D$ and all variables in t_i are declared in D $(i = 1, 2)$.

(MP) $$\frac{\phi \rightarrow \psi, \phi}{\psi}$$

(N) $$\frac{\phi}{[e]\phi}$$

(U) $$\frac{\phi}{\forall x : s :: \phi}$$

This axiom system is a part of DDL axiom systems given elsewhere [SWM93, Spr93, Wie91a, WM93]. The (K) axiom together with the (N) inference rule says that we have a normal (multi)modal system. The (R) axiom says that if event terms e_1 and e_2 are interpreted as the same element in $[\![EVENT]\!]_{\mathcal{M}}$, then they denote the same transition on possible worlds. This is an extremely important axiom to reason about events. (PosFr) and (NegFr) are positive and negative frame assumptions for nonupdatable predicates (including $=$). This tells us nothing about what frame axioms to assume for updatable symbols, but, as we will see below, we can use it to prove some interesting properties of specifications. The Barcan formula enforces a constant domain in all possible worlds. (Refl) axiomatizes the reflexiveness property of equality. (Transitivity and symmetry of equality is derivable from the system.) (Subst) formalizes substitution of equals for equals, and (ConF) and (ConP) require all function, attribute and predicate symbols to behave like a congruence with respect to equality. The explicit quantification in front of the equations is needed to avoid problems with empty sorts; completeness of the equational fragment of the logic in turn requires the (Abs) inference rule [GM82].

4 Using DDL as a logic for objects, class migration and role-playing

In this section we use DDL as a logic for reasoning about objects. To do this we impose a few restrictions on signatures, and thus on the generated language. The resulting language is called Dynamic Object Language (DOL). The logic of DOL specifications is the same as that of DDL specifications.

4.1 Class specification

An **object signature** $\Sigma_{DOL} = ((\mathbb{S}, \leq), \mathbb{C}, \mathbb{F}, \mathbb{E}, \mathbb{A}, \mathbb{P}, \mathbb{B})$ consists of the following sets:

- $((\mathbb{S}, \leq), \mathbb{F}, \mathbb{E}, \mathbb{P})$ is a DDL signature.
- $\mathbb{C} \subseteq \mathbb{S}$ is a set of **class names** such that if $C \in \mathbb{C}$, then all sort names compatible with C by $\leq$ are in $\mathbb{C}$.
- $\mathbb{A} \subseteq \mathbb{F}$ is a set of **attribute** declarations, which all have the form $a : C \rightarrow s$ for $C \in \mathbb{C}$ and $s \in \mathbb{S}$. All updatable function symbols are attribute symbols.
- $\mathbb{B} \subseteq \mathbb{P}$ is a set of predicate declarations of the form $P : C$ for $C \in \mathbb{C}$. All updatable predicate symbols are in $\mathbb{B}$.
- There is a universal class $C_U \in \mathbb{C}$ such that $C \leq C_U$ for all $C \in \mathbb{C}$.
- There is an updatable predicate symbol $Exists : C_U \in \mathbb{B}$.

Thus, an object signature is just a DDL signature with some distinguished sets of declarations and an existence predicate and with all updatable symbols unary.

A **class specification** over Σ_{DOL} is a pair (Σ_{DOL}, E), where E is a finite set of formulas over Σ_{DOL}. In the following example class specification, the specifications of $NATURAL$ and $PERSON$ (a class) are not shown.

classes

$VEHICLE$

functions

$v_0 : VEHICLE$

$next : VEHICLE \rightarrow VEHICLE$

attributes

$weight : VEHICLE \rightarrow NATURAL$

$owner : VEHICLE \rightarrow PERSON$

$price : VEHICLE \rightarrow NATURAL$

$registration\_nr : VEHICLE \rightarrow NATURAL$

updatable predicates

$Wreck : VEHICLE$

events

$change\_owner : VEHICLE \times PERSON \times NATURAL \rightarrow EVENT$

$change\ weight : VEHICLE \times NATURAL \rightarrow EVENT$

axioms

— 1. Static integrity constraint.

$\forall v : VEHICLE :: weight(v) < 10000$

— 2. Static integrity constraint.

$\forall v_1, v_2 : VEHICLE :: v_1 = v_2 \leftrightarrow$
$registration\_nr(v_1) = registration\ nr(v_2)$

— 3. Axiom defining the effect of *change_owner*.

$\forall v : VEHICLE, p : PERSON, m : NATURAL ::$
$[change\_owner(v, p, m)]owner(v) = p \wedge price(v) = m$

— 4. Necessary precondition for success of *change owner*.

— A vehicle can only be sold when it is not a $Wreck$.

$\forall v : VEHICLE, p : PERSON, m : NATURAL ::$
$\langle change\_owner(v, p, m)\rangle true \rightarrow \neg Wreck(v)$

— 5. Effect axiom for *change_weight*.

$\forall v : VEHICLE, n : NATURAL :: [change\ weight(v, n)]weight(v) = n$

The function declarations for the $VEHICLE$ class define infinitely many closed terms of sort $VEHICLE$ of form $next^n(v_0)$ for $n \geq 0$. We view these as formal counterparts of internal (system-generated) $VEHICLE$ identifiers. The attributes and predicates hold the state of $VEHICLE$s and the events define the local state changes of a $VEHICLE$ instances. Axioms using updatable symbols but containing no modalities are called **static integrity constraints**. By axiom 2, the value of $registration\ nr(v)$ can be used as external (visible to the user) identifier of v.

The proof system of DDL can be used to prove properties of specifications. For example, we can derive a precondition for the application of *change_weight*. The bracketed numbers in the following refer to axioms in the vehicle specification, the unbracketed numbers refer to lines in the proof.

1 $\forall v : VEHICLE, n : NATURAL ::$	
$[change\_weight(v, n)]weight(v) = n$	(5)
2 $\forall v : VEHICLE :: weight(v) < 10000$	(1)
3 $weight(v) < 10000$	2, (Inst)
4 $\forall v : VEHICLE, n : NATURAL ::$	
$[change\_weight(v, n)]weight(v) < 10000$	(U), (N), 3
5 $\forall v : VEHICLE, n : NATURAL ::$	
$[change\_weight(v, n)]n < 10000$	1, 4, (ConF)
6 $\forall v : VEHICLE, n : NATURAL ::$	
$\langle change\_weight(v, n)\rangle true \rightarrow \langle change\_weight(v, n)\rangle n < 10000$	5
7 $\forall v : VEHICLE, n : NATURAL ::$	
$\neg n < 10000 \rightarrow [change\_weight(v, n)]\neg n < 10000$	(NegFr)
8 $\forall v : VEHICLE, n : NATURAL ::$	
$\langle change\_weight(v, n)\rangle n < 10000 \rightarrow n < 10000$	7
9 $\forall v : VEHICLE, n : NATURAL ::$	
$\langle change\_weight(v, n)\rangle true \rightarrow n < 10000$	6, 8

4.2 Specifying static partitions

We saw in the vehicle example that the extension of a class is just the set of internal identifiers generated for the class, and that we generate these identifiers by declaring a constant v_0, which we will call a **seed**, and a function *next*, which we will call a **generator**. This idea is generalized to all static partitions: We specify a seed and a generator for every species and we do not specify a seed or generator for any other class. Note that the requirement that we have species makes the implementation of this idea simpler than it would otherwise have been.

To implement the above idea, we must explicitly declare all species, and, in general, this can require the declaration of a large number of intersection classes. To avoid this, we drop all seeds and generators from a specification and add some syntactic sugar to indicate what the static partitions in a specification are:

classes

$CAR, TRUCK, OTHER\_VEHICLE$ **static partition of** $VEHICLE$
$GAS, DIESEL, OTHER\_ENGINE$ **static partition of** $VEHICLE$

This suffices to declare the static taxonomic structure in the intended semantics of the specification. That is, in the initial semantics of the abstract data type defined by the nonupdatable part of the specification, a static taxonomic structure is defined by placing a seed and a generator in every species. The details of this are straightforward but tedious and we omit them.

4.3 Specifying dynamic partitions

A **dynamic object signature** Σ_{DOL} is a tuple $((\mathbb{S}, \leq), \mathbb{C}_S, \mathbb{C}_D, \mathbb{F}, \mathbb{E}, \mathbb{A}, \mathbb{P}, \mathbb{B})$ that satisfies the following requirements:

- $((\mathbb{S}, \leq), \mathbb{C}_S \cup \mathbb{C}_D, \mathbb{F}, \mathbb{E}, \mathbb{A}, \mathbb{P}, \mathbb{B})$ is an object signature.
- $\mathbb{C}_S$ and $\mathbb{C}_D$ are disjoint. Their elements are called **static** and **dynamic** class symbols, respectively.
- If $C' \in \mathbb{C}_D$ and $C \leq C'$, then $C \in \mathbb{C}_D$.
- For each $C \in \mathbb{C}_D$, the set $\{C' \in \mathbb{C}_S | C \leq C'\}$ is non-empty and has a unique smallest element, which we call the **natural kind** of C, written as $C' = nk(C)$.
- For each $C \in \mathbb{C}_D$, there is an updatable function symbol $r_{nk(C)\to C} : nk(C) \to C$, called a **retract** for C.
- For each $C \in \mathbb{C}_D$, there is a **class predicate** $P_C : C \in \mathbb{B}$.

Thus, classes are partitioned into static and dynamic classes. All subclasses of a dynamic class are dynamic, and each dynamic class has a unique least upper bound in the set of static classes. Finally, each dynamic class has a retract and an updatable class predicate. The use of retracts and class predicates is illustrated in a moment. Retracts have been introduced by Goguen, Jouannaud and Meseguer [GJM85]. Class predicates are used in FOOPS, OBJ and EQLOG [GM86, GM87b]. Retracts have not been used before to define dynamic subclasses.

A **dynamic subclass specification** is a class specification over $L(\Sigma_{DOL})$ containing for each retract $r_{C'\to C} : C' \to C$, an axiom

$$\forall x : C' :: \phi \leftrightarrow r_{C'\to C}(x) = x.$$

The formula ϕ says exactly when an object is an instance of the dynamic subclass C. The intended semantics of a dynamic subclass specification is that seeds and generators are only defined for static subclasses and not for dynamic subclasses. The use of retracts and class predicates is illustrated in the following example.

static classes
$VEHICLE$
dynamic classes
$ACTIVE, WRECK$
taxonomy
$ACTIVE \leq VEHICLE$
$WRECK \leq VEHICLE$
updatable functions
$r_{VEHICLE\to ACTIVE} : VEHICLE \to ACTIVE$
$r_{VEHICLE\to WRECK} : VEHICLE \to WRECK$
updatable predicates
$Active : VEHICLE$
$Wreck : VEHICLE$
events
$wreck\_the\_car : VEHICLE \to EVENT$
$create : VEHICLE \to EVENT$
axioms
-- 1, 2. *Active* and *Wreck* partition the states of a $VEHICLE$.

$\forall x : VEHICLE :: \neg(Active(x) \wedge Wreck(x))$
$\forall x : VEHICLE :: Active(x) \vee Wreck(x)$

— 3. All *Active* vehicles have sort $ACTIVE$.
$\forall x : VEHICLE :: Active(x) \leftrightarrow r_{VEHICLE \to ACTIVE}(x) = x$

— 4. All *Wrecked* vehicles have sort $WRECK$.
$\forall x : VEHICLE :: Wreck(x) \leftrightarrow r_{VEHICLE \to WRECK}(x) = x$

— 5. All $ACTIVE$ vehicles are *Active*.
$\forall x : ACTIVE :: Active(x)$

— 6. All $WRECK$s are *Wrecked*.
$\forall x : WRECK :: Wreck(x)$

— 7. Object creation
$\forall x : VEHICLE :: [create(x)]Exists(x) \wedge Active(x)$

— 8. Class migration.
$\forall x : ACTIVE :: [wreck\ the\_car(x)]Wreck(x)$

$VEHICLE$ is the natural kind of $ACTIVE$ and $WRECK$. Note that we may very well have static subclasses like CAR and $TRUCK$ of $VEHICLE$. Thus, the natural kind of a dynamic class need not itself be a smallest class in the set of static classes.

Axioms 1 and 2 are called the **partition axioms** for the *Active*, *Wreck* partition.

The **retract axiom** (3) states that the retract of $ACTIVE$ is the identity function precisely when the sort predicate *Active* is true. Axiom (4) says something analogous for $WRECK$. These axioms show that there is a redundancy between class predicates and retracts. We still introduce class predicates as well as the retract, because both are convenient at different places for reasoning about dynamic subclasses.

Axioms (5, 6) are called **class predicate axioms**. They say that *Active* is a necessary condition for being an instance of $ACTIVE$ and that *Wreck* is a necessary condition for being a $WRECK$. This is the converse of the retract axioms (3, 4), which say that if for a $VEHICLE$ x we have $Active(x)$, then x is of type $ACTIVE$. Together, these axioms provide the connection we want between dynamic subclasses and class predicates.[3]

Axioms 1, 7 and modal reasoning allow us to prove from the specification that

$$\forall x : VEHICLE :: [create(x)]Exists(x) \wedge Active(x) \wedge \neg Wreck(x).$$

We simplify the examples by adding some syntactic sugar in the form of a declaration of a **dynamic subclass partition** for each dynamic partition, that summarize the declaration of class predicates, conditional retracts, and their axioms. Using this syntactic sugar, the above example can be abbreviated to the following:

[3] Dynamic subclasses could also have been defined by means of sort constraints [GJM85, MG93]. We use retracts, because the logic of sort constraints is not yet clear to us.

classes
$ACTIVE, WRECK$ **dynamic partition of** $VEHICLE$
events
$wreck\_the\_car : VEHICLE \rightarrow EVENT$
$create : VEHICLE \rightarrow EVENT$
axioms
$\forall v : VEHICLE :: [create(v)]Exists(v) \wedge Active(v)$
$\forall v : ACTIVE :: [wreck\_the\_car(x)]Wreck(x)$

Note that it is useful to have both sort names for dynamic classes as well as class predicates. The sort names allow us to declare predicates and attributes that are applicable to dynamic subclasses only; the class predicates allow us to specify class migration easily.

4.4 Specifying roles

A **role signature** Σ_{DOL} is a tuple $((\mathbb{S}, \leq), \mathbb{C}_S, \mathbb{C}_D, \mathbb{R}, \mathbb{F}, \mathbb{E}, \mathbb{A}, \mathbb{P}, \mathbb{B})$ that satisfies the following requirements.

- $((\mathbb{S}, \leq), \mathbb{C}_S \cup \mathbb{R}, \mathbb{C}_D, \mathbb{F}, \mathbb{E}, \mathbb{A}, \mathbb{P}, \mathbb{B})$ is a dynamic object signature.
- $\mathbb{R}$ is disjoint from $\mathbb{C}_S$.
- For every $R \in \mathbb{R}$ there is exactly one $C \in \mathbb{C}_S \cup \mathbb{C}_D \cup \mathbb{R}$ such that there is a declaration $played\ by : R \rightarrow C \in \mathbb{A}$.
- $\mathbb{A}$ contains no loop of declarations $played\_by : C_0 \rightarrow C_1$, ..., $played\_by : C_n \rightarrow C_0$.
- $\leq$ and $played\ by$ are disjoint, i.e. at most one of $s_1 \leq s_2$ and $s_1 \xrightarrow{played\ by} s_2$ can be the case.

Thus, roles can have static and dynamic partitions and they can be played by an instance of a static or dynamic subclass, as well as by a role. Note that we require there to be exactly one player class for each role class. This player class may however have subclasses. An example role specification follows:

static classes
$PERSON$
role classes
$STUDENT, EMP$
functions
$s_0 : STUDENT$
$e_0 : EMP$
$next : STUDENT \rightarrow STUDENT$
$next : EMP \rightarrow EMP$
attributes
$played\ by : STUDENT \rightarrow PERSON$
$played\ by : EMP \rightarrow PERSON$
events
$become\ student : PERSON \times STUDENT \rightarrow EVENT$

axioms

— 1. Define $STUDENT$, EMP as role group of $PERSON$.

$\forall s : STUDENT, e : EMP :: \neg played\_by(s) = played\_by(e)$

— 2, 3. Existence of players required.

$\forall s : STUDENT :: Exists(s) \rightarrow Exists(played\_by(s))$

$\forall e : EMP :: Exists(e) \rightarrow Exists(played\_by(e))$

— 4, 5. *become student* is a role creation event.

$\forall p : PERSON, s : STUDENT :: \langle become\_student(p, s)\rangle true \rightarrow \neg Exists(s)$

$\forall p : PERSON, s : STUDENT ::$

$[become\_student(p, s)]Exists(s) \wedge played\_by(s) = p$

Axiom 1 is called a **role group axiom** and axioms 2 and 3 **player existence axioms.**

It is easy to prove that

$$\forall p : PERSON, s : STUDENT :: [become\_student(p, s)]Exists(p) \wedge Exists(s)$$

and

$$\forall p : PERSON, s : STUDENT, e : EMP ::$$

$$[become\ student(p, s)]played\ by(e) \neq played\_by(s).$$

More interesting would be, however, derivation a proof of

$$\forall p : PERSON, s : STUDENT ::$$

$$\langle become\ student(p, s)\rangle true \rightarrow Exists(p) \wedge \neg Exists(s),$$

for this would give us a necessary precondition for the occurrence of *become student*(p, s). However, such a derivation would require a frame assumption for updatable predicate symbols, which says that $Exists(p)$, if true after *become student*(p, s), also was true before this event. Such a frame assumption is easy to find in this object-oriented context: every event can be required, by putting syntactic restrictions on the axioms, to have only *local* effects. Since p is different from s, an event local to s will not affect the $Exists(p)$. In this view, *become student*(p, s) is local to s, and a name like *create student*(s, p) would therefore be more intuitive. Space limitations prevent us from working this out.

Another addition is to ensure that upon creation, a role has a fresh identifier, never used before. This can be easily done by introducing a predicate $Used$ that every state of the world has as extension the set of all instances of $STUDENT$ and EMP that have been created (added to the existence set). It is a trivial matter to add the appropriate axioms to the specification, and we omit it here.

A third addition omitted here is the specification of cardinality constraints. In general, we may want to specify constraints such as each person can only play at most one $STUDENT$ role at the same time. Cardinality constraints can be specified by defining an inverse of the *played by* attribute, say *inv played by* : $PERSON \rightarrow STUDENTS$, where $STUDENTS$ is the type of finite sets

of $STUDENT$ instances. We can then specify a cardinality constraint on the $STUDENT$ role by an axiom like

$$\forall p : PERSON :: card(inv\_played\_by(p)) \leq 1.$$

Definition of the set type $STUDENTS$ and of the inverse attribute inv_played_by is straightforward and is omitted here.

We can again define some syntactic sugar, in the form of a **role group** declaration, to abbreviate the specification:

classes
$STUDENT, EMP$ **role group of** $PERSON$
events
$become\_student : PERSON \times STUDENT \rightarrow EVENT$
axioms
$\forall p : PERSON, s : STUDENT, e : EMP :: \langle become\_student(p, s)\rangle true \rightarrow$
$\neg Exists(s) \wedge Exists(p) \wedge \neg played\_by(e) = p$
$\forall p : PERSON, s : STUDENT ::$
$[become\ student(p, s)]Exists(s) \wedge played\ by(s) = p$

The **role group of** section abbreviates the part of the **functions** section that determines the placement of seeds and generators for the role classes in the same way as is done for the static partitions. In addition, it is shorthand for the role group and player existence axioms.

In the syntax introduced so far, an attribute application like $age(s)$ for s of type $STUDENT$ is a type error. This can be circumvented by having the parser of an application $a(t)$ resolve any type mismatches between the expected argument sort of a and the sort of t by the insertion of *played by* functions. Thus, if $a : C_1 \rightarrow C_2$ and C_1 is not among the sorts of t, then the parser would replace $a(t)$ by $a(played\_by(t))$ and see if the mismatch still exists. If the mismatch still exists and $sort(played\_by(t))$ is a role class, then this can be repeated. Because there are no loops in *played by* declarations and the $\leq$ and *played by* relations are disjoint, the *played by* graph does not contain cycles, this replacement process will terminate. The term $a(s)$ will now be replaced, by the parser, by $age(played\ by(s))$ and this term can be parsed correctly. This replacement process is a compile-time version of **delegation**: s delegates the answering of the *age* message to its player.

Note that in DOL, we allow only one player class of a role class. This means that if we want to let objects of class $C_1, \ldots, C_n$ be possible players of roles of class R, then we should define a supertype $C \geq C_i$ $(i = 1, \ldots, n)$ and define C to be the player class of R.

5 Conclusions

We have shown that there is a clear difference between static and dynamic subclasses, and, independently from that, between object classes and role classes.

We have also shown how these taxonomic structures, can be specified in an order-sorted logic and how we can specify class migration and role playing in a subset of Dynamic Database Logic. For each of the taxonomic constructs, we gave methodological principles which can be used to discover which kind of class is being modeled. These principles are language-independent. DOL resembles Modal Action Logic [GMS83, JKM86, KMS86, RFM91], but differs from it in that DOL is integrated with equational ADT specification and is used to specify objects rather than databases. Conditional retracts and class predicates are used to specify class migration, and a very simple delegation mechanism is defined for roles. These constructs are not completely language-independent, but they are presented in such a way that they may feasibly be combined with particular logical object specification languages, such as LCM [FW93], Troll [JSHS91] and Oblog [CSS89].

Acknowledgements: This paper profited from discussion with Jacques van Leeuwen and from comments given on an earlier versions by Remco Feenstra, Wiebe van der Hoek and Maarten de Rijke. Thanks are due to the anonymous referees for their constructive comments, which led to the elimination of some errors.

References

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings of the 18th International Conference on Very Large Databases*, pages 39–51, Dublin, Ireland, August 24–27 1993.

[BD77] C.W. Bachman and M. Daya. The role concept in data models. In *Proceedings of the Third International Conference on Very Large Databases*, pages 464–476, 1977.

[Car84] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Datatypes*, pages 51–67. Springer, 1984. Lecture Notes in Computer Science 173.

[Cha93] G. Chambers. Predicate classes. In *European Conference on Object-Oriented Programming (ECOOP93)*, pages 268–296. Springer, 1993.

[Cod79] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4:397–434, 1979.

[CSS89] J.F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 User's Manual, version 2.3.* Instituto Superior Técnico, Lisbon, May 1989.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press/Prentice-Hall, 1990.

[EEAK91] R. Elmasri, I. El-Assal, and V. Kouramajian. Semantics of temporal data in an extended er model. In H. Kangalasso, editor, *Entity-Relationship Approach: The Core of Conceptual Modelling*, pages 239–154. Elsevier, 1991.

[EGL89] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* B.G. Teubner, 1989.

[EJD93] H.-D. Ehrich, R. Jungclaus, and G. Denker. Object roles and phases. In U.W. Lipeck and G. Koschorreck, editors, *Proceedings of the Interna-*

tional Workshop on Information Systems – Correctness and Reusability (ISCORE'93), pages 114–121. Institut für Informatik, Universität Hannover, Postfach 6009, Hannover, Germany, 1993.

[EK92] R. Elmasri and V. Kouramajian. A temporal query language based on conceptual entities and roles. In G. Pernul and A.M. Tjoa, editors, *Entity-Relationship Approach –ER'92*, pages 375–388. Springer, 1992. Lecture Notes in Computer Science 645.

[FW93] R.B. Feenstra and R.J. Wieringa. LCM 3.0: a language for describing conceptual models. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1993.

[GJM85] Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Operational semantics of order-sorted algebra. In W. Brauer, editor, *Proceedings, 1985 International Conference on Automata, Languages and Programming*, pages 221–231. Springer, 1985. Lecture Notes in Computer Science, Volume 194.

[GM73] D. Gabbay and J.M. Moravcsik. Sameness and individuation. *The Journal of Philosophy*, 70:513–526, 1973.

[GM82] J.A. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notices*, 17(1):9–17, 1982.

[GM86] J.A. Goguen and J. Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, 1986.

[GM87a] J.A. Goguen and J. Meseguer. Models and equality for logical programming. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'87) Volume 2*, pages 1–22. Springer, 1987. Lecture Notes in Computer Science 250.

[GM87b] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[GM92] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[GMS83] F. Golshani, T.S.E. Maibaum, and M.R. Sadler. A modal system of algebras for database specification and query/update support. In *Proceedings of the Nineth International Conference on Very Large Databases*, pages 331–359, 1983.

[GSR93] G. Gottlob, M. Schreffl, and B. Röck. Extending object-oriented systems with roles. Manuscript, 1993.

[Har84] D. Harel. Dynamic logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic II*, pages 497–604. Reidel, 1984.

[HOT76] P. Hall, J. Owlett, and S. Todd. Relations and entities. In G.M. Nijssen, editor, *Modelling in Database Management Systems*, pages 201–220. North-Holland, 1976.

[JKM86] P. Jeremaes, S. Khosla, and T.S.E. Maibaum. A modal (action) logic for requirements specification. In D. Barnes and P. Brown, editors, *Software Engineering 86*, pages 278–294. Peter Peregrinus Ltd., 1986.

[Jos16] H.W.B. Joseph. *An Introduction to Logic.* Clarendon Press, 1916.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The TROLL language. Technical report, Abt. Datenbanken, Tech. Universität Braunschweig, P.B. 3329, Braunschweig, Germany, 1991.

[KC86] S.N. Khoshafian and G.P. Copeland. Object identity. In *Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, 1986. SIGPLAN Notices 22 (12).

[KMS86] S. Khosla, T.S.E. Maibaum, and M. Sadler. Database specification. In T.B. Jr. Steel and R. Meersman, editors, *Database Semantics (DS-1)*, pages 141–158. North-Holland, 1986.

[KT90] D. Kozen and J. Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 789–840. Elsevier Science Publishers, 1990.

[Lee91] Jacques van Leeuwen. *Individuals and Sortal Concepts: An Essay in Logical Descriptive Metaphysics.* PhD thesis, University of Amsterdam, 1991.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications*, pages 214–223, October 1986.

[LZ87] Jacques van Leeuwen and Henk Zeevat. Identity and common nouns in intensional logic. In J. Groenendijk, M. Stokhof, and F. Veltman, editors, *Proceedings of the Sixth Amsterdam Colloquium*, pages 219–241, Amsterdam, 1987. Institute for Language, Logic and Information.

[Mes93] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O.M. Nierstrasz, editor, *European Conference on Object-Oriented Programming (ECOOP'93)*, pages 220–246, Kaiserslautern, July 1993. Springer. Lecture Notes in Computer Science 707.

[MG93] J. Meseguer and J.A. Goguen. Order-sorted algebra solves the constructor-selector, multiple representation, and coercion problems. *Information and Computation*, 103:114–158, 1993.

[MQ93] J. Meseguer and X. Qian. A logical semantics for object-oriented databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data*, pages 89–98. ACM Press, May 26–28 1993. Sigmod Record 22(2), June 1993.

[Per90] B. Pernici. Objects with roles. In *IEEE/ACM Conference on Office Information Systems*, Cambridge, Mass., 1990.

[Res64] N. Rescher. *Introduction to Logic.* St. Martin's Press, 1964.

[RFM91] M. Ryan, J. Fiadeiro, and T. Maibaum. Sharing actions and attributes in modal action logic. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 569–593. Springer, 1991. Lecture Notes in Computer Science 526.

[RS91] J. Richardson and P. Schwartz. Aspects: Extending objects to support multiple, independent roles. In *ACM-SIGMOD International Conference on Management of Data*, pages 298–307, Denver, Colorado, May 1991. ACM. Sigmod Record, Vol. 20.

[Sci89] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, 7(2):103–122, 1989.

[Spr93] P.A. Spruit. Function symbols in dynamic database logic. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[Str59] P. Strawson. *Individuals.* Methuen, 1959.

[Su91] J. Su. Dynamic constraints and object migration. In G.M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 233–242, Barcelona, Spain, September 3-6 1991.

[SWM92] P.A. Spruit, R.J. Wieringa, and J.-J.Ch. Meyer. Axiomatization, declarative semantics and operational semantics of passive and active updates in logic databases. Technical Report IR-294, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, June 1992. To be published, *Journal of Logic and Computation.*

[SWM93] P.A. Spruit, R.J. Wieringa, and J.-J.Ch. Meyer. Dynamic database logic: The first-order case. In U.W. Lipeck and B. Thalheim, editors, *Modelling Database Dynamics*, pages 103–120. Springer, 1993.

[Wie89] R.J. Wieringa. Role change in database domains. Technical Report IR-180, Faculty of Mathematics and Computer Science, Free University, Amsterdam, 1989.

[Wie91a] R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566.

[Wie91b] R.J. Wieringa. Steps towards a method for the formal modeling of dynamic objects. *Data and Knowledge Engineering*, 6:509–540, 1991.

[Wig80] D. Wiggins. *Sameness and Substance.* Basil Blackwell, 1980.

[WJ] R.J. Wieringa and W. de Jonge. Object identifiers, keys, and surrogates. In preparation.

[WJ91] R.J. Wieringa and W. de Jonge. The identification of objects and roles. Technical Report IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1991.

[WM93] R.J. Wieringa and J.-J.Ch. Meyer. Actors, actions, and initiative in normative system specification. *Annals of Mathematics and Artificial Intelligence*, 7:289–346, 1993.