

Ownership as a conceptual modeling construct

Michael Halper ^{a,*}, Li-min Liu ^b, James Geller ^c, Yehoshua Perl ^c

^a *Department of Mathematics and Computer Science, Kean University, 1000 Morris Avenue, Union, NJ 07083-0411, USA*

^b *Department of Applied Mathematics, Chung Yuan Christian University, Chung-Li, Taiwan*

^c *CS Department, NJIT, Newark, NJ 07102, USA*

Received 4 January 2005; accepted 27 July 2006

Available online 18 September 2006

Abstract

Ownership is a relationship that pervades many aspects of our lives, from the personal to the economic, and is particularly important in the realm of the emerging electronic economy. As it is understood on an intuitive level, ownership exhibits a great deal of complexity and carries a rich semantics with respect both to the owner and the possession. A formal model of an ownership relationship that inherently captures varied ownership semantics is presented. This ownership relationship expands the repertoire of available conceptual data modeling primitives. It is built up from a set of characteristic dimensions, namely, exclusiveness, dependency, documentation, transferability, and inheritance, each of which focuses on a specific aspect of ownership semantics. The data modeler has the ability to make a variety of choices along these five dimensions, and thus has access to a wide range of available ownership features in a declarative fashion. These choices ultimately impose various constraints (specified in OCL) on the states of data objects and their respective ownership activities, including transactions such as acquiring and relinquishing ownership. To complement the formal aspects of the ownership model and enhance its usability, we present a graphical ownership notation that augments the Unified Modeling Language (UML) class diagram formalism. An implementation of the ownership relationship in a commercial object-oriented database system is discussed.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Ownership relationship; Semantic relationship; Ownership semantics; Conceptual modeling; Object-oriented modeling; Object-oriented database system; Electronic commerce

1. Introduction

The inclusion of the semantics of relationships in knowledge and information systems has played a crucial role in improving their ability to model the world. Any relationship representation that possesses inherent constraints and functionalities that mirror human understanding and conventions is bound to be more useful than a representation where a relationship is just a meaningless label [12,13,68]. Fittingly, such relationships are

* Corresponding author. Tel.: +1 908 737 3792.

E-mail addresses: mhalper@kean.edu (M. Halper), limin@cycu.edu.tw (L.-m. Liu), geller@oak.njit.edu (J. Geller), perl@oak.njit.edu (Y. Perl).

called *semantic relationships*. Indeed, their importance derives in no small measure from the fact that they capture aspects of our everyday discourse and our perception of real-world systems and enterprises.

While the semantic relationships IS-A and part-whole have received widespread coverage in the data modeling community (see Section 2), it is surprising that ownership—a very important element of human discourse and economic activity—has been given scant attention. Such a construct is well deserving of a formal treatment and inclusion in data modeling methodologies. Indeed, the proper maintenance of ownership semantics, including all the subtleties of its surrounding activities such as buying and selling, is a critical issue, particularly with the emergence of E-Commerce [41] over the last decade. In order to build correct database implementations involving the vast range of ownable objects from real estate and equities to trademarks, copyrights, and patents, a sound theoretical basis in the form of an ownership model is necessary. Consider, for example, a business that is owned by several partners with defined percentages. The constraint that the total of all these percentages remain at 100% as new partners are added must be enforced. Another such constraint is non-transferability, a characteristic exhibited, e.g., by airline tickets. Any attempt to transfer the ownership of a given airline ticket from one person to another should automatically be disallowed.

In this paper, we present a semantic relationship, called the *ownership relationship*, that extends the available repertoire of conceptual data modeling primitives. It provides the data modeler with access to an extensive range of ownership semantics in a declarative manner. The ownership relationship imposes a wide variety of constraints and restrictions on ownership transactions, such as those mentioned above, that must be satisfied automatically by a system (e.g., DBMS) that is made “ownership aware.” In this way, the ownership relationship allows applications to mirror the behavior expected of real-world ownership activities. We emphasize that a simple association construct with “owner” and “possession” roles, but without built-in constraints and behaviors, will not suffice for our purposes. Our goal is to capture the full semantics of the ownership relationship in a declarative form.

Ownership has been analyzed alongside the part-whole relationship [1,24,33,48,72], with which it is often confused. Ownership may be defined as a relationship between a legal entity or a natural person (the “owner”) and an owned object (the “possession”), such that the owner has some defined rights to use the possession for the owner’s self-determined purposes. Possessions in general can be categorized as either *real*, *intellectual*, or *personal*. A real possession refers to land or things closely related to it. An intellectual possession constitutes the rights held to an idea (e.g., the design of an invention) or a creative work (such as a musical composition or a novel). Copyrights and patents are the ordinary forms of intellectual possessions. The notion of personal possession encompasses everything that is not real or intellectual. Ownership can have different semantic characteristics (e.g., exclusive versus joint, transferable versus non-transferable) depending on the kind of possession, and these are often defined by statute, securities regulations, and the court of law.

Our ownership relationship comprises a set of semantic (or characteristic) dimensions [24,30,72], each of which captures one aspect of the nature of ownership. Differences between the many kinds of ownership are reflected by the various potential values along these dimensions. Overall, our ownership relationship has five dimensions, referred to as: (1) exclusiveness, (2) dependency, (3) documentation, (4) transferability, and (5) inheritance. In addition to presenting formal specifications of the ownership relationship and its dimensions with the use of OCL [54,70], we define a graphical ownership notation that enhances the Unified Modeling Language (UML) class diagram formalism [9,16,60]. Data modelers can thus draw their desired ownership characteristics in an intuitive way.

Previously, we have used a similar dimensional approach for modeling the part-whole relationship [24]. We have shown with two implementations that it is possible to extend existing object-oriented database (OODB) systems [2,4,10,21,32,34,40,74] with a dimensional semantic relationship such as part-whole [24,39]. In addition to the formal specification of the ownership relationship, we discuss its incorporation into ObjectStore, a commercial OODB management system. We see this extended OODB system as one component of an overall “ownership management system.” Such a system would also require a knowledge-based component, such as the work of McCarty [44]. Consider, e.g., that if two people own a house, then permission must be obtained from each before selling. This is beyond the auspices of the enhanced ownership model that we present here. Additionally, in the world of modern finance involving such instruments as options, futures, mutual funds, etc., some ownership transfers have become bewilderingly complex. An ownership management system

comprising a theoretical model of ownership that incorporates these complexities could function as an “expert” and support aspects of ownership that are not widely known.

A language-based, informal analysis of the differences between the dimensions of the part–whole relationship and the dimensions of ownership has appeared in [19], based on preliminary work in [27,73]. The current dimensional analysis is formally constructed with the use of OCL. The analysis in [19] did include certain additional dimensions beyond those suitable for direct incorporation into a database system. Those dimensions would need to be included in the ownership management system mentioned above, of which the database system is but one component.

While ownership has received little attention in the field of data modeling, it has been investigated in legal reasoning. In [43], a description of the legal groundwork for ownership is presented, and the “language for legal discourse” (LLD) is used to model the ownership relationship. In [44], LLD is used to model a corporate tax case. While the theme of [44] is to implement the legal logic of the case in the TAXMAN II theory, it does give an example of the ownership relationship represented in LLD, which is implemented in PROLOG. The modeling of legal reasoning has played an important foundational role in efforts to model legal concepts in other computer applications [17,20,41,45–47,58,65,71]. In the field of software engineering, it has been argued that ownership is a consequent property of a set of primary aggregation properties [29]. While this characterization of the ownership relationship may work well within the scope of software engineering, our view of ownership is based on real-world examples of its usage.

The remainder of this paper is organized as follows. Section 2 briefly surveys other semantic relationships. In Section 3, the formal aspects of the general ownership relationship are presented. Discussions and OCL formalizations of its five characteristic dimensions, along with accompanying graphical ownership notations for UML, appear in Sections 4–8, respectively. A description of our prototype implementation of the ownership relationship follows in Section 9. Section 10 contains the conclusions.

2. Other semantic relationships

Arguably, the most important and widely used semantic relationship is IS-A (variously referred to as a-kind-of, is-kind-of, subcategory, subclass, etc.). IS-A forms the basis for generalization/specialization and supports inheritance of properties and subsumption-based reasoning about categories of objects. This relationship has been pervasive in almost every knowledge representation system ever developed, including semantic networks [38,66], description logics [51], frame systems [6], object-oriented knowledge representation systems [50], and conceptual graphs [67]. In fact, some people trace an analysis of the IS-A relationship (under a different name) back to Aristotle [57,67]. IS-A has appeared in traditional data modeling methodologies like SDM [28] and extended ER [14,15], and has been the backbone of OODB systems and object-oriented modeling methodologies including UML [9,16,42,55,60], OMT [5,59], Booch [7,8], and Coad/Yourdon [11].

The second-most prevalent semantic relationship has been the part–whole relationship (also called part, part-of, or meronymy). The part relationship is fundamental to our understanding of the physical world. A proposal has been made to judge the quality of existing ontologies partially on their inclusion of the part relationship and the extent of its expressiveness [53]. Treatments of the part relationship can be found in fields ranging from cognitive science [72] and logic [63,64,69] to OODBs [1,33,48]. In previous work, we have developed a comprehensive part–whole model for OODB systems [24]. An extension to the UML metamodel to accommodate part–whole has been proposed in [3].

Other semantic relationships have also been dealt with in the data modeling community, including the role-of relationship [18,52,61,62] and materialization [37,56]. The member-of relationship [49], a set-modeling construct connecting member classes with grouping classes, can have constraints superimposed on it and can thus be seen as a semantic relationship.

3. The ownership relationship

The ownership relationship defined in this section is a conceptual modeling construct that allows data modelers to effectively capture the variegated semantics of ownership in a declarative fashion. This can be accom-

plished in UML class diagrams using a graphical notation that we introduce. The modelers are thus not burdened with the task of having to “hand code” the desired constraints, behavior, etc., associated with a common understanding of ownership. Moreover, repeated rewriting and re-implementation of ownership semantics within different applications is avoided. The incorporation of the ownership relationship into a database system, for example, ultimately passes the burden of integrity enforcements to the underlying system itself. (Such an incorporation of ownership into ObjectStore is discussed in Section 9.)

The formal definition of the ownership relationship follows the model we have previously utilized to define a part-of relationship [24]. The ownership relationship is defined as a tuple, whose components are referred to as characteristic dimensions. Different settings for the various dimensions allow for the specification of a variety of semantic options, all enforced automatically by an underlying database system.

The ownership relationship is a binary relationship connecting a *possession class* with an *owner class*. It is assumed that a class playing the role of the owner class models a legal entity (e.g., a person) capable of functioning in that capacity. That is, the instances of the class can have some defined rights to use instances of the possession class for their self-determined purposes. Additionally, we limit the entities that can be possessions to those objects for which some ownership right(s) can be assigned to the owner. The space of possible possessions is extensive and diverse, and includes real estate, intellectual property, financial instruments, etc. As in everyday life, the ownership relationship can, at times, be built up to form a hierarchy, with objects that are owned in one relationship doing the owning in the context of another. For example, a conglomerate or holding company owns businesses that themselves are in possession of other entities, whether they be real estate, machinery, or even other businesses.

The ownership relationship $O_{B,A}$ connecting the possession class B to the owner class A is defined as the following six-tuple, with the last component a triple in its own right:

$$O_{B,A} = (\Omega_{B,A}, \text{excl}, \text{dpnd}, \text{docu}, \text{trnsfr}, (\text{up}_+, \text{up}_\cup, \text{down}_\cup))$$

where $\Omega_{B,A}$ is a relation from the set $B.\text{allInstances}()$ to $A.\text{allInstances}()$.¹ The pair $(b,a) \in \Omega_{B,A}$ means that the instance b of class B is owned by the instance a of class A .

The remaining elements of the tuple are the relationship’s characteristic dimensions which are called respectively: exclusiveness, dependency, documentation, transferability, and inheritance. Their respective domains and semantics will be discussed in the subsequent sections. We will be using OCL [54] to define the semantics. To accomplish this, there is a need for “reflection” regarding the ownership relationships, i.e., there must be manifest ownership relationships (see, e.g., [26]). In other words, a class itself must be queriable with respect to its ownership relationships. In particular, any class participating in an ownership relationship will have the following two operations: owner-class and possession-class. These operations are class-scoped as indicated by the UML convention of underlined names [60]. The first operation returns the set of all classes that are owner classes with respect to a particular class C . Its header is:

$C : \underline{\text{owner-class}}() : \text{Set}(\text{Class})$

The second operation returns the set of all classes that are possession classes with respect to a particular class C . It has the header:

$C : \underline{\text{possession-class}}() : \text{Set}(\text{Class})$

Additionally, the values of the respective characteristic dimensions can be queried from the participating owner class. (One could assume the same features for the possession class.) In Section 8, we will particularly need the ability to query with respect to the inheritance dimension to form OCL expressions for the derivations of inherited attribute values. Details are given in that section.

The dual operations *owner* and *possession* allow an object to be queried regarding its current set of owners and possessions, respectively. Their headers are:

¹ The *allInstances* feature of OCL [54] returns the set of all instances of a specified class.

$A : \text{possession}(B : \text{Class}) : \text{Set}(B)$

and

$B : \text{possession}(A : \text{Class}) : \text{Set}(A)$

Assuming the existence of an ownership relationship $O_{B,A}$, $a.\text{possession}(B)$ returns the set of all of a 's possessions that are from class B , where $A.\text{allInstances}() \rightarrow \text{includes}(a)$. (Note that the result is `OclUndefined` if the argument B is not in $A.\text{possession-class}()$.) The resultant set of $a.\text{possession}(B)$ will be called the *possession set* of a with respect to $O_{B,A}$. Likewise, $b.\text{owner}(A)$ returns the set of all of b 's owners from class A . Its result is called the *owner set* of b with respect to $O_{B,A}$. As straightforward extensions of *owner* and *possession*, the additional dual operations *allOwners* and *allPossessions* return, respectively, the set of all owners (irrespective of class) and the set of all possessions of a particular object.

The operations above can be captured in UML by augmenting the metamodel with a new metaclass, say, *OwnerPossession*. (Cf. [3] for the inclusion of the metaclass *Whole-Part* for the whole-part relationship.) In previous work, we have described this kind of construction for general semantic relationships (including whole-part) in an object-oriented database environment exhibiting a metaclass facility [26]. Details are beyond the scope of this paper. In Section 9, we do utilize a variant of the notion of metaclass in our implementation of the owner relationship in *ObjectStore*.

In the context of a UML class diagram [9,16,60], a generic ownership relationship $O_{B,A}$ is drawn as a thick line connecting class B to class A (Fig. 1). An open circle (an “O”) appears as the association end on the side of the owner class (in this case A). The “O” serves as a mnemonic for “ownership.” As each of the characteristic dimensions is presented, adornments to this relationship symbol will be introduced to capture the respective dimensional values.

4. Exclusiveness dimension

One way to differentiate between kinds of ownership is to examine how the ownership of a given object can be distributed among owners. Overall, we can form a three-way differentiation along this dimension: (1) an object can be owned exclusively by one owner, (2) it can be held jointly (and in equal amounts) by two or more owners, or (3) its ownership can be distributed unequally over many owners.

An example of exclusive ownership can be seen in the case of an individual retirement account (IRA), which can be owned by exactly one person. Other examples include credit cards, library cards, passports, driver's licenses, and academic diplomas. We note that typically a granting agency (such as a bank or government) keeps legal ownership of an item (as indicated explicitly, e.g., on a credit card) for the purpose of being able to revoke the item, if necessary. However, other major rights of usage are in the hands of the holder of the instrument. Thus, in such cases, we will consider the usage rights as the major rights and the holder as the actual owner. Stock options granted by a firm as part of an employee's compensation package are also owned exclusively by that employee.

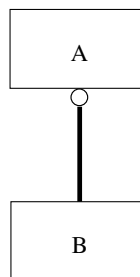


Fig. 1. Ownership relationship between classes B and A .

Joint ownership can be found in many scenarios: A joint bank account, a car with two names on its title, and a house with two names on its title are examples. In these cases, there is an equal partition of the possession. However, many ownership structures are divided unevenly among partners. In a private enterprise shared by two people, one partner may hold 60% while the other holds 40%. With publicly held companies, ownership is distributed among shareholders in percentages that are a function of the respective number of shares held.

To capture these three possibilities, the domain of the exclusiveness dimension is defined to be:

$\{exclusive, joint, percentage\}$

The formal interpretations of each of these values appear in the following, where we also present a graphical notation that enhances the UML class diagram [9,16,60]. Note that OCL keywords such as “context” and “inv” (short for “invariant”) are written in boldface. The former is used to denote the context within which the constraint is defined. The latter labels an OCL expression that specifies an invariant condition in the given context. OCL comments begin with “– –”.

Definition 1. For the ownership relationship $O_{B,A}$, $excl = exclusive$ implies the following invariant:

context $b : B$ **inv:**

$b.owner(A) \rightarrow size() \leq 1$

That is, the cardinality of the owner set (with respect to $O_{B,A}$) for any instance of B is at most one, meaning that such an object can have at most one owner.

The existence of an exclusive ownership relationship implies that only one owner can own a particular object from the given possession class. In other words, if the object is owned, then it cannot enter into an ownership relationship with another owner without completely dissolving the current relationship. These limitations on the owner-set cardinality and associated transactions are automatically enforced.

An exclusive ownership will be denoted with an “X” adornment as shown in Fig. 2, where we see the relationship connecting the class *IRA* with the class *Person*. In this case, any IRA will be owned by at most one person.

Definition 2. For the ownership relationship $O_{B,A}$, $excl = joint$ implies that there are no constraints on the cardinalities of any owner sets with respect to $O_{B,A}$.

With a joint ownership relationship, no restrictions are placed on the number of owners that a given object can have. Each owner is deemed to own the object equally.

Pictorially, a value of *joint* in this dimension is denoted by the lack of the two adornments “X” (introduced above) and “%” (to be introduced below). In Fig. 3, we see that bank accounts can be jointly owned by any number of people.

For the case of percentage ownership, the owner class A in the ownership relationship $O_{B,A}$ is augmented with a method called *share_B* that for a given instance b of the possession class B returns the percentage of ownership exhibited by the particular owner. More specifically:

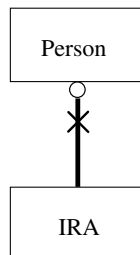


Fig. 2. Exclusive ownership of IRAs.

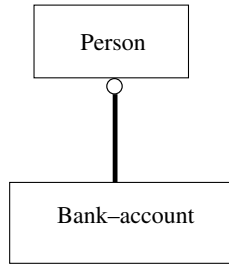


Fig. 3. Joint ownership of bank accounts.

Definition 3. For the ownership relationship $O_{B,A}$, *excl = percentage* implies that there exists a method on A with the header:

$A : \text{share_B}(b : B) : \text{Real}$

such that:

context $a : A$

- if a does *not* own some b , then a 's share of b is 0%, i.e., $a.\text{share_B}(b) = 0.0$
inv: $B.\text{allInstances}() \rightarrow \text{forAll}(b \mid a.\text{possession}(B) \rightarrow \text{excludes}(b) \text{ implies } a.\text{share_B}(b) = 0.0)$
- if a does own some b , then a 's share of b is greater than 0% but no greater than 100%
inv: $B.\text{allInstances}() \rightarrow \text{forAll}(b \mid a.\text{possession}(B) \rightarrow \text{includes}(b) \text{ implies } a.\text{share_B}(b) > 0.0$
and $a.\text{share_B}(b) \leq 100.0$)

and:

context $b : B$

- the sum of the shares of ownership of b cannot exceed 100%
inv: $b.\text{owner}(A) \rightarrow \text{collect}(\text{share_B}(b)) \rightarrow \text{sum}() \leq 100.0$

This kind of ownership relationship allows for the assignment of explicit ownership percentages to the owners of a given object. This is captured formally by the definition of the “share_B” method at the ownership class A . Constraints on share_B, which we refer to as the *individual share method*, are written in terms of three invariants. The first two, on the class A , state respectively that all non-owners have 0% share, and all owners have some share but not more than the total owned object. The third invariant, written with respect to class B , states that the total percentage of ownership of a given possession may not exceed 100%. Ordinarily, we expect all owners to be represented explicitly, in which case:

$b.\text{owner}(A) \rightarrow \text{collect}(\text{share_B}(b)) \rightarrow \text{sum}() = 100.0$

for a given b . However, missing owners can be accommodated by the inequality.

It will be noted that if a single owner happens to have 100% ownership, then he would be the sole proprietor at the given time. That condition is not as strong as an exclusive ownership designation. In the percentage case, the owner could relinquish a portion of the ownership. Under the exclusiveness constraint, only the entire holding could be relinquished.

The individual share method can be defined even in cases of exclusive and joint ownership. For the exclusive case, the value of $a.\text{share_B}(b)$ can only be 0.0 or 100.0. With joint ownership, the value of $a.\text{share_B}(b)$ would be $\frac{1}{n}$, where n is the current number of owners of b .

We can discretize the individual share function to capture the situation of stock ownership, where discrete amounts (shares) of an entity are distributed among a group of owners. In such a situation, share_B would

yield an Integer representing the number of shares owned, such that, in total, no one owned more than the outstanding number of shares. In this context, for each given possession object b of B , there would exist a positive number S_b representing the total number of shares of b . For example, for the company IBM, $S_{\text{IBM}} = 1,550,000,000$. The second invariant on A and the invariant on B in Definition 3 would then be written with respect to S_b rather than the value 100.0. With this construction, we can capture the fact that a person owns 20 shares of IBM, independent of the exact percentage that these 20 shares represent.

A percentage ownership relationship is denoted using the symbol “%” as an adornment. Fig. 4 shows that businesses can be shared on a percentage basis by a number of individuals.

Variations on joint and percentage ownership arise when there may be owners of various kinds for a given possession class. For example, cars may be owned by private citizens or by public companies. If we wished to allow joint ownership of cars between persons and companies, we would simply define a pair of joint ownership relationships, one connecting *Person* and *Car*, and the other connecting *Company* and *Car*. If we wanted to be more restrictive and permit joint ownership only among persons or companies, but not both, we could avail ourselves of the inter-relationship “exclusive-or” (“xor”) constraint of UML and tie the two relationships together, as shown in Fig. 5. In this diagram, both ownership relationships are denoted as joint, so the interpretation is that a car can be owned by any number of persons or any number of companies but not by a mix of persons and companies.

To distribute percentage ownership across classes of owners, we again must utilize the constraint mechanism of UML. For example, a public company can be owned by individuals, mutual funds, and even other companies on a percentage (or discrete share) basis. To model this situation, we define a new inter-relationship constraint, denoted “{ % },” to tie together the three ownership relationships connecting *Company* to *Person*, *Mutual-fund*, and *Company*, respectively (Fig. 6). (Note that the latter is a self-relationship.) The interpretation

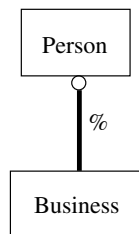


Fig. 4. Businesses owned on a percentage basis.

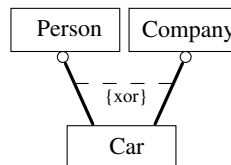


Fig. 5. Cars owned by persons or companies, but not both.

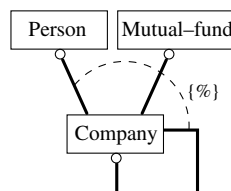


Fig. 6. Companies owned by persons, mutual funds, and other companies.

of the $\{\%\}$ constraint is that the relationships' individual share functions now must collectively satisfy the constraints of Definition 3. For example, the total percentage ownership of a given company across all classes of owners cannot exceed 100%.

5. Dependency dimension

Under certain circumstances, the deletion of an owner object logically implies the need to delete one or more of its possession objects. The reverse, at times, is also true: the deletion of a possession calls for the simultaneous deletion of the owner. The dependency dimension of the ownership relationship deals with this propagation of the deletion operation from owners to possessions, and vice versa.

It might appear, at first glance, that there is no reason to expect that just because one of the objects in an ownership arrangement ceases to exist—or the ownership connection itself dissolves—the other object in the relationship should be deleted, too. However, consider the relationship between a person and their car. While in reality the existences of these objects do not depend on each other, a dependency may arise within a conceptual model. For example, an insurance company may want to distinguish customers that own cars from those who do not. Toward that end, its database schema may include a class *Car-owner*, whose extent consists of all customers that are also car owners, along with a class *Car* and an ownership relationship connecting the two. Because a customer is a car owner by virtue of the fact that there exists an occurrence of an ownership relationship connecting the person to some car, the deletion of that car automatically means that the customer is no longer a car owner (assuming that the customer originally had just one car). Therefore, when an owned car is deleted, we expect the related instance of *Car-owner* to be deleted, too. This kind of dependency is called *owner-on-possession*.

An example of the inverse *possession-on-owner* dependency can be seen in the relationship between an employee and certain benefits like stock options. An employee of a company might be granted an option to purchase shares of the company's stock after, say, three years of employment. If for some reason the employee leaves prior to that time, the option is revoked. In other words, the deletion of the employee implies the deletion of the option.

Of course, an ownership relationship need not exhibit any dependency semantics, which is captured by a value of *nil* for this dimension. Overall, the domain of the dependency dimension is:

$\{\text{owner-on-possession}, \text{possession-on-owner}, \text{nil}\}$

In formalizing this dimension, we will assume the existence of a class-scoped operation *delete* that deletes a given instance of a class. We will denote an application of this operation as $C.\text{delete}(x)$, where C is the class of the object x that is to be deleted. (When the context is clear, C will be omitted.) The *delete* operation always has the postcondition:

post: $x.\text{oclIsUndefined}()$

That is, on completion of the operation, the instance x no longer exists. To capture the semantics of the cascading deletion in OCL, we define the additional operation “ownerDeletionCandidates” on the possession class participating in an ownership relationship exhibiting dependency as follows:

context $b : B$

- returns the set of owners (from A) that only own the given object b . When b is deleted, these owners
- will be deleted, too.

def: $\text{ownerDeletionCandidates}(A : \text{Class}) : \text{Set}(A) = \text{owner}(A) \rightarrow \text{select}(a \mid a.\text{possession}(B) = \text{Set}\{b\})$

In the context of the ownership class participating in such a relationship, *delete* is subject to the following constraint. The OCL notation “ $b@pre$ ” means the object b at the precondition stage. Its use is necessary because b will no longer exist at postcondition time.

Definition 4. For $O_{B,A}$, $dpnd = \text{owner-on-possession}$ implies that:

context $B: \text{delete}(b : B): \text{OclVoid}$

- In addition to deleting b , delete all its owners that qualify, as defined above.
- post:** $b@pre.ownerDeletionCandidates(A) \rightarrow \text{forAll}(a \mid a.\text{oclIsUndefined}())$

In other words, in the case of *owner-on-possession* dependency, if an object a owns only an object b , then the deletion of b implies the deletion of a . Note that we have defined this dependency in a “multi-valued” fashion, in that the deletion of a is avoided if it happens to own another instance of the class B in addition to b .

For the opposite *possession-on-owner* case, the additional property “*possessionDeletionCandidates*” is defined on the owner class in the ownership relationship:

context $a : A$

- returns the set of possessions (from B) whose only owner is the given object a . When a is deleted,
- these possessions will be deleted, too.

def: $\text{possessionDeletionCandidates}(B : \text{Class}) : \text{Set}(B) =$
 $\text{possession}(B) \rightarrow \text{select}(b \mid b.\text{owner}(A) = \text{Set } \{a\})$

The *delete* operation is subject to the following constraint.

Definition 5. For $O_{B,A}$, $dpnd = \text{possession-on-owner}$ implies that:

context $A: \text{delete}(a : A): \text{OclVoid}$

post: $a@pre.possessionDeletionCandidates(B) \rightarrow \text{forAll}(b \mid b.\text{oclIsUndefined}())$

Here, again, the deletion of the dependent possession b is not carried out if it has owners other than a .

Following UML conventions, dependency is denoted by adding an arrowhead to the ownership relationship pointing in the direction of the dependency. Moreover, the relationship is drawn as a dashed line. For *owner-on-possession* dependency, the arrowhead points toward the possession class; for *possession-on-owner* dependency, the arrowhead points toward the owner class and appears directly behind the “O” at the association end. In Fig. 7, we see the class *Car-owner* with a dependency on its possession class *Car*. The possession class *Stock-option* is dependent on its owner class *Employee* in Fig. 8.

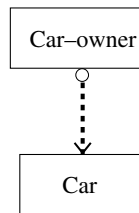


Fig. 7. An owner class dependent on its possession class.

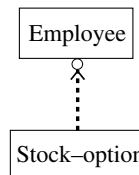


Fig. 8. A possession class dependent on its owner class.

It will be noted that oftentimes exclusive ownership implies a dependency of the possession on the owner. In other words, a value of *exclusive* in the exclusiveness dimension is likely to imply a value of *possession-on-owner* in this dimension. Consider the examples of credit cards, passports, social security cards, driver's licenses, etc. The existence of these exclusively owned objects is predicated on the existence of an owner. A credit card without a card-holder and a driver's license for a non-existent driver should never be issued. A data modeler should keep this thought in mind during the construction of an ownership schema.

6. Documentation dimension

Legal documentation may be a necessary accompaniment to certain ownership relationships. For example, a title is required in the case of car or home ownership. The ownership relationship allows for the explicit specification of the need for the supporting document or certificate to appear as an object in its own right. The documentation domain is defined as:

$\{documented, undocumented\}$

A value of $docu = undocumented$ for a given ownership relationship $O_{B,A}$ means that the relationship requires no supporting data objects verifying ownership connections between instances of B and A .

The existence of an ownership relationship $O_{B,A}$ with $docu = documented$, on the other hand, implies the existence of a special concomitant object class called the *document class* (denoted $Doc_{B,A}$ for the relationship $O_{B,A}$). The instances of $Doc_{B,A}$ represent the documents for ownership of instances of B by instances of A . Note that there is a one-to-one correspondence between the sets:

$B.allInstances() \rightarrow select(b \mid b.owner(A) \rightarrow size() > 0)$ and $Doc_{B,A}.allInstances()$,

but not between the sets $\Omega_{B,A}$ and $Doc_{B,A}.allInstances()$. That is, every instance of B which is actually owned will have a single document associated with it, even if there is more than one owner. No document exists for an object that is not currently owned.

The pair $(b, a) \in \Omega_{B,A}$ implies $Doc_{B,A}.allInstances() \rightarrow includes(d)$ such that d is the document confirming a 's ownership of b . Formally, a documented ownership is a triple, written in OCL as:

Tuple {possession-of-record: B , owner-of-record: A , supporting-document: $Doc_{B,A}$ }

As shown in Fig. 9, this is captured in the model by defining a pair of associations between $Doc_{B,A}$ and A and between $Doc_{B,A}$ and B . The association between $Doc_{B,A}$ and A is many-to-many: Owners can have many possessions, each requiring a document, and a single object (of class B) can have many owners (unless this is overridden by the exclusiveness constraint), with all those owners associated with the same document. Moreover, an existing document must have at least one owner. The association between $Doc_{B,A}$ and B exhibits the following multiplicities, as noted above: An existing document must have exactly one associated possession-of-record, while an instance of B will have a supporting document (at most one) only if it is actually owned.

The instances of $Doc_{B,A}$ can only be manipulated via special ownership transactions with respect to the classes B and A . For example, an instance of $Doc_{B,A}$ can only be created by a transaction that initially estab-

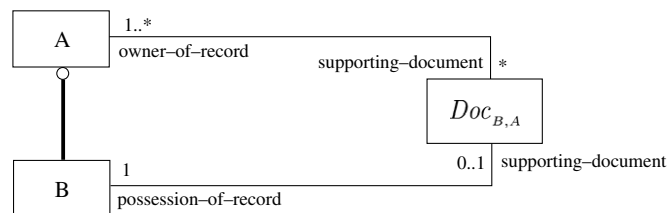


Fig. 9. Class $Doc_{B,A}$ and its associations.

lishes the ownership of an instance of B by an instance of A . Such documents may be transferred or updated during transactions involving the establishment of ownership. In summary, the creation of an ownership link connecting an instance b to an instance a must be accompanied by the creation, transfer, or update of an instance d in $Doc_{B,A}.allInstances()$. From a practical point of view, d may be an electronic certificate with a unique certification number, or it may contain information about the physical location of the ownership document.

Clearly, there exists a dependency between $Doc_{B,A}$ and B and another dependency between $Doc_{B,A}$ and A . If a possession is deleted, then there is no need for a document denoting ownership of it. Also, if all owners of an object are deleted, then a possession is no longer owned.

The creation, deletion, and update semantics with respect to $Doc_{B,A}$ are defined in the following as postconditions involving if-expressions on the operations *acquire* and *relinquish*, defined with respect to the owner class A . An application of the former, $a.acquire(b)$, establishes an ownership connection between the objects b and a , with b the possession and a the owner. On the other hand, $a.relinquish(b)$ denotes the dissolution of ownership. Note that neither *acquire* nor *relinquish* deals at all with the transfer of ownership. A transfer operation can be defined as an application of *relinquish* followed by *acquire*. An attempt to acquire a possession that is owned already by the prospective acquirer has no effect. Likewise, attempting to relinquish an object that is not currently owned has no effect.

Constraint 6.1 deals with the *acquire* operation. Its postcondition states that when an object b becomes a possession of an owner “self,” with “self” being the initial owner, then an instance of the class $Doc_{B,A}$ is created to document this ownership. Otherwise, we have a situation where an additional owner is taking possession of the object. In that case, the new owner is added to the document, as specified in the else-expression of the postcondition.

Constraint 6.2 pertains to the *relinquish* operation. Its postcondition states that the document for an object b is deleted when no owner object owns b any longer. If b happens to be acquired again at some later time, then **Constraint 6.1** dictates that a new document be created. The else-expression handles the case where the current owner “self,” which is relinquishing its ownership, is but one of many owners. In that case, “self” is removed from the document.

Constraint 6.1 (*Document Creation; or Update: acquisition by an additional owner*).

```

context  $A$  :  $acquire(b: B)$ 
  post: if  $b.owner(A)@pre \rightarrow isEmpty()$  then
    -- no owners to start with. New document is created.
     $b.supporting-document.ocIsNew()$  and
     $b.supporting-document.possession-of-record = b$  and
     $b.supporting-document.owner-of-record = Set \{self\}$ 
  else
    -- there was at least one owner to start with. Add the new owner to the document.
     $b.supporting-document.owner-of-record \rightarrow includes(self)$ 
  endif

```

Constraint 6.2 (*Document Deletion: complete relinquishment; or Update: partial relinquishment*).

```

context  $A$  :  $relinquish(b: B)$ 
  post: if  $b.owner(A)@pre = Set \{self\}$  then
    -- the only owner was this object. The document is deleted.
     $b.supporting-document.ocIsUndefined()$ 
  else
    -- there was more than one owner. Remove this owner from the document.
     $b.supporting-document.owner-of-record \rightarrow excludes(self)$ 
  endif

```

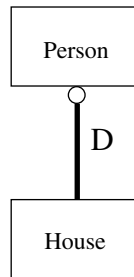


Fig. 10. Documented ownership relationship.

To simplify the semantics, we place an additional restriction on the deletion of possessions and owners in the context of a documented relationship. In such a situation, an owner or possession can only be deleted if it is not currently participating in an ownership connection. That is, some “relinquish” transaction(s) may need to be carried out before the deletion is allowed to take place.

A documented ownership relationship is denoted by adorning the arrow with a “D.” The accompanying document class is implicit and is not displayed in the class diagram—unless explicitly needed—in order not to clutter the diagram. As an example, Fig. 10 shows the documented relationship between *Person* and *House*.

7. Transferability dimension

Another important characteristic of ownership is that of transferability. Ownership can often be readily transferred from current owners to new owners, as in the sale of houses, cars, and general merchandise. Some ownership, though, does not carry this right. For example, in the case of an airline ticket, the owner is not freely permitted to sell it. We can say that this kind of ownership is immutable with respect to the possession; once ownership of the possession is established, it cannot be changed. (This, of course, is not true of all kinds of tickets. Tickets to a baseball game can be sold.) Ownership of other items such as credit cards, library cards, and passports also comes with this limitation.

This dimension of the relationship allows for an explicit distinction between these two kinds of ownership. Its domain is:

$$\{\text{transferable}, \text{non-transferable}\}$$

A value of $trnsfr = \text{transferable}$ for a given ownership relationship $O_{B,A}$ means that the relationship does not impose any restriction regarding a change in ownership of an instance of *B* with respect to instances of *A*.

The other value, *non-transferable*, forbids changes of ownership and imposes other restrictions as well. These non-transferability features are formally captured as constraints on the *make* and *acquire* operations, defined at the possession and owner classes, respectively. The *make* operation is class-scoped and is defined to return a newly created instance of the specific class. A generic specification of this operation in the context of the class *C* is:

context *C*: *make*(): *C*

– – “result” is the newly created object returned by this operation

post: result.ocIsNew() **and** result.ocIsTypeOf(*C*)

The first constraint captures the *essentiality* aspect of non-transferability. When a possession is deemed non-transferable, it naturally relies on an owner as a justification for its existence. Airline tickets and credit cards without owners are ordinarily viewed as non-viable entities. Therefore, when such a possession comes into existence, it must immediately be linked to at least one owner, as specified formally by the precondition and postconditions of [Constraint 7.1](#). Moreover, if the possession is to have more than one owner—assuming the relationship is not exclusive—then all must acquire it at instantiation-time.

Constraint 7.1 (*Essentiality: at least one owner, and all owners established at instantiation-time*).

context $B: \text{make}(\text{initial-owners: Set}(A)): B$
 – the set of initial owners is not empty.
pre: $\text{initial-owners} \rightarrow \text{notEmpty}()$
 – new possession “result” is created with the given set of initial owners.
post: $\text{result.oclIsNew}() \text{ and } \text{result.oclIsTypeOf}(B) \text{ and } \text{result.owner}(A) = \text{initial-owners}$

The second constraint deals with the immutability feature of non-transferability. Again, ownership of non-transferable objects is established at the possession’s instantiation-time and cannot be transferred at any future time.

Constraint 7.2 (*Immutability: no new owners*).

context $A: \text{acquire}(b: B)$
 – if b is not a possession of “self” already, then b is not allowed to become a possession.
post: $b.\text{owner}(A)@pre \rightarrow \text{excludes}(\text{self}) \text{ implies } b.\text{owner}(A) \rightarrow \text{excludes}(\text{self})$

Constraint 7.2 states that an application of the operation *acquire* to a non-transferable possession will fail every time. In other words, as captured by the postcondition, the owner set of b with respect $O_{B,A}$ is invariant under this operation when $O_{B,A}$ is non-transferable. (Equivalently, the possession set of “self” with respect to $O_{B,A}$ is invariant.) Of course, if *acquire* is invoked with respect to one of the initial owners, it has no effect.

Dependency may be a desirable accompanying characteristic of non-transferability. That is, one may want a non-transferable possession to be deleted automatically when its owner(s) happens to be deleted. For example, as noted above, it may make sense to have a credit card destroyed when its owner is deleted. (However, one may decide to keep the object anyway, for historical continuity.) Note, though, that this is not a constraint of the ownership model. One can define a non-transferable ownership relationship with a dependency value of *nil* in a schema. Additionally, the data modeler should keep in mind that exclusiveness often seems to imply non-transferability.

A non-transferable ownership relationship is denoted by adorning the arrow with the symbol \otimes . (The lack of this adornment means the relationship is transferable.) The example of the non-transferable (and exclusive) relationship between *Credit-card* and *Person* is shown in Fig. 11.

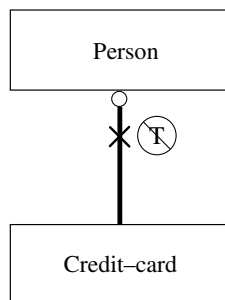


Fig. 11. Non-transferable ownership relationship.

8. Inheritance dimension

8.1. Upward inheritance

There are modeling scenarios where a certain attribute of a possession is naturally assimilated as an attribute of its owner. (As we will discuss below, the reverse assimilation, from owner to possession is also feasible.) For example, the address of a given person may be modeled as the address of the house owned by that person rather than as an intrinsic property of the person. In a database setting, the value of the attribute *address* for a person object could be stored solely with the house object and propagated upward on demand. This would eliminate duplication and guarantee the integrity of *address*'s value. We say that *address* is a derived attribute of the class *Person* that is *inherited upwards* via an ownership relationship from the class *House*.

In previous work, we have defined inheritance of properties via the part relationship and contrasted this with ordinary subclass inheritance [23,24]. Like part relationship inheritance, ownership relationship inheritance has both intensional (schema-level) and extensional (object-level) aspects. A new attribute is defined for the owner class with respect to the same attribute at its possession class(es), and this definition includes a specification of how to derive the attribute's value for a given owner from its related possession.

The derivation of the inherited property's value (for some owner) may involve more than the simple propagation of a single value—as in the address example above. It may call for an accumulation such as the construction of a bag or the summation over a collection of numbers. Extending the address example, we might define *address* as bag-valued to accommodate the situation where a person owns more than one house. For a given person, the addresses of all owned houses would be combined into a bag to form the value of the attribute *address*. (See Fig. 12 for a class diagram that captures this scenario; the notation will be explained below.)

As another example, consider an investor who can own a range of financial instruments, say, stocks, bonds, mutual funds, options, and futures. An investor's net worth is defined as the sum of the nets of all securities owned (Fig. 13). Note that in this case, the attribute *net* of *Investor* is inherited with respect to five different ownership relationships, those connecting *Investor* to *Stock*, *Bond*, etc. The inheritance of a single attribute can be with respect to any number of ownership relationships simultaneously. Therefore, formally, we define the attribute as a property of the inheriting class and define its value in terms of the inheritance dimensions of the respective ownership relationships.

The upward inheritance contribution of a given ownership relationship $O_{B,A}$ is defined by the first two elements of its inheritance dimension, the two sets of attributes up_+ and up_- . The set up_+ , called the “additive up-set,” contains all attributes of the class *B* that contribute to the definition of inherited additive attributes (which will be defined below) at the class *A*. The set up_- , called the “accumulative up-set,” contains attributes of *B* that contribute to the definition of inherited accumulative attributes at *A*. With the benefit of reflection, these up-sets can be obtained from *A* with the use of the operations *upSetAdd* and *upSetAcc*.

For a given ownership relationship $O_{B,A}$, the two up-sets have the following constraint:

context *A*

inv: $A.\text{upSetAdd}(B) \rightarrow \text{intersection}(A.\text{upSetAcc}(B)) \rightarrow \text{size}() = 0$

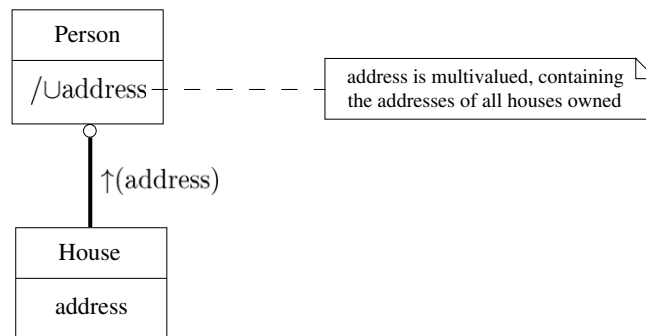


Fig. 12. Attribute *address* inherited by *Person* from *House*.

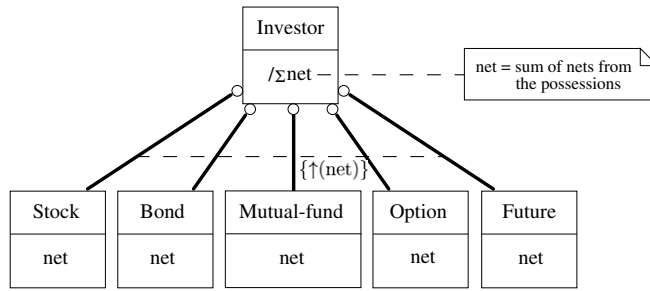


Fig. 13. Attribute *net* inherited additively by *Investor* from its five possession classes.

That is, the two sets are required to be disjoint. Additionally:

context *A*

inv: $A.\text{possession-class}() \rightarrow \text{exists}(B \mid A.\text{upSetAdd}(B) \rightarrow \text{includes}(\text{attr}))$ **implies**
not $A.\text{possession-class}() \rightarrow \text{exists}(B \mid A.\text{upSetAcc}(B) \rightarrow \text{includes}(\text{attr}))$

This constraint states that if some ownership relationship $O_{B,A}$ contributes *attr* as part of the definition of an inherited additive attribute, then no ownership relationship with the owner class *A*—including $O_{B,A}$ itself—can contribute *attr* as part of the definition of an inherited accumulative attribute. The reverse is also true: the same attribute cannot contribute to an accumulative attribute if it already contributes to an additive attribute.

Definition 6. Let $O_{B_1,A}, O_{B_2,A}, \dots, O_{B_n,A}$ ($n \geq 1$) be ownership relationships such that $\text{attr} \in \text{up}_+$ for all $O_{B_i,A}$. (Moreover, let *T* be the data type of *attr* at each class B_i .) Then there exists an attribute *attr*, called an *inherited additive attribute*, on *A* such that:

context $a : A : \text{attr} : T$

— assuming the value of *attr* is defined at all the involved possessions, then *attr*’s value here is the sum;
 — otherwise, its value is OclUndefined

derive: $a.\text{allPossessions}() \rightarrow \text{select}(b \mid A.\text{upSetAdd}(b.\text{type}()) \rightarrow \text{includes}(\text{attr})) \rightarrow \text{collect}(\text{attr}) \rightarrow \text{sum}()$

Since OCL 2.0 no longer directly supports type reflection,² we have assumed the existence of an operation “type” that returns the *immediate* or *direct* type [10,60] of the given object (see, e.g., [31,74]). Under the assumption that any instance has exactly one direct type (as in [10,31,74]), this operation can be defined in terms of the currently existing built-in operation `oclIsTypeOf`.

According to this definition, *attr* in any one of the sets up_+ induces the inheritance of a new attribute by the class *A*. The value of *attr* for a given owner *a* is defined to be the summation of the values of *attr* for all of *a*’s possessions in the respective classes. This is the reason for the name “additive up-set.” If an *a* does not have any such possessions, then its value for *attr* will be OclUndefined. Additionally, if the value of *attr* is OclUndefined for any of the existing possessions, then *attr* at *a* will also be OclUndefined.

For simplicity, we have assumed that the data type of *attr* is *T* at each of the classes B_i . It is a straightforward matter to extend the definition to allow for the inclusion of type-casting operations to reconcile different data types. So, for example, mixed integer and real-number addition can be accommodated. Moreover, utilizing an operation other than “sum” is an extension that could be easily incorporated if a data modeler deemed it meaningful and necessary.

As an example, the class *Investor* will inherit the additive attribute *net* with respect to its five possession classes, *Stock*, *Bond*, *Mutual-fund*, *Option*, and *Future*. The value for a given investor *v* is as follows:

² Previous versions of OCL included an operation `OclType` that was deprecated in OCL 2.0 [54].

context $v : \text{Investor} : \text{net} : \text{Real}$

derive: $v.\text{possession}(\text{Stock}) \rightarrow \text{union}(v.\text{possession}(\text{Bond})) \rightarrow \text{union}(v.\text{possession}(\text{Mutual-fund})) \rightarrow$
 $\text{union}(v.\text{possession}(\text{Option})) \rightarrow \text{union}(v.\text{possession}(\text{Future})) \rightarrow \text{collect}(\text{net}) \rightarrow \text{sum}()$

If a given investor does not own any such security, then his net value will be `OclUndefined`.

Inherited accumulative attributes are defined similarly to additive attributes, with the difference being that the many data values from the possessions are brought together into a single bag rather than being added together into a single data value of the type.

Definition 7. Let $O_{B_1,A}, O_{B_2,A}, \dots, O_{B_n,A}$ ($n \geq 1$) be ownership relationships such that $\text{attr} \in \text{up}_{\cup}$ for all $O_{B_i,A}$. (Moreover, let T be the data type of attr at each class B_i .) Then there exists an attribute attr , called an *inherited accumulative attribute*, on A such that:

context $a : A : \text{attr} : \text{Bag}(T)$

– assuming the value of attr is defined at all the involved possessions, then attr 's value here
 – is the bag containing them; otherwise, its value is `OclUndefined`

derive: $a.\text{allPossessions}() \rightarrow \text{select}(b \mid A.\text{upSetAcc}(b.\text{type}()) \rightarrow \text{includes}(\text{attr})) \rightarrow \text{collect}(\text{attr})$

An inherited accumulative attribute is bag-valued, with type $\text{Bag}(T)$. If the value of attr is `OclUndefined` for any possession, then it is `OclUndefined` for the owner, too. The value of attr cannot be the empty bag at any owner.

From our example above, we have the accumulative attribute *address* of *Person* inherited from *House*:

context $r : \text{Person} : \text{address} : \text{Bag}(\text{String})$

derive: $r.\text{possession}(\text{House}) \rightarrow \text{collect}(\text{address})$

It should be noted that the inheritance is defined with respect to only one ownership relationship in this case.

The inheritance of an attribute via ownership is denoted in a class diagram by two features, one adorning the relationship line(s) and the other augmenting the owner class. First, if only one ownership relationship contributes to the inheritance, then it is adorned with a label consisting of the name of the attribute in parentheses, preceded by an up-arrow as in “↑(address).” Otherwise, the label is written using the UML constraint notation tying together all contributing ownership symbols. (As usual, the label is shown inside braces.) Second, the name of the inherited attribute is written together with the other intrinsic attributes as part of the definition of the owner class. An inherited attribute is distinguished by the presence of either “/Σ” or “/∪” as a prefix to its name; in the case of the former prefix, we have an additive attribute; in the case of the latter, an accumulative attribute. The example of the inherited accumulative attribute *address* is shown in Fig. 12. Since it involves only one ownership relationship, the label is placed directly on that relationship's line. The attribute *address* at the class *Person* has “/∪” as its prefix. The example of the inherited additive attribute *net* appears in Fig. 13, where the five ownership relationships are tied together to indicate that they all contribute to the inheritance. The attribute *net* at *Investor* has the prefix “/Σ” in this case.

8.2. Downward inheritance

The assimilation of an attribute defined at an owner by its possession is also a viable modeling feature. Consider, e.g., a joint bank account whose “name” is exactly the combination of the names of its owners. A credit card also exhibits similar behavior (with the name limited to that of a single owner). A passport's address is its owner's address (which, in fact, might be derived via an upward inheritance from the owner's house, as discussed above).

The downward inheritance contribution of a given ownership relationship $O_{B,A}$ is defined by the set of attributes down_{\cup} , the last element of its inheritance dimension. This set, called the “down-set,” contains all

attributes of the class A that contribute to the definition of inherited attributes at the class B . The value of the down-set for $O_{B,A}$ is available via the operation *downSet* on A .

The sets $down_{\cup}$ and up_{+} are required to be disjoint in order to avoid circular definitions. The same is true of $down_{\cup}$ and up_{\cup} . It is possible, though, that an attribute will appear in one of the up-sets of another ownership relationship and in $down_{\cup}$. That is, the same attribute might be inherited upwards across one ownership relationship and downwards across another. The example of the passport's address is such a case.

Formally, a downward inherited attribute $attr$ is defined similarly to an upward accumulative attribute. (It will be noted that a downward analog to an upward additive attribute is not included in our model because we have found no motivating examples to justify it.)

Definition 8. Let $O_{B,A_1}, O_{B,A_2}, \dots, O_{B,A_n}$ ($n \geq 1$) be ownership relationships such that $attr \in down_{\cup}$ for all O_{B,A_i} . (Moreover, let T be the data type of $attr$ at each class A_i .) Then there exists an attribute $attr$, called a *downward inherited attribute*, on B such that:

context $b : B :: attr : Bag(T)$

– – assuming the value of $attr$ is defined at all the involved owners, then $attr$'s value here

– – is the bag containing them; otherwise, its value is *OclUndefined*

derive: $b.allOwners() \rightarrow select(a \mid a.type().downSet(B) \rightarrow includes(attr)) \rightarrow collect(attr)$

A downward inherited attribute is bag-valued, with data type $Bag(T)$. If the value of $attr$ is *OclUndefined* for any owner, then it is *OclUndefined* for the possession, too.

While formally we allow contributions to a downward inherited attribute to come from any number of ownership relationships, in practice we have found that most downward inherited attributes are derived via a single relationship. An example is the attribute *name* defined at *Bank-account* via an inheritance from *Person*:

context $c : Bank-account :: name : Bag(String)$

derive: $c.owner(Person) \rightarrow collect(name)$

While there may be multiple names on an account, they are all contributed through one ownership relationship in the schema. Of course, this one relationship has multiple occurrences, one for each owner.

In cases where it is guaranteed that a possession will have no more than one owner (i.e., when the ownership relationship is exclusive), we can formally forgo the accumulation feature and allow the inheritance to be “invariant” [24], with the value at the possession exactly that at its owner. In other words, the single value need not be accumulated into a bag. We have such a situation with the name on a credit card:

context $e : Credit-card :: name : String$

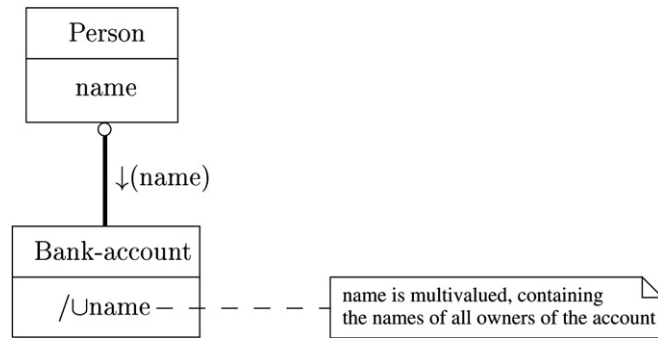
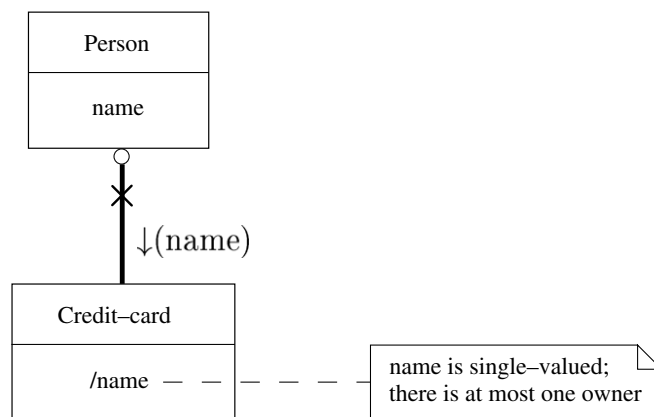
– – if there's an owner, then the owner's name is the card's

derive: $e.owner(Person) \rightarrow any(true).name$

The downward inheritance of an attribute via an ownership relationship is denoted in a similar manner to upward inheritance. The ownership relationship contributing to the inheritance is adorned with a label consisting of the name of the attribute in parentheses preceded by a down-arrow. The name of the inherited attribute, prefixed with “/∪,” is written as part of the definition of the possession class. The example of the downward inherited attribute *name* of the class *Bank-account* appears in Fig. 14. In the case of an invariant attribute, we omit the “∪” in its prefix, as can be seen in Fig. 15.

9. Incorporating ownership into an OODB system

We have developed a preliminary implementation of our ownership model using the commercial OODB system ObjectStore 6.0 Service Pack 7, running on a Sun SPARCcenter 2000 workstation with a Sun Workshop 5.3 C++ compiler under Solaris 8. In ObjectStore, each persistent class has an associated metaclass that consists of definitions internal to ObjectStore that are used to describe the user class.

Fig. 14. Attribute *name* inherited downwards by *Bank-account* from its owner class *Person*.Fig. 15. Attribute *name* inherited by *Credit-card*.

The use of the term “metaclass” in ObjectStore is therefore slightly different from the common definition. In object-oriented database theory, metaclasses are classes that have normal classes as their instances. Metaclasses are necessary because languages such as C++ do not maintain important information about classes across compilation. Thus, this information is maintained at run-time by creating an instance of a metaclass that represents all the information of the corresponding class. Metaclasses are a powerful implementation construct that has been utilized by the authors and others in previous research, including in the implementation of a part model [3,24,25,35,36].

In our ownership implementation, we create two metaclasses to represent the necessary information and operations implementing ownership semantics. These classes are called *ownership_obj* and *ownership_spec*.

The class *ownership_obj*, whose definition is shown below, is the base class for objects that participate in ownership relationships, namely, owner and possession class instances. It contains two sets of references (*os_set* in ObjectStore) of owner objects and possession objects. Only one of the sets will have a value in most situations. But if a certain class is both an owner class and a possession class (which is possible with companies, for example), then both reference sets may contain object references. The class *ownership_obj* also defines methods such as *addOwner*, *addPossession*, *deleteOwner*, *replaceOwner*, and the *MAKE* function we mentioned earlier in Section 6.

```

class ownership_obj {
private:
    os_Set<ownership_obj *> _owner;
    os_Set<ownership_obj *> _possession;
    :
}
  
```

```

public:
    ownership_obj(); // constructor
    ~ownership_obj(); // destructor
    void addOwner(ownership_obj* _o);
    void addPossession(ownership_obj* _p);
    :
}

```

The partial definition of the class *car_owner* (from Fig. 7) appears in the following. It inherits from *ownership_obj* all the ownership-related member functions. Note that the function *MAKE* should be used to create a *car_owner* object, passing a specific possession object (a car) as an argument. *MAKE*, in turn, calls functions defined in the class *ownership_obj* to verify whether this is a legal ownership object creation.

```

class car_owner: public ownership_obj {
private:
    // private data members for class car_owner
    :
public:
    MAKE(car* _c);
    :
}

```

The class *ownership_spec* maintains the specification of ownerships defined between classes. Only one object will be created for one ownership relationship, and this object is shared by all participating objects. For the example shown in Fig. 15, only one object will be created for the ownership relationship between the classes *Person* and *Credit_card*.

The ownership specifications in the application schema of a user have to be loaded into the database before any ownership objects can be created. This is because any ownership related operations require access to the specification defined between owner and possession classes. For example, the function *MAKE* will consult the ownership specification to determine whether the creation is legal.

Our implementation of the ownership model is on the Web at:

<http://vague.njit.edu:8080/ownership/>

Necessary operations are included at the Web-site to allow users to experiment with the system. The current version of the implementation contains the basic functionality of creating, deleting, displaying, and connecting objects through ownership relationships.

The Web-site contains five demonstrations, each focusing on one of the dimensions defined in our ownership model. The first demonstration is based on the schema in Fig. 4, containing the classes *Business* and *Person*, and shows how sharing on a percentage basis is enforced by our model. When users invoke functions (via our form-based interface) such as *addOwner* and *addPossession*, the functions check the exclusiveness definition before establishing an ownership relationship between owner and possession objects. These functions automatically enforce the constraints defined for the individual share function (Section 4). If, for example, the addition of a new owner were to imply a total ownership greater than 100%, then the transaction would be disallowed. In our OODB implementation, the *ownership_obj* class has an object reference *_SHARE_* pointing to a persistent object (class *_exclusive_percentage_*) that maintains the percentage of the ownership relationship. Note that the *_SHARE_* attribute in owner and possession objects points to the same percentage object. Fig. 16 shows the screen through which the user issues a command to establish a percentage ownership. In this case, the user is trying to give person Smith 90% ownership of the business PizzaShed. The screen image in Fig. 17 shows that the command was unsuccessful because two other persons, Date and Booch, already own a combined 60% of PizzaShed.

Fig. 16. Form-based request to establish percentage ownership.

The second demonstration refers to Fig. 7. The classes *Car* and *Car_owner* are connected with a dependent relationship, with an owner-on-possession dependence. It shows that deletion of an object is propagated to the dependent object. Similar to object creation with the *MAKE* function, users employ the function *DELETE*, defined in *ownership_obj*, to delete an object. This function will check the ownership specification between owner and possession objects. If there is a dependence defined, the associated objects will be deleted. If the extra object deletion causes an ownership violation, the original deletion will be aborted by throwing an exception.

Fig. 10 shows a documented ownership relationship. This functionality is shown in the third demonstration. The classes *House* and *Person* are connected in a way such that every ownership of a house by a person is documented. A persistent class *_document_Person_House_* will be created automatically to carry the document information. An object of this class will be created when an ownership relationship is established between a house and a person object. When the ownership is removed, this object will be destroyed. Users may access the owner and possession objects through this object.

Fig. 11 is implemented in the fourth demonstration. Because credit cards are non-transferable, we use the classes *Credit_card* and *Person*. The transferability dimension is set to non-transferable. In the class *ownership_obj*, there is a Boolean attribute *_transferable_* to implement the transferability dimension. Any change-of-ownership operation will first check whether this attribute is true. If it is defined as false, any attempt to change the existing ownership relationship will be aborted (by throwing an exception).

Finally, inheritance, as shown in Fig. 14, is demonstrated. The classes *Bank_account* and *Person* are used for downward inheritance. This makes it possible to retrieve the “name” of a person that owns a number of banking accounts, without writing any code. In our implementation, we use functions to realize inheritance, instead of creating an extra reference. When we create our owner and possession classes, extra functions will

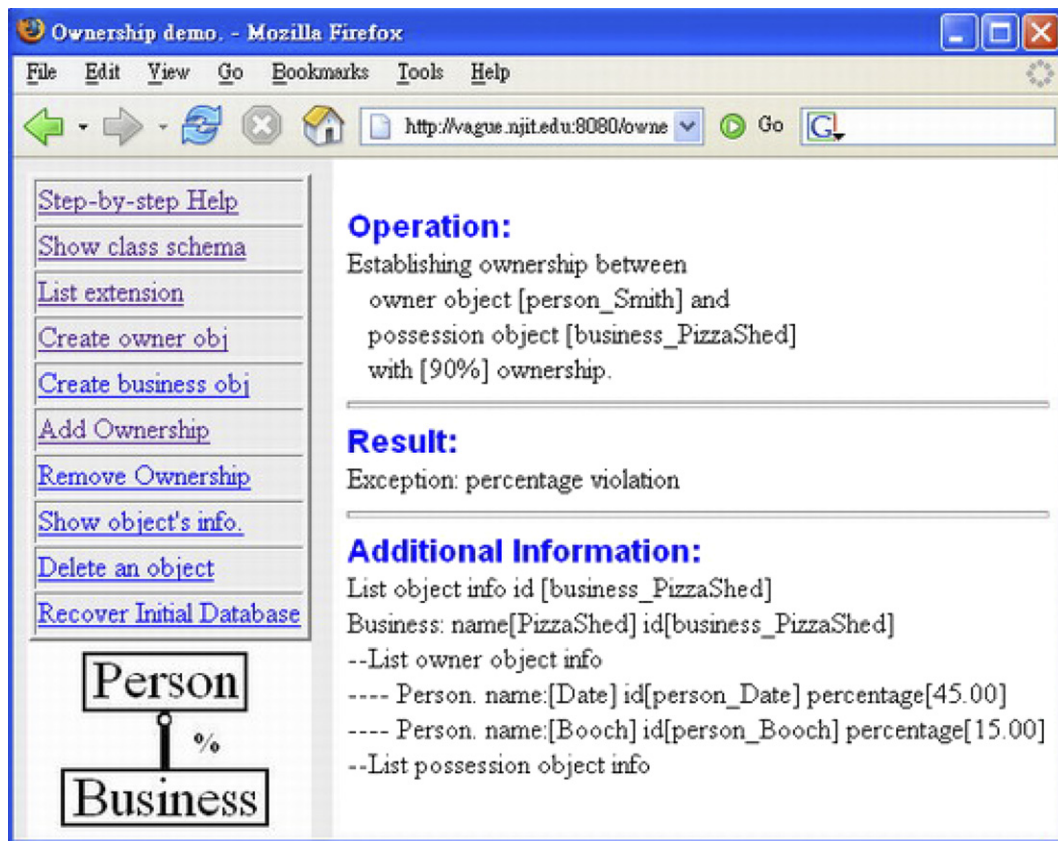


Fig. 17. The result screen for the command issued in Fig. 16.

be added into the original C++ code. These functions can be generated by a utility according to the schema. In our demonstration, an extra function `get_name()` is put in the class *Bank_account*. It references the owner object inherited from *ownership_obj* and invokes `get_name()` defined in the class *Person*.

10. Conclusions

We have presented a comprehensive ownership relationship as a conceptual data modeling construct. This multi-faceted relationship can exhibit a wide range of ownership semantics. It imposes various ownership constraints and requirements on the states of owner objects and possession objects and the transactions that can be carried out with respect to them. Formally, the ownership relationship is divided into five characteristic dimensions, covering the issues of exclusive versus joint ownership, dependency among owners and possessions, documentation, transferability, and inheritance of attributes. Using OCL, formal definitions were provided for all distinctions along these characteristic dimensions. Additionally, we formally defined two kinds of inherited attributes, additive attributes and accumulative attributes, that can be associated with ownership relationships exhibiting the inheritance feature. Such attributes allow, e.g., the declarative definition of a person's net worth as the sum of the values of his or her financial possessions (such as stocks, bonds, mutual funds, etc.). This kind of specification can be made without the need for tedious low-level programming. Overall, the ownership relationship allows an application designer interested in modeling ownership to work at a high conceptual level and avoid a great deal of programming.

To complement the formalisms, we presented graphical ownership notations that augment UML class diagrams and allow for the diagrammatic specification of ownership schemas. A partial implementation of our ownership model has been carried out in the OODB system ObjectStore. A sample application is available on the Web.

Regarding future work, we foresee a database system equipped with our ownership relationship as one component of a larger “ownership management system,” comprising additional rule-based and knowledge features. Such a system will be able to manage subtle aspects of ownership and its associated transactions. Overall, it will encompass other characteristics of ownership not captured by our current ownership model. Examples include default values for inherited attributes and settlement issues involving ownership transference. Ownership also tends to imply obligations. Thus, the owner of a store must clean the sidewalk in front of it. This goes well beyond what our model addresses. Another class of characteristics involves action and temporal limitations. An example of an action limitation is the suspension of the rights to use an owned object, such as with a credit card where the credit limit has been exceeded. A temporal limitation can often be found in the ownership of a time-share, which tends to expire after a prescribed period of time, say, 99 years. Also, a stock option has an expiration date, and ownership effectively ceases after that time.

References

- [1] A. Artale, E. Franconi, N. Guarino, L. Pazzi, Part-whole relations in object-centered systems: an overview, *Data & Knowledge Engineering* 20 (3) (1996) 347–383.
- [2] F. Bancilhon, C. Delobel, P. Kanellakis (Eds.), *Building an Object-Oriented Database System: The Story of O₂*, Morgan Kaufman Publishers, Inc., San Mateo, CA, 1992.
- [3] F. Barbier, B. Henderson-Sellers, A. Le Parc-Lacayrelle, J.-M. Bruel, Formalization of the whole-part relationship in the Unified Modeling Language, *IEEE Transactions on Software Engineering* 29 (5) (2003) 459–470.
- [4] E. Bertino, L. Martino, *Object-Oriented Database Systems: Concepts and Architectures*, Addison-Wesley Publishing Company, New York, NY, 1993.
- [5] M. Blaha, W. Premerlani, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, Upper Saddle River, NJ, 1998.
- [6] D. Bobrow, T. Winograd, An overview of KRL, *Knowledge Representation Language*, *Cognitive Science* 1 (1) (1997) 3–46.
- [7] G. Booch, *Object-Oriented Design*, Benjamin/Cummings, Redwood City, CA, 1991.
- [8] G. Booch, *Object-Oriented Analysis and Design with Applications*, second ed., Benjamin/Cummings, Redwood City, CA, 1994.
- [9] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, second ed., Addison-Wesley, Boston, MA, 2005.
- [10] R.G.G. Cattell, D.K. Barry (Eds.), *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2000.
- [11] P. Coad, E. Yourdon, *Object-Oriented Analysis*, second ed., Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [12] E.F. Codd, Extending the database relational model to capture more meaning, *ACM Transactions on Database Systems* 4 (4) (1979) 397–434.
- [13] C.J. Date, *An Introduction to Database Systems*, seventh ed., Addison-Wesley, Boston, MA, 2002.
- [14] R. Elmasri, S.B. Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Co., Inc., New York, NY, 1989.
- [15] R. Elmasri, J. Weeldreyer, A. Hevner, The category concept: an extension to the entity-relationship model, *International Journal of Data and Knowledge Engineering* 1 (1) (1985).
- [16] M. Fowler, *UML Distilled*, third ed., Addison-Wesley, Boston, MA, 2004.
- [17] R. García, J. Delgado, An ontological approach for the management of rights data dictionaries, in: M.-F. Moens, P. Spyns (Eds.), *Legal Knowledge and Information Systems: JURIX 2005: The Eighteenth Annual Conference*, Brussels, Belgium, December 2005, pp. 137–146.
- [18] J. Geller, Y. Perl, E. Neuhold, Structure and semantics in OODB class specifications, *SIGMOD Record* 20 (4) (1991) 40–43.
- [19] J. Geller, C. Rush, L. Liu, M. Halper, Y. Perl, Ownership semantics in the financial domain, in: R. Meersman, T. Dillon, K. Aberer (Eds.), *Proc. 9th IFIP 2.6 Working Conference on Database Semantics (DS-9): Semantic Issues in e-Commerce Systems*, Hong Kong, April 2001, pp. 340–356.
- [20] G. Governatori, A. Rotolo, Modelling contracts using RuleML, in: T. Gordon (Ed.), *Legal Knowledge and Information Systems: JURIX 2004: The Seventeenth Annual Conference*, Berlin, Germany, December 2004, pp. 141–150.
- [21] P.M.D. Gray, K.G. Kulkarni, N.W. Paton, *Object-Oriented Databases: A Semantic Data Model Approach*, Prentice Hall, New York, NY, 1992.
- [22] R. Gupta, E. Horowitz (Eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [23] M. Halper, J. Geller, Y. Perl, Value propagation in OODB part hierarchies, in: B. Bhargava, T. Finin, Y. Yesha (Eds.), *CIKM-93, Proc. 2nd Int. Conference on Information and Knowledge Management*, Washington, DC, November 1993, pp. 606–614.
- [24] M. Halper, J. Geller, Y. Perl, An OODB part-whole model: semantics, notation, and implementation, *Data & Knowledge Engineering* 27 (1) (1998) 59–95.

- [25] M. Halper, J. Geller, Y. Perl, W. Klas, Integrating a part relationship into an open OODB system using metaclasses, in: N. Adam, B. Bhargava, Y. Yesha (Eds.), CIKM-94, Proc. 3rd Int. Conference on Information and Knowledge Management, Gaithersburg, MD, 1994, pp. 10–17.
- [26] M. Halper, L. Liu, J. Geller, Y. Perl, Frameworks for incorporating semantic relationships into object-oriented database systems, *Concurrency and Computation: Practice and Experience* 15 (15) (2003) 1337–1362.
- [27] M. Halper, Y. Perl, O. Yang, J. Geller, Modeling business applications with the OODB ownership relationship, in: R.S. Freedman (Ed.), Proc. 3rd Int. Conference on AI Applications on Wall St., New York, NY, June 1995, pp. 2–10.
- [28] M. Hammer, D. McLeod, Database description with SDM: a semantic database model, *ACM Transactions on Database Systems* 6 (3) (1981) 351–386.
- [29] B. Henderson-Sellers, F. Barbier, What is this thing called aggregation? in: R. Mitchell, A.C. Wills, J. Bosch, B. Meyer (Eds.), Proc. TOOLS29, Nancy, France, June 1999, pp. 216–230.
- [30] M.N. Huhns, L.M. Stephens, Plausible inferencing using extended composition, in: Proc. IJCAI-89, Detroit, MI, 1989, pp. 1420–1425.
- [31] S.E. Keene, Object-Oriented Programming in Common Lisp, Addison-Wesley Publishing Co., Inc., Reading, MA, 1989.
- [32] H.-J. Kim, Algorithmic and computational aspects of object-oriented database schema design, in: [22], pp. 26–61.
- [33] W. Kim, E. Bertino, J.F. Garza, Composite objects revisited, in: Proc. 1989 ACM SIGMOD Int. Conference on the Management of Data, Portland, OR, June 1989, pp. 337–347.
- [34] W. Kim, F.H. Lochovsky (Eds.), Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, NY, 1989.
- [35] W. Klas, A Metaclass System for Open Object-Oriented Data Models, PhD thesis, Technical University of Vienna, January 1990.
- [36] W. Klas, Tailoring an object-oriented database system to integrate external multimedia devices, in: Workshop on Heterogeneous DBs & Semantic Interoperability, Boulder, CO, 1992.
- [37] M. Kolp, A metaobject protocol for reifying semantic relationships into reflective systems, in: Proc. 4th Doctoral Consortium of the 9th Int. Conference on Advanced Information Systems Engineering (CAiSE'97), Barcelona, Spain, June 1997, pp. 89–100.
- [38] F. Lehmann (Ed.), Semantic Networks in Artificial Intelligence, Pergamon Press, Tarrytown, NY, 1992.
- [39] L. Liu, M. Halper, Incorporating semantic relationships into an object-oriented database system, in: Proc. Thirty-Second Hawaii Int. Conference on System Sciences (HICSS-32), Maui, HI, January 1999, 10 p. (CD-ROM) © 1999 by the IEEE.
- [40] M.E.S. Loomis, Object Databases: The Essentials, Addison-Wesley Publishing Co., Reading, MA, 1995.
- [41] N. Love, M.R. Genesereth, Computational law, in: Proc. Tenth Int. Conference on Artificial Intelligence and Law (ICAAIL 2005), Bologna, Italy, June 2005, pp. 205–209.
- [42] R.A. Maksimchuk, E.J. Naiburg, UML for Mere Mortals, Addison-Wesley, Boston, MA, 2005.
- [43] L.T. McCarty, Ownership: a case study in the representation of legal concepts (preliminary draft), in: Conference in Celebration of the 25th Anniversary of the Instituto Documentazione Giuridica, Florence, Italy, 1993, pp. 1–23.
- [44] L.T. McCarty, An implementation of Eisner v. Macomber, in: Proc. Fifth Int. Conference on Artificial Intelligence and Law, College Park, MD, May 1995, pp. 276–286.
- [45] L.T. McCarty, Some arguments about legal arguments, in: Proc. Sixth Int. Conference on Artificial Intelligence and Law, Melbourne, Australia, July 1997, pp. 215–224.
- [46] R.N. Moles, Logic programming: an assessment of its potential for artificial intelligence applications in law, *Journal of Law and Information Science* 2 (2) (1991).
- [47] L. Mommers, Application of a knowledge-based ontology of the legal domain in collaborative workspaces, in: Proc. Ninth Int. Conference on Artificial Intelligence and Law (ICAAIL 2003), Edinburgh, Scotland, June 2003, pp. 70–76.
- [48] R. Motschnig-Pitrik, The semantics of parts versus aggregates in data/knowledge modelling, in: Proc. CAiSE'93, Lecture Notes in Computer Science, no. 685, Springer, 1993, pp. 352–373.
- [49] R. Motschnig-Pitrik, V.C. Storey, Modelling set membership: the notion and the issues, *Data & Knowledge Engineering* 16 (1995) 145–185.
- [50] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis, Telos: representing knowledge about information systems, *TOIS* 8 (4) (1990) 325–362.
- [51] D. Nardi, R.J. Brachman, An introduction to description logics, in: F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook, Cambridge University Press, 2002, pp. 5–44.
- [52] E.J. Neuhold, M. Schrefl, Dynamic derivation of personalized views, in: Proc. 14th VLDB, Long Beach, CA, 1988.
- [53] N.F. Noy, C.D. Hafner, The state of the art in ontology design: a survey and comparative review, *AI Magazine* 18 (3) (1997) 53–74, Fall.
- [54] Object Management Group, OCL 2.0 Specification, June 2005. Available from: <<http://www.omg.org>>.
- [55] M. Page-Jones, Fundamentals of Object-Oriented Design in UML, Addison-Wesley, Boston, MA, 2000.
- [56] A. Pirotte, E. Zimányi, D. Massart, T. Yakusheva, Materialization: a powerful and ubiquitous abstraction pattern, in: Proc. VLDB'94, Santiago, Chile, 1994, pp. 630–641.
- [57] Porphyry, On Aristotle's Categories, Cornell University Press, Ithaca, NY, 1992 (trans. S.K. Strange).
- [58] J.H. Reichman, Electronic information tools: the outer edge of world intellectual property law, *University of Dayton Law Review* 17 (1992) 797–838.
- [59] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [60] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, second ed., Addison-Wesley, Boston, MA, 2005.
- [61] M. Schrefl, E.J. Neuhold, A knowledge-based approach to overcome structural differences in object-oriented database integration, in: Proc. IFIP Working Conference on the Role of AI in Database and Information Systems, Guangzhou, China, North Holland, 1988.

- [62] M. Schrefl, E.J. Neuhold, Object class definition by generalization using upward inheritance, in: Proc. 4th Int. Conference on Data Engineering, Los Angeles, CA, February 1988, pp. 4–13.
- [63] P. Simons, *Parts, A Study in Ontology*, Clarendon Press, Oxford, 1987.
- [64] B. Smith, Mereotopology: a theory of parts and boundaries, *Data & Knowledge Engineering* 20 (3) (1996) 287–303.
- [65] L.B. Solum, Legal personhood for artificial intelligences, *North Carolina Law Review* 70 (1992) 1231.
- [66] J.F. Sowa, *Principles of Semantic Networks, Explorations in the Representation of Knowledge*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.
- [67] J.F. Sowa, *Knowledge Representation*, Brooks/Cole, Pacific Grove, CA, 2000.
- [68] V.C. Storey, Understanding semantic relationships, *VLDB Journal* 2 (4) (1993) 455–488.
- [69] A.C. Varzi, Parts, wholes, and part-whole relations: the prospects of mereotopology, *Data & Knowledge Engineering* 20 (3) (1996) 259–286.
- [70] J.B. Warmer, A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, second ed., Addison-Wesley, Boston, MA, 2003.
- [71] R. Winkels, A. Boer, E. de Maat, T.M. van Engers, M. Breebaart, H. Melger, Constructing a semantic network for legal content, in: Proc. Tenth Int. Conference on Artificial Intelligence and Law (ICAIL 2005), Bologna, Italy, June 2005, pp. 125–132.
- [72] M.E. Winston, R. Chaffin, D.J. Herrmann, A taxonomy of part-whole relations, *Cognitive Science* 11 (4) (1987) 417–444.
- [73] O. Yang, M. Halper, J. Geller, Y. Perl, The OODB ownership relationship, in: Proc. Int. Conference on Object-Oriented Information Systems (OOIS'94), London, UK, December 1994, pp. 389–403.
- [74] S.B. Zdonik, D. Maier, Fundamentals of object-oriented databases, in: S.B. Zdonik, D. Maier (Eds.), *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990, pp. 1–32.



Michael Halper received the Ph.D. degree in computer science from NJIT. During his graduate studies, he was the recipient of a Garden State Graduate Fellowship from the State of New Jersey. Dr. Halper is a professor of computer science at Kean University, and a research associate at NJIT's Ontology and Medical Informatics Laboratory. His research interests include conceptual and object-oriented data modeling, part-whole modeling, extensible data models, object-oriented database (OODB) systems, and medical informatics (with an emphasis on medical terminologies). He has worked on the OOHVR project—funded by the National Institute of Standards and Technology (NIST) Advanced Technology Program—to model controlled terminologies using OODB technology. Dr. Halper is currently a Co-PI on a National Library of Medicine (NLM) grant that focuses on partitioning and abstraction techniques for auditing and extending the Unified Medical Language System (UMLS). Dr. Halper's work has also been supported by the New Jersey Commission on Science and Technology. He has had numerous papers in international journals, conferences, and workshops. Dr. Halper received Kean University's 2003 Presidential Excellence Award for Distinguished Scholarship. He is a member of The Honor Society of Phi Kappa Phi.



Li-min Liu is an assistant professor of the Department of Applied Mathematics at Chung Yuan Christian University, Taiwan. He received his Ph.D. in computer and information science (CIS) from New Jersey Institute of Technology (NJIT) in 1999; M.S. in CIS from Syracuse University in 1994; and B.S. in CIS from Tamkang University in Taiwan in 1990. Dr. Liu is also a research associate of the Ontology and Medical Informatics Lab of the CS Department at NJIT. His research interests include object-oriented (OO) database systems, OO modeling, knowledge representation, medical informatics, controlled terminologies, parallel databases, image processing, and Computer Assisted Language Learning (CALL).



James Geller received an Electrical Engineering Diploma from the Technical University, Vienna, Austria, in 1979, and the MS Degree (1984) and Ph.D. degree (1988) in Computer Science from the State University of New York at Buffalo. He joined NJIT in 1988, was granted tenure in 1993 and promoted to full professor in 2000. He is Director of the Semantic Web and Ontologies Lab at the CS Department and Associate Chair for Undergraduate Studies. Previously, James Geller was Co-director and Graduate Advisor of the Biomedical Informatics MS and Ph.D. Programs jointly administered with UMDNJ. He has published in numerous journals and conferences on topics in Artificial Intelligence, Knowledge Representation, Parallel Reasoning, Databases, Semantic Modeling in Object-Oriented Databases, Medical Informatics, Medical Vocabularies, and Auditing of Medical Terminologies. His work on Medical Vocabularies was funded by the National Institute of Standards and Technology with over one million dollars. His work on Web Mining for Marketing was supported by a five-year grant from the NJ Commission on Science and Technology. Recent topics of interest include Information Integration and E-Commerce.



Yehoshua Perl received his Ph.D. degree in Computer Science in 1975 from the Weizmann Institute of Science, Israel. He was lecturer and senior lecturer at the Bar-Ilan University, Israel during the years 1975–1985. From 1982 to 1985, he was visiting associate professor at Rutgers University (New Brunswick). Since 1985, he has been in the Computer Science Department at the New Jersey Institute of Technology (NJIT) where he was appointed professor in 1987. He received the Harlan Perlis Research Award of NJIT in 1996. Dr. Perl is the author of more than 100 papers in international journals and conferences. His publications are in object-oriented databases, medical informatics, medical terminologies, design and analysis of algorithms, sorting networks, graph theory, and data compression. From 1995 to 1999, Dr. Perl was involved in the large OOHVR (Object Oriented Healthcare Vocabulary Repository) project supported by the National Institute of Standards and Technology, ATP Program. Currently, he is working on a National Library of Medicine grant, “Partitioning to Support Auditing and Extending the UMLS.” Highlights of his research include: The shifting algorithm technique for tree

partitioning; analysis of interpolation search; the design of periodic sorting networks; partitioning, abstraction, and auditing of terminologies; and enhancing the semantics of OODBs.