Set Operations in Semantic Data Models

Elke A. Rundensteiner and Lubomir Bic

Department of Information and Computer Science
University of California, Irvine
June, 1989

Technical Report 89-22

Set Operations in Semantic Data Models

ELKE A. RUNDENSTEINER and LUBOMIR BIC

Department of Information and Computer Science

University of California, Irvine

June, 1989

Abstract

Class creation by set operations has largely been ignored in the literature. Precise semantics of set operations on complex objects require a clear distinction between the dual notions of a set and a type, both of which are present in a class. Our paper fills this gap by presenting a framework for executing set-theoretic operations on the class construct. The proposed set operations determine both the type description of the derived class as well as its set membership. For the former, we develop inheritance rules for property characteristics such as single- versus multi-valued and required versus optional. For the later, we borrow the object identity concept from data modeling research. Our framework allows for property inheritance among classes that are not necessarily is-a related.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.1 [**Database Management**]: Conceptual Design - *data models.* H.2.3 [**Database Management**]: Languages – *data description language*;

Additional Key Words and Phrases: Set Operations, Property Inheritance, Class Derivation, Complex Objects, Semantic Data Models, Data Modeling.

Contents

1 INTRODUCTION

Current trends in database research have developed numerous conceptual data models that attempt to capture real-world information in a natural and non-ambiguous manner. Examples are object-oriented [6, 10] and semantic database systems [1, 8, 12]. Common to these data models is the concept of a *class.* Most models support a rich class definition facility based on restricting inherited properties, that is, special forms of the specialization and generalization abstractions [11, 7]. On the other hand, the potential of set operations has largely been unexplored. The reason for that is that, while set operations on simple elements are well understood, precise semantics for set operations on complex objects have not yet been developed. Such definitions require a clear distinction between the dual notion of a class, which represents a *set* and also provides a *type* description. This distinction is usually blurred in the literature. Class derivation mechanisms commonly supported, such as the specialization abstraction, are *type-oriented;* they perform some operation on the type aspect of a class which then automatically implies a particular set relationship between the original and the derived class. Set operations work in a contrary manner. They perform some operation on the set-aspect of a class and the particular type relationships are implied. Such implied types, however, are not always correct, as we will show in this paper.

This paper presents a framework for executing set-theoretic operations on complex objects. We consider the four most common set operations - union, intersection, difference, and symmetric difference. Our approach is to extend set theory to the world of classes while preserving as much as possible of the well-known set-theoretic semantics. The specification of set operations on classes is based on our distinction between the set and type aspect of classes. In other approaches, these are commonly combined and treated as one relationship, the "is-a" relationship. Here, we first define the effect of a set operation on the membership of the corresponding class. For this, we borrow the concept of object identity from data modeling research. Second, the type description of the resulting class is specified. We design rules that describe how property characteristics are to be inherited

through these set operations. The rules accommodate property characteristics such as multi-valued versus single-valued and required versus optional.

Our treatment of set operations sheds some light on assumptions commonly made in the literature. The analysis of relationship types which hold between classes derived by set operations shows that the resulting class relationships are not necessarily "is-a" relationships. In fact, a class derived by a symmetric difference operation will never stand in any "is-a" relationship with its base classes. Consequently, our framework allows for the inheritance of properties between classes which are not "is-a" related. We know of no other data model which has this capability.

The paper is organized as follows. In Section 2, we review the basics of conventional set theory as needed for the remainder of the paper. Section 3 familiarizes the reader with conceptual data model terminology; special emphasis is placed on the distinction between subset/superset and subtype/supertype relationships of classes. In Section 4 we present definitions for set operations as well as rules for the inheritance of properties. Throughout this section we give pragmatic examples that support the usefulness of our framework. Results are presented in Section 5. Related research is discussed in Section 6, and conclusions are presented in Section 7.

2 SET THEORY

Set operations in conventional set theory are well understood. They assume simple objects and do not address typing and associated problems. We briefly survey the basics of set theory, since our goal is to preserve these well-accepted semantics of set operations when extending them to typed objects and classes.

A *set* S is a collection of objects where objects may be anything including symbols, physical objects or abstract concepts. Objects within a set S are referred to as *elements.* In set theory, all elements, be they complex like a Person or simple like an integer, are represented by one symbol. In other words, set theory considers all objects to be simple, not typed, and lacking any associated properties. We write 's $\in$ S' ('s $\notin$ S') to mean that the object s *is (not) an element* of the set S.

New sets can be formed from existing ones by the following operations: union, intersection, difference, and symmetric difference. In set theory, these operations determine the exact membership of the newly created sets. Let S1 and S2 be any two sets. The *difference* of S1 with respect to S2 is defined as S1 - S2 = $\{s|s \in S1$ and $s \notin S2\}$. The *union* of S1 and S2 is defined as S1 $\cup$ S2 = $\{s|s \in S2$ or $s \in S2$ or both $\}$. The *symmetric difference* of S1 and S2 is defined as S1 $\triangle$ S2 = $\{s|s \in S2$ or $s \in S2$ but not both $\}$. The *intersection* of S1 and S2 is defined as S1 $\cap$ S2 = $\{s|s \in S2$ and $s \in S2\}$.

A set S1 is defined to be a *subset* of the set S2, written S1 $\subseteq$ S2, if and only if every element of S1 is also an element of S2. Formally, S1 $\subseteq$ S2 iff (s $\in$ S1 $\Longrightarrow$ s $\in$ S2). The subset operation is not a mechanism to create a new set out of a given one, instead it only models a *relationship* between two sets. To actually create a subset, operators such as the set difference, union, etc., must be used. It is of course also possible to explicitly create a subset of a given set by selecting some of its elements and grouping them into a new set. Figure 1 lists the subset relationships between initial sets and the sets resulting from applying these set operators.

set operation	resulting subset relationships
intersection	(S1 $\cap$ S2 $\subseteq$ S1) and (S1 $\cap$ S2 $\subseteq$ S2)
union	(S1 $\subseteq$ S1 $\cup$ S2) and (S2 $\subseteq$ S1 $\cup$ S2)
difference	S1 - S2 $\subseteq$ S1
symmetric difference	none

Figure 1: Inferred Subset relationships.

In the remainder of this paper, we discuss how these operations can be generalized for dealing with classes and complex objects found in data modeling environments. In particular, we will study whether in this new context the set operations preserve the subset relationships as shown in figure 1.

3 TYPES VERSUS SETS

A key to the solution of well-defined set operations on complex objects is the explicit distinction between the type and set aspect of a class. Set operations on collections of untyped elements (mathematical sets) are well understood. Thus we have to study the effect of set operations on the type description of a class while preserving the semantics of the class's set notion. Below we introduce data modeling concepts and terminology needed for subsequent discussions.

3.1 Entities and Classes

An entity may be a simple or a complex object. An entity always consists of an identity and a state [9, 13]. The identity is a globally unique reference which is not visible to the user. Each time a new entity is created, an identity is assigned to it by the system. In this paper, we use the entity's identifier as entity reference. To indicate that we refer to the identifier as such rather than the entity itself, we use the notation <entity reference>.id. The state of an entity corresponds to a collection of one or more property names and associated values. We refer to the properties (attributes) of an entity by:

$< property\ name > (< entity\ reference >)$.

A class is formed by grouping together a collection of similar entities. Every class can be identified by its name, since class names are unique with respect to all other class names in the schema. The concept of a class serves a dual purpose: it does not only represents a collection of entities but it also provides a type description for all its members. The term type refers to the collection of properties associated with all entities that belong to that class. The set of properties of a class is specified by:

$< property\ name >:< domain > [< characteristics >]$;

The domain can be any user-defined class of the model or a predefined base domain like an integer range. Characteristics are general descriptions of properties, for instance,

whether a given property is required or optional. (More on this is presented later in this section.) An example of the notation is given next.

Example 1 *The class Person is defined by the following:*

class *Person with properties:*

Name : String [identifying, required];

Age : 0:100 [single-valued];

The following notation is used to refer to the properties of a class:

$< class\ name > . < property\ name >$.

We refer to the domain of a property by:

domain($< class\ name > . < property\ name >$).

Given the previous example, we refer to the Name property of the class Person by Person.Name and we refer to the domain of this property by **domain**(Person.Name). We use the following predicate to test whether a property is defined for a class or not:

$< class\ name > . < property\ name >$?.

For instance, in example 1 the predicate Person.Name? returns true because the Name property is defined for the class Person whereas the predicate Person.Friend? returns false.

We use the set-theoretic predicate 'e $\in$ C' to denote that the entity e is an *instance-of* the class C. The predicate 'e $\in$ C' is solely based on the object identity of e [13]. An entity may take on values for different sets of properties when viewed as member of different classes. For instance, a person will exhibit different characteristics when viewed as a spouse than as an employee. To refer to the properties of an entity as the participant in a particular class we use the following notation:

$< property\ name > (< entity\ reference >$ **as** $< class\ name >)$.

For example, we assign the value \$4,000 to the property Salary of the entity Jack in class Employee by "Salary(Jack **as** Employee) := \$4,000". The entity Jack does not have

a Salary property when viewed as a member of the Person class (by example 1), and thus the assignment "Salary(Jack **as** Person) := $4,000" is illegal.

A class is either a base class or a non-base class. A base class is defined independently of all other classes in the database. A non-base class is defined in terms of one or more classes of the database. There are numerous types of class creation abstractions for non-base classes, such as specialization/generalization abstractions [8], the aggregation abstraction, also called "part-of" relationship in object-oriented systems, and groupings found in semantic data models [7, 13]. The most common one is specialization, which is supported by virtually all conceptual database systems, notably, SDM [7], TAXIS [11], and IFO [1]. Specialization creates a non-base class by constraining the property description of an existing class (e.g., Red-Cars are defined as Cars with Color=Red), by specifying an additional property on the class (e.g., Grad-Students are defined as Students with the additional property Type-of-employment), or by explicitly collecting some elements to belong to it (e.g., Banned-Ships could be a class of Ships categorized by some criteria not known to the data model). Non-base classes can also be derived by means of set operations. This approach, which is much less common, is the main focus of the research presented in this paper.

3.2 Characteristics of Properties

Most object-oriented and semantic data models [7, 8] associate some or all of the following characteristics with each class property:

1. required versus optional;
2. identifying versus non-identifying; and
3. single- versus multi-valued.

If an entity is member of a class then it takes on some values (not equal 'undefined') for all properties defined for that class. The first characteristic distinguishes between required (mandatory) and non-required (optional) properties. Each entity of a class must

have a value for its required properties, i.e., a value not equal 'unknown'. It may or may not have a value for the optional ones, i.e., its value may be 'unknown'. Note that if a property is not defined for a class, then the value of that property would be 'undefined', meaning not applicable, instead of 'unknown'. This characteristic is redefinable, since what may be a mandatory property for some classes may be optional for others.

The identifying characteristic corresponds to the concept of a key in relational database theory. It is not as important in object-oriented data models since the underlying concept of object identities allows entities to be identified independently of their values. It can however still be used by human users who wish to maintain their own unique values as entity references.

A property is defined to be either single- or multi-valued independent of the class for which it is initially introduced. If a property p is single-valued then for any entity e of class C, p(e **as** C) has to be an element of **domain**(C.p) or be unknown. If a property p is multi-valued then for any entity e of class C, p(e **as** C) is a subset of **domain**(C.p) or is unknown.

The following simplistic convention guarantees that the names of all properties are unique throughout the entire schema: The name of a property is prefixed by the name of the class for which it is initially defined. Consequently, if a property is inherited from another class then its property name is prefixed by the name of the class in which it was originally defined. For instance, if the Employee class and the Administrator class both have a newly defined property called Salary, then the system refers to the Employee's property as Employee.Salary and the Administrator's property as Administrator.Salary.

3.3 Class Relationships

Most existing systems ignore the set/type duality of the class construct and hence cannot provide clean semantics for set operations on classes. In the following, we emphasize this dual notion by studying the meaning of class relationships which in the literature are generally referred to as subclass/superclass or "is-a" relationships. We disambiguate their meaning by distinguishing between two types of relationships:

- subset/superset and
- subtype/supertype relationships.

These two aspects of a class are not equivalent, i.e., a type relationship does not determine a set relationship, and vice versa. If an entity belongs to a class then this entity will necessarily be described by all properties of the type description of that class (which includes possibly 'unknown' values if some properties are characterized as optional). However, if an entity has all the properties of a class then this does not imply that the entity necessarily belongs to the class. Hence, properties are necessary but not sufficient conditions for a class membership.

The subset relationship between two classes is based on the identities of their entities. In particular, we have the following definition.

Definition 1 *The following set relationships can exist between two classes C1 and C2:*

1. *C1 is a* **subset** *of C2, denoted by* $C1 \subseteq C2$, *which is defined by* $C1 \subseteq C2 := (\forall\ e1)\ (\ e1 \in C1)\ ((\exists\ e2 \in C2)\ (e1.id = e2.id))$.

2. *C1 is a* **strict subset** *of C2, denoted by* $C1 \subset C2$, *as defined by* $C1 \subset C2 \Longleftrightarrow (C1 \subseteq C2$ *and* $((\exists\ e)\ (e \in C2$ *and* $NOT(\ e \in C1))))$.

3. *C1 is* **set equivalent** *to C2, denoted by* $C1 \equiv^s C2$, *as defined by* $C1 \equiv^s C2 \Longleftrightarrow (C1 \subseteq C2$ *and* $C2 \subseteq C1)$

4. *C1 is* **set inequivalent** *(set incompatible) with C2, denoted by* $C1 \not\equiv^s C2$, *as defined by* $C1 \not\equiv^s C2 \Longleftrightarrow (\ NOT(C1 \subseteq C2)$ *and* $NOT(C2 \subseteq C1))$.

This definition is based strictly on object identity. It disregards the type description associated with the respective classes. Hence, the class C1 may have *more*, *less*, or *the same* number of attributes as C2. For example, the class Students in figure 2 could be a subset of the class Employees (if every Student were working) - in spite of the fact that the elements of these two classes are described by different properties. This stands in

contrast to conventional set theory where elements of the sub- and superset always look alike [13].

The subtype/supertype relationship is concerned with the type description of classes and consequently with the state of all elements that participate in them. The *subtype* relationships between two classes is based on their type descriptions.

Definition 2 *The following type relationships can exist between two classes C1 and C2:*

1. *C1 is a* **subtype** *of C2, denoted by* $C1 \preceq C2$, *as defined by* $C1 \preceq C2 := (\forall\ p)$ (*C2.p?=true* $\Longrightarrow$ *C1.p?=true*) *and* (**domain**(*C1.p*) $\subseteq$ **domain**(*C2.p*)).

2. *C1 is a* **strict subtype** *of C2, denoted by* $C1 \prec C2$, *as defined by* $C1 \prec C2$, $\Longleftrightarrow$ ($C1 \preceq C2$ *and* $((\exists\ p)$ (*C2.p?=true and C1.p?=true and* (**domain**(*C1.p*) $\subset$ **domain**(*C2.p*))) *or* $((\exists\ p)$ (*C1.p?=true and NOT(C2.p?=true)*))))).

3. *C1 being* **type equivalent** *to C2, denoted by* $C1 \equiv^t C2$, *is defined by* $C1 \equiv^t C2$ $\Longleftrightarrow$ ($C1 \preceq C2$ *and* $C2 \preceq C1$).

4. *C1 is* **type inequivalent** *(type incompatible) with C2, denoted by* $C1 \not\equiv^t C2$, *which is defined by* $C1 \not\equiv^t C2 \Longleftrightarrow$ (*NOT*($C1 \preceq C2$) *and NOT*($C2 \preceq C1$)).

If C1 has more properties or more restricted domains than C2, then C1 is a strict subtype of C2 (#2). If two classes C1 and C2 have identical properties and domains, then they are equivalent types (#3). If there is no type relationship between two classes, i.e., C1 $\preceq$ C2 and C1 $\succeq$ C2 are both false, then we use the symbol C1 $\not\equiv^t$ C2 to denote their type incompatibility (#4).

A subtype has all properties of its supertype and optionally some additional ones. Hence, a subtype has either *more* or *the same* number of properties than its supertype. The domain of the subtype properties may be equal or contained in those of the corresponding properties of the supertype. For instance, the class Banned-Ships is a subtype of the Ships class and both have the same properties with the same domains. The type relation does not make any assumptions about the corresponding class memberships. Hence, theoretically,

a subtype could have *more, less,* or *the same* number of elements as its supertype, or their instances may even be totally unrelated. The set of given class derivations and their semantics ultimately determine how type and set relationships interact within a given data model as will be shown in Section 4.

The term *is-a* relationship has been misused to mean many different things [3]. We can now define the is-a relationship in terms of the two just defined class relationships.

Definition 3 *C1 is-a C2* $\Longleftrightarrow$ *C1* $\preceq$ *C2 and C1* $\subseteq$ *C2.*

Informally, we say that C1 is-a C2 if (1) every instance of C1 is an instance of C2 (the subset relationship) and (2) every property defined for C2 is also defined for C1 (the type relationship) [11]. More precisely, if C1 is-a C2 and <C2.p? = true> and <**domain**(C2.p) := C2'> then C1 has the same property p, i.e., <C1.p? = true>, and < **domain**(C1.p) := C1'> with C1' is-a C2'.

4 SET OPERATIONS IN CONCEPTUAL DATA MODELS

4.1 Introduction

This section discusses how the set operations can be applied in conceptual data models. As mentioned earlier, set operations in conventional set theory are well understood, since the underlying objects are simple, i.e., are not typed and don't have any associated properties. Thus, typing and related problems such as the inheritance of properties are not addressed in set theory. When dealing with conceptual data models, typing becomes an issue. An important distinction between sets in set theory and classes in data modeling is that a set represents a collection of simple elements whereas a class represents a collection of complex entities. In addition, it also provides their type description. Consequently, a well-defined set-theoretic operation on a class must specify the effect on both the type description and the resulting membership of that class.

The membership of a class derived by a set operation is based on the object identities of the involved entities [13]. The resulting type description, however, is at large based

on the properties defined for the original classes. The latter has no correspondence in conventional set theory, where a set is completely described by enumerating its members. Consequently, there is nothing in set theory to dictate the treatment of the type description of the resulting class. Other data models [7, 17] have made certain (arbitrary) choices in this regard without giving a convincing argument to support their choice. The often raised point that an entity has a certain property and hence this property has to be reflected in the type description of the class it belongs to is not well founded. First, properties can be optional and, second, when viewed as member of a class the entity may grant access to only some of its properties as described in Section 3.

Determinating the type description of classes derived by set operations is related to the issue of property inheritance. The difference being that the inheritance of properties usually takes place between *two* classes while set operations always deal with three classes. In other words, set operations are similar to the problem of multiple inheritance. However, the literature assumes that the inheritance of properties takes place between classes which stand in an is-a relationship to one another; this is not necessarily the case for classes derived by set operations. As will be illustrated in Section 5, the latter assumption appears to be the reason for the limited type of set operations found in the literature. Below, we address the property inheritance problem by determining what characteristics properties of a derived class should have – once inherited. This leads to the development of general rules for property inheritance.

The following two definitions are given to simplify the remainder of this section. They are based on the naming convention given in Section 3.2, which guarantees the uniqueness of property values: If two classes define the same single-valued property, then an entity that appears in both cannot have two distinct values for it. The property value is either unknown in one of them or the two values are identical.

Definition 4 *Let p be a property. Let e be a member of C1 and/or C2. Then the operation COMBINE is defined as follows.*

If p is a single-valued property we have

COMBINE(p(e **as** *C1), p(e* **as** *C2)) :=*

$$\begin{cases} \text{p(e as C1)} & \text{if e} \in \text{C1 and C1.p? and p(e as C1)} \neq \text{unknown} \\ \text{p(e as C2)} & \text{if e} \in \text{C2 and C2.p? and p(e as C2)} \neq \text{unknown} \\ \text{unknown} & \text{otherwise} \end{cases}$$

If p is a multi-valued property then COMBINE is defined as follows

COMBINE(p(e **as** *C1), p(e* **as** *C2)) :=*

$$\begin{cases} \text{p(e as C1)} \cup \text{p(e as C2)} & \text{if e} \in \text{C1 and C1.p? and p(e as C1)} \neq \text{unknown and} \\ & \text{e} \in \text{C2 and C2.p? and p(e as C2)} \neq \text{unknown} \\ \text{p(e as C1)} & \text{if e} \in \text{C1 and C1.p? and p(e as C1)} \neq \text{unknown and} \\ & (\text{e} \notin \text{C2 or NOT(C2.p?) or p(e as C2)} = \text{unknown}) \\ \text{p(e as C2)} & \text{if e} \in \text{C2 and C2.p? and p(e as C2)} \neq \text{unknown and} \\ & (\text{e} \notin \text{C1 or NOT(C1.p?) or p(e as C1)} = \text{unknown}) \\ \text{unknown} & \text{otherwise} \end{cases}$$

It is understood that the union operation ($\cup$) removes all duplicate values.

The previous definition describes how a value is to be combined if it is inherited from more than one source - assuming the naming convention described in section 3.2. Next, operations on type descriptions are introduced.

Definition 5 *Let C1 and C2 be two classes.*

Then, C1 $\bigvee$ C2 := { p | C1.p? or C2.p? }.

And, C1 $\bigwedge$ C2 := { p | C1.p? and C2.p? }.

Intuitively, C1 $\bigvee$ C2 denotes the collection of all properties defined for either C1 or C2. C1 $\bigwedge$ C2, on the other hand, consists of all properties common to the type description of both classes. Therefore we refer to the first operation as *collecting* and to the second as *extracting.*

In figure 2 we present a template of a simple conceptual data model. It will be used for subsequent examples to show the result of the inherited type description as well as the

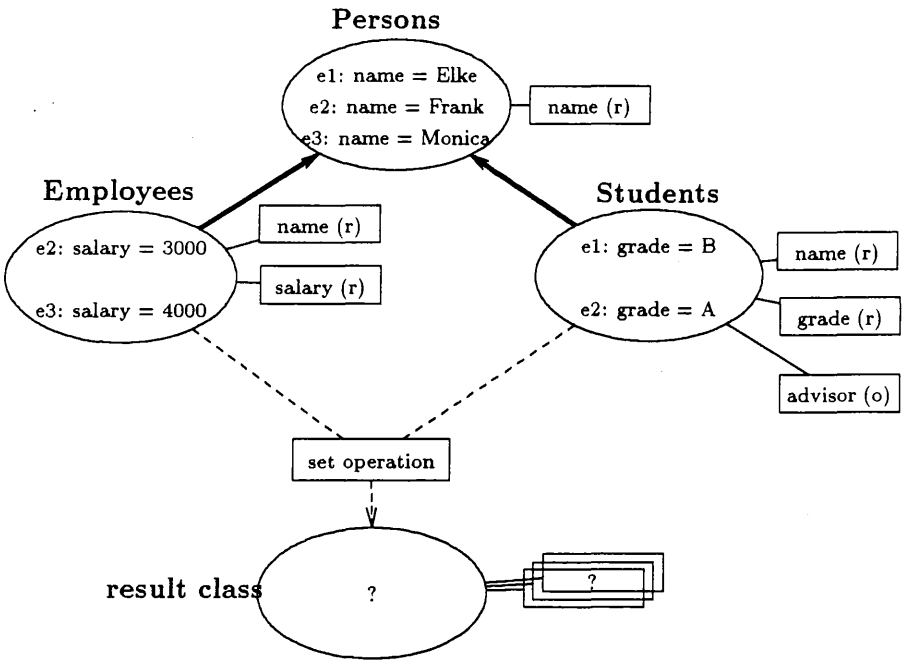

Figure 2: Template of a Derived Class.

membership of the derived class for each set operation. The examples are given to show the usefulness of the proposed procedures for the propagation of characteristics.

Example 2 *Figure 2 depicts the classes Employees and Students. Both classes are subclasses (by is-a relationship) of the Person class, which is depicted by solid dark arrows. The class Employees has two required properties, the inherited property Name and the newly defined property Salary. The class Students has two required and one optional property, namely, the inherited property Name, and the newly defined properties Grade and Advisor. The class containing the question mark represents the result of performing a set operation on the classes Employees and Students. Subsequent examples will instantiate the generic set operation to one of the possible four choices and show the corresponding result class.*

4.2 Difference Operations

We propose two types of difference operations. The first is derived automatically by the system while the second is determined by the user by specifying the desired type description. The property values of all entities included in the newly derived class will automatically be calculated once the type description of the new class has been established. This is correct for all set-theoretic operations and parallels the situation of set operations in conventional set theory.

Definition 6 *Let P be a collection of properties defined by* $P := \{p|\ C1.p? = true\}$. *The* **(automatic) difference** *of C1 with respect to C2, denoted by* $C1 \dot{-} C2$, *is defined by*

$C1 \dot{-} C2 := \{e|e \in C1$ *and* $e \notin C2\ \}$

with $(\forall\ p \in P)\ (\ (C1 \dot{-} C2).p? := true\ )$

and $(\forall\ e \in C1 \dot{-} C2)\ (\forall\ p \in P)\ (\ p(e$ **as** $C1 \dot{-} C2) := p(e$ **as** $C1)\ )$.

Definition 7 *Let P be as in the previous definition. The* **user-specified difference** *operation of C1 with respect to C2, denoted by* $C1 \tilde{-} C2$, *is specified by giving some* $Q \subseteq P$. *It is defined like the automatic difference except for replacing P with Q.*

There is an important difference between the just presented user-specified set operation and the user-specifiable subclass mechanism commonly found in the literature [7]. Here, the user has to specify the type description once - namely during the creation of the derived class. Thereafter, the class can be instantiated automatically by the system according to the semantics of the applied set difference operation. Therefore, the level of user involvement is minimal. This contrasts strongly with the user-specified subclass mechanism where the user has to explicitly insert all entities into the class. The user-specified set operations can be classified as automatic class derivation mechanisms.

Given these set definitions, let us now study rules for the inheritance of property characteristics. We distinguish between single- and multi-valued properties. As described in section 3.2, this characteristic once defined is fixed throughout the data model and thus does not change when a property is inherited by another class.

The second characteristic determines whether a property is identifying or not. The following simple rule is sufficient to describe the propagation of this characteristic from the base classes to the derived class.

Rule 1 *Let P1 and P2 be sets of identifying properties for C1 and C2, respectively. If C is a class resulting from the difference of C1 relative to C2 as defined by definitions 6 and 7 then P1 will be identifying for C if inherited.*

The previous rule is self-explanatory. The result class will contain only entities from the class C1. Hence if a set of properties P1 is sufficient to distinguish between all entities of C1 then it will also be sufficient to distinguish between the ones of a subset of C1, i.e., the difference class. Next we address the third characteristic which determines whether a property is required or non-required for a class.

C1	C2	C1 − C2
optional	-	optional
required	-	required
-	optional	-
-	required	-
optional	required	optional
required	optional	required
optional	optional	optional
required	required	required

Figure 3: Inheritance of Property Characteristics for a derived Difference class.

Rule 2 *The table in figure 3 lists the propagation rules for the inheritance of the required/optional characteristics of a class derived by a difference operation.*

The table is to be read as follows. The symbol "required" refers to a required property, "optional" refers to a not required (but existing) property, and "-" means that the particular property is not defined for that class. The third column gives the characteristic of the inherited property in the derived class C1 $\tilde{-}$ C2 or C1 $\check{-}$ C2, based on the characteristics of the corresponding property in C1 and C2 (first and second column). The difference

operations are not symmetric and hence the figure contains entries for all possible combinations. Every member of a difference class is also a member of C1. Therefore, all properties of C1 as well as their characteristics can be directly inherited by C1 $\tilde{-}$ C2 or C1 $\check{-}$ C2. This explains why the third column of the table in figure 3 is an exact copy of the first column.

Example 3 *The difference operation C1 $\check{-}$ C2 defined in definition 6 is applied to the conceptual data model shown in figure 2. The result, shown in figure 4, is a class that consists of all Persons who are Employees but not Students. Hence, all its elements are also members of the Employees class. Consequently, properties required for the Employees class , in this case, Name and Salary, are also required for the difference class. If the Employees class had optional properties, then those would still be optional for the newly created class. Properties of the Students class are not relevant to the resulting class since no entity of the result class would have any value for them. Hence, we do not include them in the resulting type description, and thus choose the difference operation C1 $\check{-}$ C2 as defined in definition 6 over the operation C1 $\tilde{-}$ C2.*

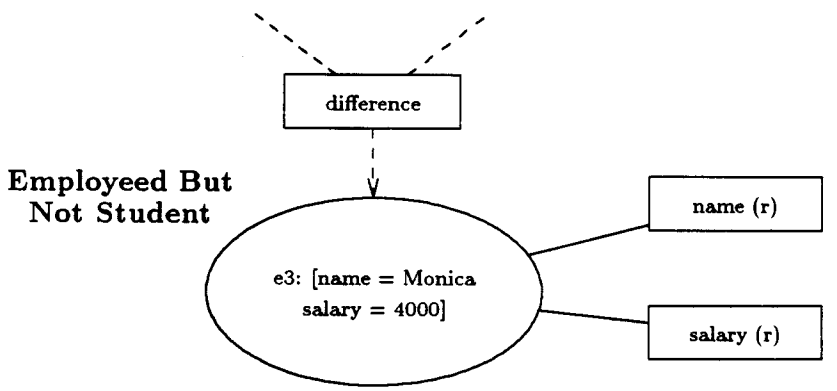

Figure 4: Derived class created by the difference operation.

4.3 Union Operations

Next, we distinguish three types of union operations. The type descriptions of first two are derived automatically by the system, whereas the third one is determined by the user.

Definition 8 *Let P be C1 $\bigvee$ C2. The first union operation of C1 and C2, denoted by C1 $\check{\cup}$ C2, is called the* **collecting union**. *It is defined by*

C1 $\check{\cup}$ C2 := $\{e | e \in C2$ or $e \in C2$ or both $\}$

with ($\forall$ p $\in$ P) ((C1 $\check{\cup}$ C2).p? := true)

and ($\forall$ e $\in$ C1$\check{\cup}$C2) ($\forall$ p $\in$ P) (p(e **as** *C1$\check{\cup}$C2) := COMBINE(p(e* **as** *C1),p(e* **as** *C2)).*

Definition 9 *The second type of union operation of C1 and C2, denoted by C1 $\hat{\cup}$ C2, is called the* **extracting union**. *The extracting union is defined as in the previous definition except for replacing P with P := C1 $\bigwedge$ C2.*

Definition 10 *The user-specified union operation of C1 and C2, denoted by C1 $\tilde{\cup}$ C2, is defined by specifying a collection of properties Q with Q $\subseteq$ C1 $\bigvee$ C2. The definition of C1 $\tilde{\cup}$ C2 is equivalent to the one in Definition 8 with Q substituted for the symbol P.*

Note here that $\tilde{\cup}$ contains the other two union operations as special cases, since the user could choose the properties in the two cases as automatically derived by the system for $\check{\cup}$ or $\hat{\cup}$.

Next, we study the general rules for property inheritance. As described in section 3.2, the data model distinguishes between single- and multi-valued properties. This characteristic does not change when a property is inherited. Consequently, the rule of inheritance for this characteristic is trivial.

The second type of characteristic is whether a property is identifying or not. The following rule describes the propagation of this characteristic from the base classes to the derived class.

Rule 3 *Let P1 be a set of identifying properties for C1 and P2 a set of identifying properties for C2. If C is a class resulting from the union of C1 and C2 (any of the three union types) then P1 together with P2 will be identifying for C if both are inherited.*

The rule can best be explained with an example. Assume a situation similar to figure 2 where the entities of the class Students are identified by the property Student-Id and the Employee entities by the property Employee-Id. Then, in the collection of Employees and Students each individual Student entity can be distinguished from all other Student entities by its Student-Id and from Employees entities by not having an Employee-Id. The converse is true for all Employee entities. If both properties are not inherited then some of the entities in the derived class may be indistinguishable to the user but the system is still able to distinguish them based on their object identities. This is so because object identities are globally unique identifiers maintained by the system [13]. The example shows that if a database user intends to use some property as unique identifier then he has to declare it as a required property. Furthermore, he has to use set operations that propagate this property to the derived class. Thus, we establish the rule here that an *identifying property must always be a required property.*

C1	C2	C1 ∪ C2
optional	-	optional
required	-	optional
optional	optional	optional
required	optional	optional
required	required	required

Figure 5: Inheritance of Property Characteristics for a derived Union class.

Rule 4 *The table in Figure 5 lists the general inheritance rules for the required/optional characteristic when deriving a class by one of the three union operations.*

Note that a property can only be required if it has been required for both base classes. In all other cases, it cannot be guaranteed that all entities will take on a value for a

property and hence they can only be asserted as optional. Next, an example is given to demonstrate this inheritance mechanism.

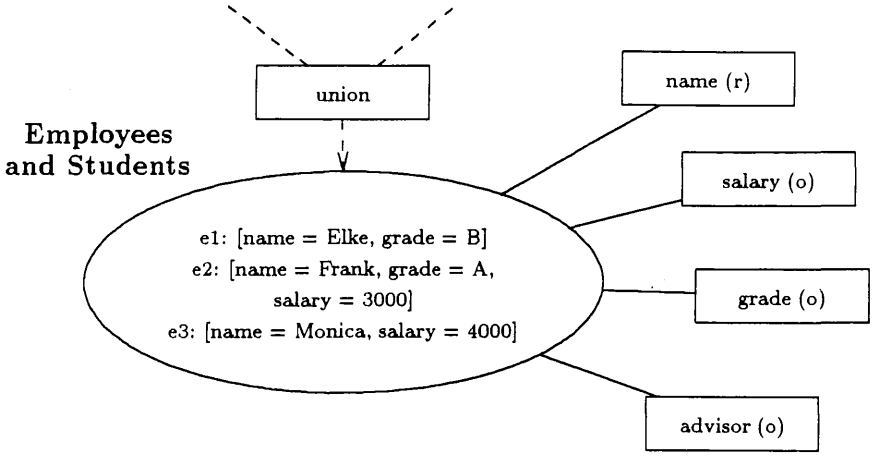

Figure 6: derived class created by union

Example 4 *The union operation C1 ∪̌ C2 defined in definition 8 has been applied to the situation in figure 2; the result is shown in figure 6. The new class consists of all those elements which were members of either the Employees or the Students class. Properties that were required for both classes can still be required for the resulting class, e.g., the inherited property Name. However, properties that were required for only one of these two classes can no longer be required. They can at best be optional in the new class. This is so since, for example, the person e1 does not have a value for the Salary property, even though the Salary property is required for the Employee class. The optional attributes must stay optional.*

4.4 Intersection Operations

Next we propose three types of intersection operations similar to the three unions. The first two again are automatically derived by the system. The third one is user-specified. The latter contains the other two as special cases.

Definition 11 *Let P be C1 $\bigvee$ C2. The intersection of C1 and C2, denoted by C1 $\check{\cap}$ C2, is called the* **collecting intersection.** *It is defined by*

C1 $\check{\cap}$ C2 := $\{e | e \in C2$ and $e \in C2\}$

with ($\forall$ p $\in$ P) ((C1 $\check{\cap}$ C2).p? := true)

and ($\forall$ e $\in$ C1$\check{\cap}$C2) ($\forall$ p $\in$ P) (p(e **as** *C1$\check{\cap}$C2) := COMBINE(p(e* **as** *C1), p(e* **as** *C2)).*

Recall that if a property p is defined for both classes C1 and C2 then an entity which takes on values for p in both classes will have the same value in both cases.

Definition 12 *Let P now be C1 $\bigwedge$ C2. The second type of intersection of C1 and C2, denoted by C1 $\hat{\cap}$ C2, is defined by as in definition 11 with the new meaning for P. It is called the* **extracting intersection.**

Definition 13 *The* **user-specified intersection** *operation of C1 and C2, denoted by C1 $\tilde{\cap}$ C2, is defined by specifying a collection of properties Q with Q $\subseteq$ C1 $\bigvee$ C2. The definition of C1 $\tilde{\cap}$ C2 is equivalent to the one in Definition 11 with Q substituted for the symbol P.*

The effect of the intersection operation on the characteristics of properties is evaluated next. Again, the single- and multi-valued property is fixed throughout the data model and therefore does not change when a property is inherited (Section 3.2).

The identifying characteristic can be propagated from the base classes to the derived class by the following rule:

Rule 5 *Let P1 and P2 be sets of identifying properties for C1 and C2, respectively. If C is a class resulting from any of the three just defined intersection operations of C1 and C2 then P1 or P2 will be identifying for C if inherited.*

Again, the rule is self-explanatory. The result class will be a subset of both C1 and C2. Thus, if P1 is sufficient to distinguish between the entities of C1 then it is sufficient to uniquely identify them when they appear in a subset of C1, i.e., the derived class. The same is true for P2.

Rule 6 *Figure 7 gives the inheritance rules for the required/optional characteristic of properties defined for an intersection class.*

C1	C2	C1 ∩ C2
optional	-	optional
required	-	required
optional	optional	optional
required	optional	required
required	required	required

Figure 7: Property Inheritance of derived Intersection class.

To summarize, a property can be required for the result class if and only if it is defined for both base classes and if it is required for at least one of them. In all other cases, the property can only be asserted to be optional. The following example uses the previous definition for property inheritance.

Example 5 *In figure 8 the intersection operation C1 ∩̌ C2 of definition 12 has been applied to the situation in figure 2. The new class consists of all Persons who are Employees and Students at the same time. Properties that were required for either of the two classes are also required for the resulting class, i.e., the inherited properties Name, Salary and Grade. This is a sensible rule since all members of the intersection class will be guaranteed to take on values for these properties. The optional attribute Advisor can still be optional.*

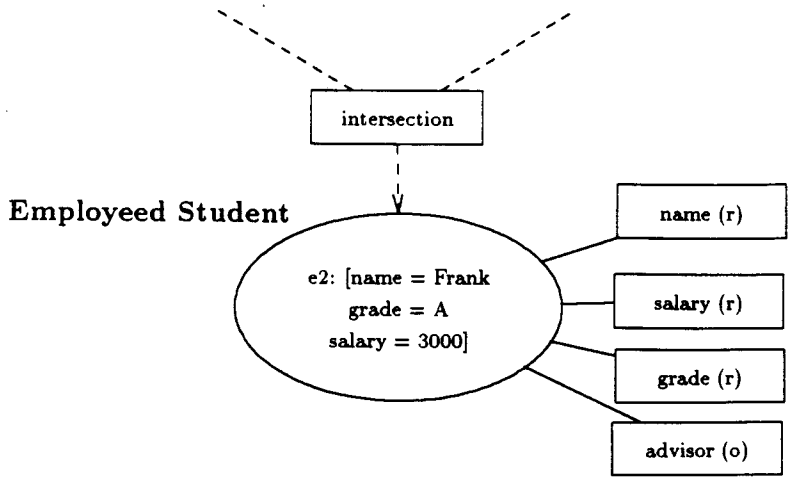

Figure 8: derived class created by intersection

4.5 Symmetric Difference Operations

Next we introduce three types of symmetric difference operations. In all three cases, the entities of the result class will come from exactly one of the two base classes. Consequently, no merging of properties by means of the COMBINE operation is needed - not even for multi-valued properties.

Definition 14 *Let P be C1 $\bigvee$ C2. The symmetric difference of C1 and C2, C1 $\check{\Delta}$ C2, is called the* **collecting symmetric difference**. *It is defined by*

C1 $\check{\Delta}$ C2 := {e|e $\in$ C1 or e $\in$ C2 but not both }

with ($\forall$ p $\in$ P) ((C1 $\check{\Delta}$ C2).p? := true)

and ($\forall$ e $\in$ C1 $\check{\Delta}$ C2) ($\forall$ p $\in$ P)

((e $\in$ C1 and C1.p? $\Longrightarrow$ p(e **as** *C1 $\check{\Delta}$ C2) := p(e* **as** *C1)) and*

(e $\in$ C2 and C2.p? $\Longrightarrow$ p(e **as** *C1 $\check{\Delta}$ C2) := p(e* **as** *C2)) and*

(((e $\in$ C1 and NOT(C1.p?)) or (e $\in$ C2 and NOT(C2.p?))) $\Longrightarrow$ p(e **as** *C1 $\check{\Delta}$ C2) := unknown).*

Note that in the previous definition for all entities e of the result class at most one of the three predicates will be true since either e $\in$ C1 or e $\in$ C2 but not both.

Definition 15 *Let P be C1 $\wedge$ C2. The symmetric difference of C1 and C2, C1 $\hat{\triangle}$ C2, is called* **extracting**. *It is defined by*

C1 $\hat{\triangle}$ C2 := $\{e | e \in C1$ or $e \in C2$ but not both $\}$

with ($\forall$ p $\in$ P) ((C1 $\hat{\triangle}$ C2).p? := true)

and ($\forall$ e $\in$ C1 $\hat{\triangle}$ C2) ($\forall$ p $\in$ P)

((e $\in$ C1 $\Longrightarrow$ p(e **as** *C1 $\hat{\triangle}$ C2) := p(e* **as** *C1)) and*

(e $\in$ C2 $\Longrightarrow$ p(e **as** *C1 $\hat{\triangle}$ C2) := p(e* **as** *C2))).*

In the previous definition, the properties in P are defined for both C1 and C2 and hence do not have to be tested.

Definition 16 *The* **user-specified symmetric difference** *operation of C1 and C2, denoted by C1 $\tilde{\triangle}$ C2, is defined by specifying a collection of properties Q with Q $\subseteq$ C1 $\vee$ C2. The definition of C1 $\tilde{\triangle}$ C2 is equivalent to the one in Definition 14 with the symbol P replaced by Q.*

For the same reasons mentioned earlier, the user-specified symmetric difference operation contains the other two automatic symmetric difference operations as special cases.

As described in section 3.2, the single- and multi-valued property characteristic is fixed throughout the data model. Therefore, when a property is inherited it simply keeps its characteristic.

The second type of characteristic is whether a property is identifying or not. The following rule is sufficient to describe the propagation of this characteristic from the base classes to the derived class.

Rule 7 *Let P1 and P2 be sets of identifying properties for C1 and C2, respectively. If C is a class resulting from a symmetric difference of C1 and C2, then P1 together with P2 will be identifying for C if both are inherited by C.*

The rule of inheritance described in the previous rule can be justified by an argument similar to the one given for the union operation (rule 5).

C1	C2	C1 △ C2
optional	-	optional
required	-	optional
optional	optional	optional
required	optional	optional
required	required	required

Figure 9: Inheritance of Property Characteristics for a derived Symmetric Difference class.

Rule 8 *The inheritance of property characteristics for a derived symmetric difference class is defined by the rules described in the table of figure 9.*

Again, a property can only be required in the new class if it has been required for both classes. In all other cases, it cannot be guaranteed that all entities of the result class will take on values for these properties. This is shown in the next example.

Example 6 *The symmetric difference operation C1 $\check{\triangle}$ C2 of definition 14 is used in figure 10. It results in a derived class that consists of all Persons who are Employees but not Students or Students but not Employees. Only properties which were required for both classes are required by rule 8. All others are optional. This is so since members of the result class will be exactly of one of the two types. For instance, the Salary property required for the entities of the Employees class could not be required in the result class since Student members of the latter would not have a value for it, and vice versa. The person e1 in figure 10 does, for example, not have a value for the Salary property.*

4.6 Other Examples of Set-theoretic Operations

Some more examples of set-theoretic operations on classes defined in this section are presented next.

Example 7 *Assume the situation depicted in figure 2. Then, the extracting intersection operation, i.e., Employees $\hat{\cap}$ Students, results in a class with the type description "Employed-Students.Name [required]" and the content { e2: [Name = Frank] }.*

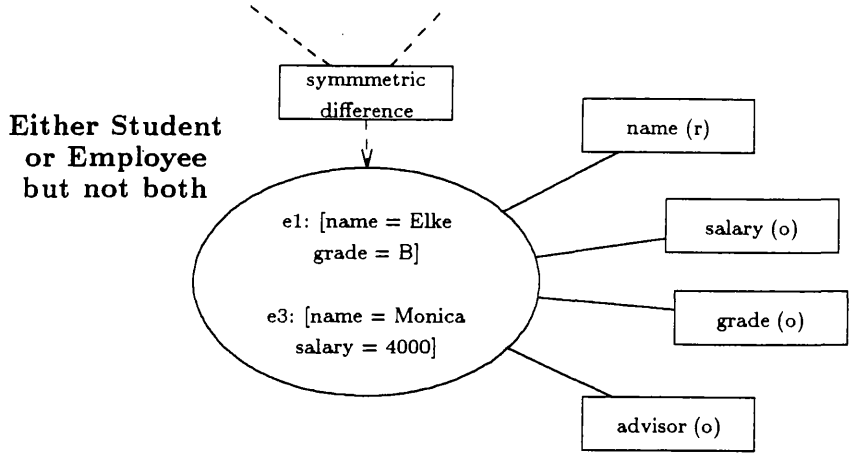

Figure 10: Derived class created by the Symmetric Difference operation.

The corresponding union operation, Employees $\hat{\cup}$ Students, results in a class with the same type description and the content { e1: [Name = Elke]; e2: [Name = Frank]; e3: [Name = Monica] }.

Example 8 *Assume the database designer is interested in Students who are employeed but still are good students. In this case, the user may not be interested in details of their employment status but to estimate how good a student is, the Grade property would be relevant. Hence, the user-specified operation Employees $\tilde{\cap}$ Students is the appropriate choice for creating the desired class. The chosen type specification is "Name" and "Grade". Then, the derived class has the content { e2: [Name = Frank, Grade = A] }.*

5 SET OPERATIONS AND CLASS RELATIONSHIPS

In this section we investigate what combinations of subset and subtype relationships could occur within a well-defined data model. This analysis shows the consequences of performing a set operation: both, sets and types of the new and old classes obey certain

relationships as shown below. We distinguish between several cases of user-specified set operations depending on the choice of the desired type description. The different choices for the type description of the resulting class, denoted by Q in definitions 10, 13, and 16, result in distinct class relationships. Let $\check{P}$ = C1 $\bigvee$ C2 and $\hat{P}$ = C1 $\bigwedge$ C2. Let P1 = $\{p|C1.p?\}$ and P2 = $\{p|C2.p?\}$. Recall that always Q $\subseteq \check{P}$. Then the choices for Q are listed below and are also shown in figure 11:

- case 1: P1 $\subseteq$ Q and P2 $\subseteq$ Q [$\Longrightarrow \check{P}$ = Q];
- case 2: P1 $\subseteq$ Q and NOT(P2 $\subseteq$ Q);
- case 3: P1 $\supseteq$ Q and NOT(P2 $\supseteq$ Q);
- case 4: P1 $\supseteq$ Q and P2 $\supseteq$ Q [$\Longrightarrow$ Q $\subseteq \hat{P}$];
- case 5: P1 $\not\equiv^t$ Q and P2 $\not\equiv^t$ Q.

Case 1 models the situation of the first type of an automatic set operation (the collecting type) which collects all properties of C1 and C2. In case 2, Q contains all properties defined for C1 and possibly some (but not all) properties defined for C2. The three set operations (excluding the difference operation) are symmetric and hence the analogous situation obtained by exchanging C1 and C2 is also covered by case 2. Q of case 3 contains a subset of P1 and at least one of its elements is not in P2. Again, the converse situation is also included in this case, since C1 and C2 can by exchanged. Case 4 includes Q = $\hat{P}$, i.e., the second type of an automatic set operation which extracts all properties common to both C1 and C2 as a special case. Case 5 illustrates the case where there is no set relationship between P1 and Q (and P2 and Q). In order for that to occur, Q must contain some (but not all) elements of P1 which are not in P2, and some (but not all) elements of P2 which are not in P1.

In figure 12 we list the set and type relationships which hold between the two classes C1 and C2 and the class derived by applying a set operation on them. (We use the symbol R as abbreviation for the result class of the set operation.) In the fifth column,

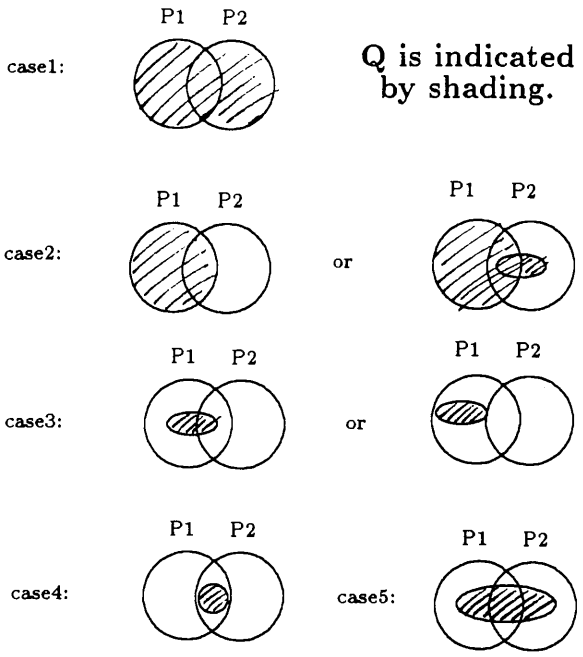

Figure 11: Five cases of type descriptions Q.

an asterisk indicates cases that contradict the requirements of an "is-a" relationship. In other words, each such row models a situation that could not appear in a database built by applying only specialization and generalization operations. Oour framework, on the other hand, allows for the inheritance of properties between classes which are not "is-a" related. We know of no other data model that has this capability.

This table shows clearly why the union definition of row 4 was chosen over the one in row 3 in the literature. Similarly, it shows why the intersection definition of row 10 was chosen over the one in row 11 in the literature. The reason is that rows 4 and 10 result in "is-a" relationships whereas the others don't. No violation of the "is-a" relationship can occur in the fourth block of figure 12 since no type relationships hold. The complexity and diversity of the resulting relationships may partly be the reason for the lack of "user-specified" set operations in the literature.

#	set operation	set relationships	type relationships	
1	R= C1 $-$ C2	R $\subseteq$ C1 and R $\not\equiv^s$ C2	R $\equiv^t$ C1 and R $\equiv^t$ C2	
2	R= C1 $\tilde{-}$ C2	R $\subseteq$ C1 and R $\not\equiv^s$ C2	R $\succeq$ C1 and R $\succeq$ C2	
3	R= C1 $\check{\cup}$ C2	R $\supseteq$ C1 and R $\supseteq$ C2	R $\preceq$ C1 and R $\preceq$ C2	*
4	R= C1 $\hat{\cup}$ C2	R $\supseteq$ C1 and R $\supseteq$ C2	R $\succeq$ C1 and R $\succeq$ C2	
5	R= C1 $\tilde{\cup}$ C2	R $\supseteq$ C1 and R $\supseteq$ C2	case 1: R $\preceq$ C1 and R $\preceq$ C2	*
6		" "	case 2: R $\preceq$ C1 and R $\not\equiv^t$ C2	*
7		" "	case 3: R $\succeq$ C1 and R $\not\equiv^t$ C2	
8		" "	case 4: R $\succeq$ C1 and R $\succeq$ C2	
9		" "	case 5: R $\not\equiv^t$ C1 and R $\not\equiv^t$ C2	
10	R= C1 $\check{\cap}$ C2	R $\subseteq$ C1 and R $\subseteq$ C2	R $\preceq$ C1 and R $\preceq$ C2	
11	R= C1 $\hat{\cap}$ C2	R $\subseteq$ C1 and R $\subseteq$ C2	R $\succeq$ C1 and R $\succeq$ C2	*
12	R= C1 $\tilde{\cap}$ C2	R $\subseteq$ C1 and R $\subseteq$ C2	case 1: R $\preceq$ C1 and R $\preceq$ C2	
13		" "	case 2: R $\preceq$ C1 and R $\not\equiv^t$ C2	
14		" "	case 3: R $\succeq$ C1 and R $\not\equiv^t$ C2	*
15		" "	case 4: R $\succeq$ C1 and R $\succeq$ C2	*
16		" "	case 5: R $\not\equiv^t$ C1 and R $\not\equiv^t$ C2	
17	R= C1 $\check{\Delta}$ C2	R $\not\equiv^s$ C1 and R $\not\equiv^s$ C2	R $\preceq$ C1 and R $\preceq$ C2	
18	R= C1 $\hat{\Delta}$ C2	R $\not\equiv^s$ C1 and R $\not\equiv^s$ C2	R $\succeq$ C1 and R $\succeq$ C2	
19	R= C1 $\tilde{\Delta}$ C2	R $\not\equiv^s$ C1 and R $\not\equiv^s$ C2	case 1: R $\preceq$ C1 and R $\preceq$ C2	
20		" "	case 2: R $\preceq$ C1 and R $\not\equiv^t$ C2	
21		" "	case 3: R $\succeq$ C1 and R $\not\equiv^t$ C2	
22		" "	case 4: R $\succeq$ C1 and R $\succeq$ C2	
23		" "	case 5: R $\not\equiv^t$ C1 and R $\not\equiv^t$ C2	

Figure 12: Set-theoretic operations and resulting set and type relationships.

Class relationships resulting from set operations (as shown in figure 12) are comparable with those in set theory. This comparison shows that the subset relationships between base classes and the classes derived by set operations are identical to those that hold between base sets and derived sets in set theory. In other words, the semantics of set operations have been preserved. Moreover, the type description associated with the resulting classes does not have any effect on the resulting set relationships. These results are presented graphically in figure 13. The dotted lines in figure 13 indicate the derivation of the

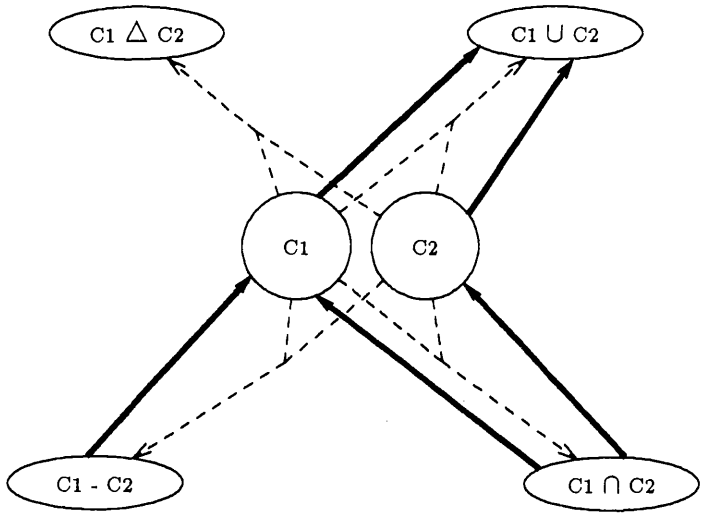

Figure 13: Set relationships of derived classes

new classes from C1 and C2. The set operation symbols, like ∪, are used as generic operators, representing all types of the corresponding operations proposed in this paper. Subset relationships are denoted by solid dark directed arcs with the arrow pointing to the superset.

We wish to emphasize that the use of set operations for class creation results in class relationships that would not exist in a database schema build solely by specialization and generalization abstractions.

6 RELATED RESEARCH

Some data models, in particular, most object-oriented models, define only type-oriented class operations. We use the term type-oriented to mean that these operations are applied

to the type description of the original class and that the resulting membership of the new class is derived automatically. Others [11, 1] also include some limited repertoire of set operations on classes. Most of these approaches are, however, ad-hoc, as discussed below.

Hammer and McLeod [7] were among the first to propose different types of derivation mechanisms for subclasses. In SDM, they list four subclass connections, namely, attribute-defined, user-controllable, set-operator defined, and existence subclasses. However, their approach towards set-operator defined classes is limited in as much as only one of all possible interpretations is chosen for each set operation. Our presentation in section 5 makes it apparent that the ones preserving "is-a" relationships were selected. The set operations considered in SDM [7] are union, intersection, and difference. Our analysis in Section 5 also explains why the symmetric difference operation has not been utilized as a class derivation mechanism: it never results in a "is-a" relationship, as shown in figure 12. Most other existing data models [10] do not even consider the use of set operations.

SAM* by Su [17] consists of seven different abstractions, referred to as association types, that construct new classes out of existing ones. One abstraction, called generalization, creates a more general concept type out of existing ones. This generalization corresponds to a form of union operation, however, it is not clear how the type description of the resulting concept type is formed. This ambiguity arises from the fact that SAM* is value- rather than object-based. No set operations other than this union are considered in SAM*.

Property characteristics, such as, mandatory, single- or multi-valued, and others, have been proposed by several researchers [7, 11]. Property inheritance has been studied extensively in the context of type-oriented class creation operations. However, to our knowledge no one discusses the effect of class derivations by set operations on property characteristics. Inheritance of properties and their characteristics is generally studied only between is-a related classes, i.e., for generalization and specialization abstractions.

7 CONCLUSIONS

The contributions of this paper are summarized below. First, the paper presents sound definitions for set operations on the class construct. We show that the semantics of set theory are preserved by these definitions since the resulting set relationships between classes correspond to those of set operations in set theory. A class derivation mechanism would not be well-defined without the specification of the exact treatment of characteristics of inherited properties (especially, when inherited from more than one class). Consequently, we develop rules that regulate the inheritance of properties and their associated characteristics. These rules take care of required versus optional, identifying versus non-identifying, and single- versus multi-valued properties.

In summary, this paper provides the designer of a data model with a framework of set operations which allows him to make an explicit and educated choice among them.

We distinguish between is-a, subset, and subtype relationships. Our analysis of class relationships resulting from applying set operations sheds some light on the implicit assumptions concerning 'is-a' relationships in the literature. Specifically, it is usually taken for granted that an is-a relationship must exist between base classes and a derived class. This assumption is unjustified since, for instance, the symmetric difference operation can never result in a "is-a" relationship (as shown in figure 12). This problem has been avoided in other approaches by simply ignoring the existence of that set operation. As far as we know, the symmetric difference operation has never been utilized as a class derivation mechanism. Our approach, which allows for the symmetric difference operation, results in a data model where property inheritance proceeds along not necessarily 'is-a' related class relationships.

Due to the generality of our approach, results of this paper apply to any conceptual data model which supports the class construct. We have ignored behavioral abstractions (methods) associated with classes of object-oriented systems [6, 10], since their inclusion would not aid the understanding of the presented concepts. We believe, however, that much of this work can be extended to also include the behavioral aspect of classes.

References

[1] S. Abiteboul, and R. Hull, IFO: A Formal Semantic Database Model. *ACM Trans. on Database Systems*, vol. 12, issue 4, Dec. 1987, 525–565.

[2] A. Borgida, Conceptual Modeling of Information Systems. *On Knowledge Base Management Systems*, Springer-Verlag. Brodie, M.L. and Mylopoulos, J. (eds), 1987.

[3] R. J. Brachman, What IS-A is and isn't: An Analysis of Taxonomic Links in Semantic Networks, *IEEE Computer*, Oct. 83, 30 - 36.

[4] P. P. Chen, The Entity-Relationship Model — Toward a Unified View of Data, *ACM Trans. on Database Systems,* vol. 1, issue 1, Mar. 1976, 9–36.

[5] E. F. Codd, Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Systems*, vol. 4, issue 4, Dec. 1979, 397–434.

[6] D. Fishman et al., Iris: An Object-Oriented Database Management System, *ACM Trans. on Office Information Systems*, vol. 5, no. 1, Jan. 1987.

[7] M. Hammer and D. J. McLeod. Database Description with SDM: A Semantic Data Model. *ACM Trans. on Database Systems*, vol. 6, no. 3, Sept. 1981, 351–386.

[8] R. Hull and R. King, Semantic Database Modeling: Survey, Applications and Research Issues, *ACM Computing Surveys*, vol. 19, no. 3, Sept. 1987, 201–260.

[9] S. N. Khoshafian and G. P. Copeland, Object Identity, *Proc. OOPSLA'86*, ACM, Sep. 1986, 406–416.

[10] D. Maier, and J. Stein, A. Otis, and A. Purdy, Development of an Object-Oriented DBMS, *First OOPSLA Conference,* Portland, OR, Sep. 1986.

[11] J. Mylopoulos, P. A. Bernstein, and H.K.T. Wong. A Language Facility for Designing Database-Intensive Applications, *ACM Trans. on Database Systems,* vol. 5, issue 2, June 1980, 185–207.

[12] J. Peckham, and F. Maryanski, Semantic Data Models, *ACM Computing Surveys*, vol. 20, no. 3, Sept. 1988, 153–189.

[13] E. A. Rundensteiner, L. Bic, J. Gilbert, and M. Yin, Set-Related Restrictions for Semantic Groupings, Uni. of Cal, Irvine, Technical Report No. 89-07, Jan. 1989.

[14] E. A. Rundensteiner, and L. Bic, Aggregates in Possibilistic Databases, *VLDB'89*, Amsterdam, Aug. 1989.

[15] D. W. Shipman, The Functional Data Model and the Data Language DAPLEX, *ACM Trans. on Database Systems,* vol. 6, issue 1, Mar. 1981, 140–173.

[16] J. M. Smith, and D.C.P. Smith. Database Abstractions: Aggregation and Generalization, *ACM Trans. on Database Systems*, vol. 2, no. 2, June 1977, 105–133.

[17] Y. W. S. Su, Modeling Integrated Manufacturing Data with SAM*, *IEEE Computer 19*, 1, 1986, 34–49.