Abstract

Object-oriented databases are used mainly in new application domains that are not covered efficiently by past-generation systems. However, requiring all users to work on the same information model limits the extensibility of the system, and introduces difficulties when, for instance, considering data restructuring and database integration. This thesis, presents a view mechanism that confronts these problems by allowing programmers to restructure the class hierarchy and modify the behaviour and structure of objects.

A view is a *virtual* schema graph created by importing data from the database, and then restructuring the imported data. The programmer derives new, *virtual* classes by populating them with existing objects, or with newly created objects. These classes are classified according to their subsumption relationships with the existing classes. Two subsumption testing techniques are presented. We specify our view mechanism by means of a deductive object-oriented language, namely *F-logic*, and address the arising issues.

Dedicated

to my parents

Harry and Irene Charalabopoulos

for their love, encouragement and support,

and to all those who stood by me

when times were rough.

In memory of my late grandmother, Ourania.

Acknowledgments

For all those who helped me complete this *journey*.

First and foremost, I thank my supervisor, Professor Anthony Bonner, for his guidance, enthusiasm and support. I also thank Professor John Mylopoulos who, as the second reader of my thesis, provided me with insightful comments and advice.

From my friends and colleagues, I am mostly thankful and indebted to Dimitris Lagkouvardos for all his precious help during the past sixteen months. For all the things you taught me, for your hospitality and your fine cooking, for those unforgettable summer barbecues, and most of all for your friendship, I thank you Dimitri with all my heart. Many thanks go to Christina Christara for her support, to Dimitris Plexousakis for his helpful guidance, and to Thodoros Topaloglou and Dimitra Vista for providing answers to my questions. I am also thankful to Kathy Yen at the graduate secretary office for her smile, her kindness and her willingness to help whenever needed. Thanks, to my best friend in Greece, Stefano Salpoglou, for always being by my side. Thank you Dora, for everything. $\lambda\alpha$.

Most importantly, I thank my family; Harry, Irene, Tasoula and Kostantis, I thank you all for your love and support. My sincere thanks go to Ted and Maria Govas for their hospitality, their generosity, and most of all, for their love.

Last but not least, I thank the Department of Computer Science and the University of Toronto for providing the means for the completion of this work. This work has been kindly supported in part by an Open Fellowship from the University of Toronto.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Views did not come along with the outburst of object orientation. They pre-existed and, as a matter of fact, proved to be a useful feature in relational database systems. Generally speaking, a *view* is a mechanism that separates the global (conceptual) database schema from the external schema ("subschema") of a particular task [SLT91]. In relational systems, a very simple mechanism is used for view definition: the query. Any query, since it returns a relation, may serve as a view definition mechanism. The additional relation returned by the query is the relational view, while the query itself can be stored in the database catalogue and serve as a view specification for the virtual relation. So a relational view is a *derived* or *virtual* relation, which is defined by a query and added to the list of existing relations. However, the explosion of object orientation in the late 80's, set the basis for the examination of views under a new perspective.

Object-oriented databases were introduced in order to deal with other applications besides the classical business data-processing ones. Computer-aided design, engineering and manufacturing systems, knowledge-based systems, multimedia systems which deal with images, voice and textual documents, as well as programming language systems are new-type applications for which relational database technology (and, in general, past-generation systems) have serious shortcomings. The reason resides in the way these application domains model information. In these applications, real-world entities are represented as *complex objects* with complex internal structures and a large number of unstructured properties. Relational or other past-generation systems are unable to deal with such objects because of the structural irregularity and high complexity, that these objects present. This lack of functionality presented by the conventional systems lead to the development of new conceptual data models, from which the most popular was the object-oriented one. Such models came out in the mid- and late eighties and include [MS87, BCG+87, Kim89].

Besides the requirements of new application domains, object-orientation was also introduced to deal with the *impedance mismatch* between programming languages for writing applications and languages for data retrieval [Ban88]. Application development in relational database systems requires the communication between a relational query language (used for data retrieval) and a programming language (used for writing applications). Unfortunately these two languages have different types and computational models, so they do not intermix well. Object-orientation may alleviate this impedance-mismatch problem.

Although there has been a great deal of experimental activity in the field of object-oriented database systems, there is yet neither a common model nor any formal foundation, like that presented by Codd [Cod70] for relational systems. However, complete object-oriented data models do exist, as aforesaid. There is simply no consensus on which one is optimal. Despite the lack of a common model and the lack of a formal foundation, the database community is converging on the gross characteristics of a family of object-oriented systems [AFD+89], which we outline here.

The conceptual entity in object-oriented systems is the *object*. The simplest objects are integers, characters, byte strings and, in general, atomic types. Object constructors (sets, bags, lists etc.) can be used to construct complex objects from simpler ones. There is an orthogonal relationship between object constructors and objects themselves. Any constructor can apply to any object, in contrast to the relational model, where the set constructor can only be applied to tuples and the tuple constructor can only be applied to atomic values. This feature of an object-oriented model makes the representation of complex objects with irregular structure possible. Each object encapsulates a private state and behavior (methods). This models data and operations on this data at the same time. Objects communicate by passing messages. Every message invokes a particular method of the recipient object. The execution of this method either changes the object's state or returns a result. Every object is assigned (by the system) a unique and time-invariant identifier, which is independent of the object's value. Object identity allows for the unique referencing of objects and distinguishes between objects that are *identical* (i.e., represent the same object) and objects that are *equal* (i.e., have the same value). Similar objects are grouped into classes, while a specific object can belong in more than one class at the same time. An object in a class is said to be an *instance* of that class. Classes can be ordered by generalization and specialization. The resulting hierarchy is called a *generalization hierarchy*. Other hierarchies are the *property inheritance hierarchy*, and the *property decomposition hierarchy*. The former organizes classes into an inheritance structure, in such a way that objects belonging in different classes can share common properties (attributes, methods, arbitrary complex functions). It is also known as the *type hierarchy*. The decomposition hierarchy, on the other hand, is orthogonal to the generalization hierarchy and models the way in which complex objects are formed. For example, if a class $C1$ has a property function p with $domain_p(C1) := C2$, then a property decomposition arc is created between $C1$ and $C2$. This type of arcs are not explicit, but rather are implicitly determined by the classes' specifications.

Having all the aforementioned features and especially the ability to represent complex objects, object-oriented database systems fulfill the needs of design-support systems and other contemporary application domains, which require the handling of complex data, graceful evolution and high performance. However, one might wonder about the necessity and the advantages of using views in such systems. The following section attempts to provide an answer to this question.

1.2 Using Object-Oriented Views

Practical reasons for supporting views in an object-oriented database system include manageability, authorization, integration in a heterogeneous environment, schema modifications, and versioning.

Manageability refers to the size of the global schema. Dealing with a large global schema is a burden. We can resolve this problem by introducing views over the global schema that split it into smaller, more manageable pieces. Another application of views, more likely in the opposite direction, is the integration of (possibly) semantically heterogeneous databases [Mot87, AB91]. An object-oriented view can be thought of as a database that imports all its data from other databases, enabling the user to simultaneously access data belonging to different databases.

Security and authorization are main reasons for using views in object-oriented databases. In many cases, there exists a need to hide some portion of data (e.g., certain attributes) from a specific group of users, while at the same time keeping this information available to other, properly-authorized user groups. In fact, there have been several proposals for authorization models for object-oriented databases. In [RBKW91], a model which supports different levels of granularity is presented. On the other hand, [Thu89] considers mandatory access control. By using views, however, we could provide a content-based access control on instances of a class [Ber92]. This could be done by defining views containing predicates on the values of some specific attributes of the class. Then only objects that meet the specified conditions may be visible through the view. Therefore, since authorization is considered upon the view, only class instances that are not filtered out by the view can be accessed. For instance the following definition of a view:

create-view BigShot as

select E from Employee where salary $> 80,000$.

permits the user to see only those employees not filtered out by the view (i.e., highly paid employees). Therefore, we support authorization depending on object contents.

Object-oriented views can also provide user convenience. This is required mostly in the case of heterogeneous environments. In such environments, users are usually interested only in a (small) portion of the global schema, a *subschema*. This subschema consists of a collection of base and/or view classes together with methods defined on them, and is tailored for some particular task (the one that the user wants to perform). For example, assume a corporate database and two groups of users. The first group is interested in seeing the class *Employee* through two views: *FemaleEmployee* and *MaleEmployee*, while the other group is interested in viewing the employees as *JuniorEmployee* and *SeniorEmployee* depending on the number of years each employee has been working for the company. We can easily define the above views as subclasses of the class *Employee*, without them constituting a partition of the class *Employee*. Note that these newly defined classes may overlap. For instance, a *FemaleEmployee* can also be a *JuniorEmployee*, and a *MaleEmployee* can also be a *JuniorEmployee*, etc.

In general, n virtual classes can overlap in $O(2^n)$ different ways [AB91]. For instance, we could define a combination of the above classes, i.e., *FemaleSenior*, *MaleJunior* but not *JuniorSenior*. A combination of two or more classes is possible if these classes are not comparable. This approach allows the introduction of multiple inheritance in the schema. So, for example, class *FemaleSenior* inherits attributes and methods from more than one superclass (i.e., *FemaleEmployee* and *SeniorEmployee*). Moreover, it allows different interpretations (*FemaleEmployee*, *FemaleJunior* etc.) of the same class (*Employee*), without complicating the class hierarchy organization. This is an important feature in the support of large software bases.

We can also use object-oriented views to support schema changes. Modification of the database schema is required in design application environments, which are expected to use object-oriented database management systems. In such environments and during the exploratory phase, it is important for the user to be able to modify the schema, without this resulting into a system shutdown or an unbearable performance penalty. We can divide schema changes into two categories: *single-schema modifications* and *schema versionings* [Kim90]. *Single-schema* modification is the direct modification of a single logical schema. We first define a taxonomy of meaningful schema changes, and then we specify the semantics of schema changes in the form of a model. This kind of schema modification is already supported in object-oriented database management systems, but presents some problems. For example, if an attribute or method is dropped from a class definition, then all instances of the class lose the values of that particular attribute or method, and there is no way for those values to be recovered at a future point in time. Furthermore, once a user executes a schema change, all other users from then on must use the modified schema. These two examples display how single-schema modification can result in information loss and restricted functionality. Views on the other hand can be used to overcome these problems. Users can experiment with schema changes before actually executing them, and each user can define his own modified schema, without affecting other users, since view definitions do not affect the underlying schema. In *schema versioning*, the system manages more than one logical schema for one common database. If the user needs to drop an attribute or a method from a class, he can define a version schema to reflect the change, while his old schema remains available to him to use in a future point in time. Again, when it comes to supporting versions, views prove to be a quite useful mechanism [Ber92].

From the aforementioned, it is clear that view mechanisms can alleviate the shortcomings of object-oriented database systems. The range of their applications includes manageability, user convenience, authorization, interoperability in heterogeneous environments, database integration, and schema evolution.

1.3 Problem Definition

As we have seen already, views in both relational and object-oriented systems, increase the flexibility of the system and customize the global database schema, so that it suits the needs of particular users. However, as we will see in this section, the view definition process in the context of the object-oriented data model, raises interesting issues. These issues did not arise in the context of the relational model, due to its simplicity and set-orientation.

Recall that a relational view is a *derived* or *virtual* relation specified by a query [Dat90]. Such a relation basically results after renaming or permuting columns of a base relation, or selecting a subset of a base relation which satisfies some predicate, or even as a join of two or more base relations. Besides their simplicity, relational views cannot be freely updated. In many cases, an ambiguity arises when trying to translate an update on the view into an update on the global schema. Furthermore, an update performed on a relational view may have undesirable side-effects on the other views. Therefore, updates on relational views are only of a very limited type, and are not even permitted in most of the existing relational database systems.

On the other hand, updating views in object oriented databases is more feasible [Run92b]. This is mainly due to object identity and abstract data types, which are two concepts inherent in the object oriented model. Since each object is assigned a unique, time-invariant identifier, it is easy to recognize and process an object, even if some of its external characteristics are hidden or modified by a view definition. Furthermore, the notion of abstract data type allows for the addition of type-specific (update) operations to an object class. This results in a view that contains access and update methods along with type descriptions. In this manner, all the updates allowed on a particular view, are pre-specified, and ambiguities are ruled out when considering updates on that view. A more detailed discussion on this issue will be made later on, in the following chapter.

In early proposals, an object-oriented view is defined as a virtual class derived by an object-oriented query [HZ90, SLT91, Kim89]. Comparing the view constructs for the relational data model and the object data model [Run92a] we can see two things (among others). First of all, the view entity in the relational model is the *virtual relation*, which is defined by a relational query and is not explicitly related to any of the other relations. In the object model, the view entity is the *virtual class*, defined by an object-oriented query but interrelated with other classes via diverse relationships. Second, the view schema in the former model is a collection of view definitions, that is, a set of unrelated relations, while in the latter model it is a complex structure of classes, which are interrelated via generalization and decomposition hierarchies. Therefore it does not seem to be a good idea to define an object-oriented view as an individual virtual class. It rather seems that a view in the object-oriented case corresponds to a *virtual, possibly restructured subschema graph of the global schema* [TYI88]. The approach we follow here is of a similar flavor.

This type of definition, though, raises interesting issues which do not exist in the relational model. Such issues include ways of restructuring the global schema graph in order to produce the view, ways of relating the derived view with the global schema, and of course ensuring the consistency of both the database after the derivation of the new view schemes, as well as of any updates performed through the view. One should be able to *modify* object instances and type descriptions of classes, to *reorganize* object memberships in classes, and to *hide* object instances, classes and corresponding relationships. Moreover, we should allow for the *restructuring* of the generalization and property decomposition hierarchies, in order to fulfill the requirements of any particular user. Since, in general, there will be several different users with potentially conflicting interests and requirements, it is not adequate to change the global schema in order to suit the needs of one particular user. Each user should rather be provided a kind of "window" through which he will be able to view the database in a convenient manner. This directly translates into the need to support *multiple* views on the same global schema.

In the upcoming chapters, we will examine the above issues in greater depth, and attempt to provide solutions for these, as well as for other problems arising during the process of object-oriented view definition. In Chapter 2, we take a look at work previously done on the issue of views, for the relational and the object-oriented cases. Chapter 3 presents our view mechanism in general, describing the derivation of view classes and the view definition process. All the ideas of this chapter are then implemented by a deductive object-oriented language (i.e., *F-logic*) in Chapter 4. The upcoming issues are considered and solutions for them are provided. Chapter 5 deals with the integration of the view classes into the hierarchy, while in Chapter 6 we present two approaches to testing for subsumption relationships between classes. The way we test for these relationships, has a major impact on the performance of the whole integration process. We conclude the thesis with a discussion of the contributions made and suggestions for future work, in Chapter 7.

Chapter 2

Database Views: A Trip Through the Past

As presented in the previous chapter, database views have been around for quite a while. The concept of a *view* was first introduced in the relational database model, and within this context, it has been studied and explored extensively. However, since object-oriented databases are relative newcomers to the database field, and since to date, there is no consensus on a unique object-oriented data model [AFD+89], object-oriented views have not been as extensively and thoroughly explored as their relational predecessors.

In this chapter, I give a spherical and descriptive presentation of related work done on database views. The presentation starts with relational views and the issues underlying their definition, and extends to related work in the object-oriented field.

2.1 Relational Views

The mathematical concept underlying the relational model is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains. A *domain* is a set of values. The Cartesian product of domains $D_1, D_2, ..., D_k$ written $D_1 \times D_2 \times ... \times D_k$ is the set of all k-tuples $(v_1, v_2, ..., v_k)$ such that v_i is in D_i. A *relation* over $D_1, D_2, ..., D_k$ is a subset of $D_1 \times D_2 \times ... \times D_k$, and is said to have *arity* k. The members of a relation are called *tuples*. A tuple $(v_1, v_2, ..., v_k)$ has k components, where the i^{th} component is v_i.

Name	Department	Salary
Brown	Comp.Eng.	50,000
White	Comp.Sci.	85,000
Greene	Comp.Sci.	80,000
Black	Math	76,000
White	Comp.Eng.	67,000
Brown	Comp.Eng.	50,000

Figure 2.1: An $EMPLOYEE$ Relation

We can view a relation as a table where each row is a tuple and each column corresponds to one component. The columns are often given names, called *attributes*. The set of attribute names for a relation is called the *relational schema*. A collection of relational schemas is called a *data schema*.

The relational schema of $EMPLOYEE$ in Figure 2.1 is $\{Name, Department, Salary\}$. Each tuple of the relation is in $STRING \times STRING \times INTEGER$ and has arity 3. Tuple $t_6 = <Brown, Comp.Eng., 50,000>$ represents an employee with $Name[t_6] = Brown$, $Department[t_6] = Comp.Eng.$ and $Salary[t_6] = 50,000$.

The relational model is *value-oriented*. This means that each attribute of a tuple has a simple value from a primitive data or enumeration type. This fact enables us to define operations from relations to relations. These operations can be easily combined with the use of an algebraic notation called *relational algebra*. All these lead to the ability of the relational model to support simple, yet powerful, declarative languages, which is perhaps the most important reason for the model's popularity [Ull88]. However, besides being an important feature, value-orientation is also a serious drawback. One reason is the lack of an explicit modelling of relationships between tuples, and of relationships between relation schemes. Relationships between tuples can be inferred by matching attribute domains and values, while relationships between relation schemes have to be inferred by comparing respective attribute domains. Hence, a relational data

Name	Department	Salary
White	Comp.Sci.	85,000
Greene	Comp.Sci.	80,000
Black	Math	76,000

Figure 2.2: The View Relation *Bigshot*

scheme corresponds to a set of unrelated relation schemes.

A *view* in this model is a *named*, *derived relation*, which is defined by a query [Dat90]. This can be seen in Figure 2.2. The view is named *Bigshot* and is derived via the query: $SELECT\ Name, Department\ FROM\ Employee\ WHERE\ Salary > 75,000$. This query is stored in the database catalogue and serve as a view specification for the derived relation *Bigshot*. Therefore, *Bigshot* corresponds to a *virtual* relation since the underlying table is not stored, but only seems to the user as if it does. The *SELECT* clause defines the structure of the derived relation, the *FROM* clause specifies the source relation(s) from which the view is derived, and the *WHERE* clause specifies the condition that the tuples of the source relation must fulfill in order to belong in the view relation. There is no apparent relationship between the view and other (base or view) relations. A collection of base and view tables corresponds to a *view schema*. For instance, the base relation *Employee* of Figure 2.1, and the virtual relation *Bigshot* in Figure 2.2 together define a view schema.

Once a virtual relation is derived, we would like to treat it as any other ordinary relation. Therefore, query and update operations should be performable on relational views. This can be done by converting operations on the view relation into equivalent operations on the underlying base relations. Querying the view for data retrieval does not present any problems. However, updating a relational view is not so straightforward. An attempt to translate an update operation on a view into a similar operation on the underlying base relations may not always be successful. There might not be a

Name	Student-id	Major
George	718932	Comp.Eng.
Angela	972113	Comp.Sci.
Tony	917432	Comp.Sci.
George	552247	Math
George	729921	Comp.Eng.
Theodora	712165	Comp.Eng.

Name	Major
George	Comp.Eng.
George	Comp.Eng.
Theodora	Comp.Eng.

Figure 2.3: The $STUDENTS$ Relation and the View $Comp.Eng.Students$

unique and unambiguous update on the global schema, or it may be the case that such an update presents undesirable side effects on the view itself. These two facts constitute the *view update problem* [BS81, DB82], that is, the problem of properly translating an update request on a view, into an update request on the underlying global schema. In general, there are only a few cases in which the view update can be unambiguously translated into an update on the base schema. This seems to explain why most existing databases either restrict the types of view updates they can support, or even disallow updatable views.

In Figure 2.3, it is easy to see that the view relation $Comp.Eng.Students$ defined by the query: $SELECT\ Name, Major\ FROM\ Student\ WHERE\ Major = Comp.Eng.$ is not updatable. If, for example, the tuple <*George*, *Comp.Eng.*> is chosen to be deleted from the view, then it is not clear to the system which of the underlying tuples <*George*, 718932, *Comp.Eng.*> or <*George*, 729921, *Comp.Eng.*> is to be deleted from the base relation. Trying to modify $George's$ major to CS also lacks an unambiguous underlying translation. It seems that the ambiguity in the translation of view updates into updates on base relations is more likely to arise when the view definition does not preserve the primary key of the underlying base relation. However, if the primary key is contained in the derived view relation, then it is more likely that the view will be updatable. For instance, if the view definition query selects $Student - id$s and $Major$s (instead of $Name$s and $Major$s) then deleting <718932, *Comp.Eng.*> can unambigu-

Name	Major
George	Comp.Eng.
Theodora	Comp.Eng.

Figure 2.4: The Actual *Comp.Eng.Students* View

ously be translated into deleting <*George*, 718932, *Comp.Eng.*> from the underlying *STUDENT* relation.

Yet another problem with the view definition in Figure 2.3 arises from the fact that since the relational model is value based, it does not allow duplicate tuples to exist in the same relation. This means that the view relation defined in the previous case will not have three tuples. The base tuples <*George*, 718932, *Comp.Eng.*> and <*George*, 729921, *Comp.Eng.*> are both reduced to <*George*, *Comp.Eng.*> in the view relation of Figure 2.3, Thus the resulting view will actually be the one in Figure 2.4.

In the object-oriented case however, each object is assigned a unique, time-invariant identifier [KC86]. This identifier assures that even if the external characteristics of a particular object are changed, the system will be able to keep track of it. Therefore, side-effects similar to the one presented in Figure 2.4 are not likely to occur. In such cases, the view would contain two *equal* object instances, which would not, however, be *identical*. Moreover, in an object-oriented view description, it is possible to include type descriptions as well as access and update methods. This can eliminate ambiguities when updating an object-oriented view. A model for updatable object-oriented views is presented in [SLT91].

2.2 Extending Relational Views

In order to deal with the relational view update problem, Mylopoulos and Brodie [BMS84, BM87] presented an approach based on the notion of an *abstract data type* (*ADT*). In this approach every object of a particular type is characterized by the operations performed on it. The details of the presentation of each object are not for the user to see. A description of all the allowed view updates along with their translations into updates over the base schema, are incorporated into the description of the view derivation from the underlying database. However, their view definition language can no longer remain in the context of the relational model and moreover, the view designer is assigned the burden of explicitly translating each operation on the view into operations on the underlying base relations. Methods for the verifying the correctness of these updates have to be used.

2.3 Other Object-Oriented Approaches

2.3.1 Allowing Multiple ADT Interfaces

Shilling and Sweeney's approach [SS89] falls into this category. They allow a class to have multiple abstract data type interfaces (types), instead of having just one. This can be used in order to limit the access rights to property functions and to control the visibility of instance variables. However, this extension of the class concept leads to a mechanism that is too complex to be implemented on top of existing object-oriented database technology. Furthermore, in their approach, Shilling and Sweeney do not consider virtual classes that result from the restriction of the membership predicate of source classes. In other words, they do not deal with derived classes that are specializations of other source classes. This is because they approach the view definition problem from a programming language point of view, and therefore are not concerned about the set of objects belonging to a class. They focus on one class only, without addressing the effects that multiple interfaces have on the class hierarchy. The restructuring of the is-a hierarchy is not considered as a way to form object-oriented views.

2.3.2 A View as an Object-Oriented Query

As in relational systems, there have been several proposals for object-oriented view models which define a view to be a *virtual class* derived by an object-oriented query. The view mechanism defined by Kim for the Orion model [Kim89], as well as the one defined for the Fugue model [HZ90] by Heiler and Zdonik are two examples that follow the aforementioned approach. However, no extra care is taken for the classification of the derived virtual classes. Unlike the relational case, where derived relations are completely unrelated to the set of existing relations (either base or derived), in the object-oriented case, the derived classes are interrelated with other classes via diverse relationships. These relationships have to be explicitly stated, in order to share properties and to avoid information duplication. This issue, as stated earlier, is neither considered in [HZ90] where the derived classes are treated as 'stand-alone' objects, nor is it properly dealt with in [Kim89] where the derived classes are attached directly as subclasses of the schema root class.

Another proposal that considers a view to be a virtual class is that of Scholl et al. [SLT91]. They present a view mechanism for the *COCOON* object model. They use a subset of the operations of the *COOL* query language to derive virtual classes, each of which denotes a separate view. However, in contrast to the other two approaches presented above, they go on to discuss the classification (i.e., placement) of the derived view classes into the global schema. They address the issue of view updates, but this is done only in the context of object preserving operations. They do not allow imaginary objects in their views, which limits the flexibility of their approach, as well as the expressive power of their language. There is no mention of how several users with potentially conflicting interests in viewing the global schema will be able to do this concurrently.

2.3.3 A View as a View Schema

Since the derived classes are interrelated via diverse relationships, it seems more natural to define a view as a *view schema*, rather than as a single virtual class. Some of the proposals for view models in this direction are by Tanaka et al. [TYI88], by E.A.Rundensteiner [Run92b] and by Abiteboul and Bonner [AB91].

Tanaka et al. deal with schema virtualization. In their approach they do not clearly distinguish between the integration of derived classes into one common schema and the generation of view schemata. Even though they point out the need for the integration of the virtual classes, no general classification algorithm is presented. On the contrary, they allow manual addition of is-a edges into a virtual schema which, in many cases, may lead to inconsistencies due to human errors.

Abiteboul and Bonner [AB91] present a view mechanism for the O_2 database system. They use a slightly modified version of the O_2 data model, which does not distinguish between (stored) attributes and (derived) methods, resulting in a flexible and informative model. Information can be derived in the form of computed attribute values (methods), allowing the rearrangement of information within an object. This rearrangement enables the view definer to tailor different views for different applications, depending on the users' needs. Based on the O_2 data model, it makes an extensive use of existing mechanisms, such as the O_2 query language, database imports, object creation, as well as inheritance and overloading. It also introduces new features such as grouping all objects with similar behaviors, and parameterized ways of grouping objects. In this approach the view definer can introduce new classes (called *virtual classes*), and populate them with existing objects from other classes, or newly created (*imaginary*) objects. To populate virtual classes with imaginary objects, a query that returns a set of values can be specified and then a new object identifier can be attached to each value. After the definition and population of the virtual classes the system derives the internal structure and the behavior of that class. The new class can then be treated as any other class.

Rundensteiner presents in [Run92b] a methodology, called *Multiview*, for supporting multiple view schemata of a global schema. The view specification process is broken into three tasks. First new classes are derived, by means of an object algebra. Then the derived classes are integrated into one global schema. And lastly, the classes that will comprise the view are selected from this global schema, and the view schema is constructed by the system. The object algebra used for the first step, is object preserving. That is, generated objects are not allowed, which limits the power of the view mechanism. The integration of the derived classes into the global schema (step 2) is performed automatically, however, not much is done on deriving the subsumption relationships between the derived class and the other classes of the global schema. Instead, the $subsumes()$ function used for this task, is assumed to be given, and to have constant complexity. It is also noteworthy, that deriving the virtual classes in groups (step 1) and then classifying them all together (step 2), does not allow for a virtual class to be treated as any other class, right after the virtual class is derived. This can be done, only after all the classes in the group of derived classes, are classified (step 2).

Chapter 3

The View Model

This chapter outlines our approach in supporting a view mechanism for object-oriented databases. We start with several definitions, and then give an overview of the view model. We then describe the view definition process, and finally we discuss issues on generating new objects.

3.1 Preliminary Definitions

In this section, we provide some basic definitions needed in the rest of the thesis. These definitions are similar in flavour to others in the literature [Ban88, AFD$^+$89, SLT91, Run92b].

Objects are instances of abstract data types (ADT's). Each object, o, in the infinite set of object instances, O, has an interface, by which the user can access and manipulate it. An object in a class is said to be an *instance* of that class.

Properties are described by a name and a signature. Properties can be either single- or set-valued. More specifically, if T is the set of all types, and $t \in T$, then $properties_t$ is the set of properties of type t. Each property can be either a value from a simple predefined enumeration type (simple attribute), or an object instance of some class (complex attribute), or an arbitrary complex function (method). We use $domain_p(t)$ to denote the domain (range) of property p in type t.

Types are used to describe the common interface of collections of objects instances. In other words, the type T of some class of objects describes the properties applicable to these objects, and their interactions. The objects are said to be *instances of type* T.

Classes are means of grouping objects of some specific type. A class, C, in the set of all classes, C^*, has a unique name, a type description and a set of object instances. The type for class C is referred to as $type(C)$, and the set of C's property functions as $properties(type(C))$ or just $properties(C)$. For each class, C, there is also a corresponding set of objects that this class contains, called $extent(C)$. The elements of this set are called members of the class and are instances of $type(C)$. Individual members of one type, may also be members of other types. This is called multiple instantiation.

Subtyping and Type Hierarchy

For two types t_1 and t_2, t_2 is called a *subtype* of t_1, denoted $t_2 \preceq t_1$, if and only if

- $properties_{t_1} \subseteq properties_{t_2}$, and
- $(\forall p \in properties_{t_2})(domain_p(t_2) \subseteq domain_p(t_1))$.

This definition defines a partial order "$\preceq$" on types. The above two conditions state respectively that a subtype has at least as many properties as its supertype, and that the domains of the properties in the subtype, are a subset of the corresponding domains in the supertype. We allow multiple inheritance, that is, types may have more than one supertype.

A *type hierarchy* is a directed acyclic graph $TH = (TV, TE)$, where TV is a finite set of types, and TE is a finite set of directed edges. Each type $t_i \in TV$ is related to some other type $t_j \in TV$ via a subtype relationship represented by an edge $e \in TE$. More specifically, $e = <t_1, t_2>$ is a directed edge from t_1 to t_2 denoting that $t_1 \prec t_2$ (i.e.,

t_1 is a subtype of t_2).

The *greatest common subtype* of two types t_1 and t_2, denoted by $t_3 = t_1 \sqcup t_2$, is defined to contain the collection of properties defined for either t_1 or t_2, as follows:

- $properties_{t_3} = properties_{t_1} \cup properties_{t_2}$, and
- $domain_p(t_3) = domain_p(t_1) \cap domain_p(t_2)$, for all $p \in properties_{t_3}$.

We assume that if $p \notin properties_{t_i}$, then $domain_p(t_i) = U$, where U is the union of all possible domains.

In a similar manner, we define the *least common supertype* of t_1 and t_2 to be a new type t_3 with:

- $properties_{t_3} = properties_{t_1} \cap properties_{t_2}$, and
- $domain_p(t_3) = domain_p(t_1) \cup domain_p(t_2)$, for all $p \in properties_{t_3}$.

The new type is a collection of properties common to both types t_1 and t_2.

Subclasses and Class Hierarchy

An *object schema* is a directed acyclic graph $S = (V, E)$, where V is a finite set of vertices, and E is a finite set of directed edges. For two classes C_1 and C_2, where $C_1, C_2 \in V$, the directed edge $e = <C_1, C_2>$ from C_1 to C_2, represents the following constraints:

- $(\forall o \in O)((o \in C_1) \Rightarrow (o \in C_2))$,
  meaning that C_1 is a *subset* of C_2 (denoted $C_1 \subseteq C_2$), and
- $type(C_1) \preceq type(C_2)$,
  meaning that C_1 is a *subtype* of C_2 (denoted $C_1 \preceq C_2$).

We say that $C1$ is a *subclass* of $C2$, or $C1$ *is-a* $C2$. We also assume the existence of a unique root node *Root*, containing all the object instances of the database, and having an empty type (i.e., no properties).

By keeping the *is-a* relationships explicitly modeled, we can check for relationships between classes by means of existing arcs or paths between the respective classes in the generalization hierarchy. For instance, C_1 is a *direct subclass* of C_2 if the edge $e = <C_1, C_2>$ exists in E. Similarly, C_1 is an *indirect subclass* of C_2, denoted C_1 *is-a** C_2, if there is a path (of length ≥ 0) through the class hierarchy, that connects C_1 and C_2. We should note that the *is-a* relationship between classes is *reflexive*, *transitive*, and *antisymmetric*, and therefore defines a partial order.

It turns out to be quite advantageous for us to explicitly model the *is-a* relationships between classes, rather than having to recompute them whenever necessary. For instance, maintaining the *subset* relationship between classes, allows us to determine the containment of object instances of one class in the extent of some other class. This can prove quite useful for query processing. Similarly, explicit representation of the *subtype* relationship, allows for efficient property inheritance. That is, property function code can be reused by inheritance from the supertype to the subtype. Hence, the redundancy of defining the same piece of code in each an every class in which it is used, is eliminated.

Virtual Classes and View Schema

Initially the database consists of a number of *base* classes, which are populated with *base objects*. Both are defined during the initial schema definition and are explicitly stored. On the other hand, by posing object-oriented queries during the lifetime of the database, the view definer produces a number of *virtual* classes. These classes are not explicitly stored; instead, their definitions are added to the existing schema. Each virtual class has an associated membership derivation predicate, which determines its extent depending on the state of the database. This extent is computed on demand.

In this thesis, we define a *view schema* (or *view*) to be a triple $VS = (id, VV, VE)$ with the following properties:

- id is a unique object identifier for the view,
- VV is a set of virtual classes, and
- VE is a set of directed edges between members of VV, denoting is-a relationships.

Assigning a unique object identifier to each view makes it possible to refer to and distinguish between different view schemas. The last two conditions state that the view contains only virtual classes and their is-a relationships. The above definition enables us to see a view just as a database, but with no proper data of its own. We will also use the term *virtual schema*, to refer to a view, as it is commonly used in [TYI88, AB91].

3.2 The Model in General

The model we are going to present in the rest of this chapter, comes from the earlier work by S.Abiteboul and A.J.Bonner. In [AB91] they give an informal description of a view mechanism for the O_2 object-oriented data model. In what follows, we outline the basic features of this approach and clarify the semantics through examples and schematical representations. We also provide a formal syntax for the view definition language and a way of deriving virtual classes as "intersections" of source classes.

A view is considered to be a database that initially imports all its data from other databases. In general, there may be more than one database in a system. In such systems, data can be shared by means of a view which imports its data from several databases. This situation is presented in Figure 3.1. The view called $view_1$, imports data from both DB_1 and DB_2. The imported data can then be restructured to satisfy the needs of particular users. We should note that the imported data is not a collection of unrelated classes, but corresponds to hierarchies of classes from the source database. This implies

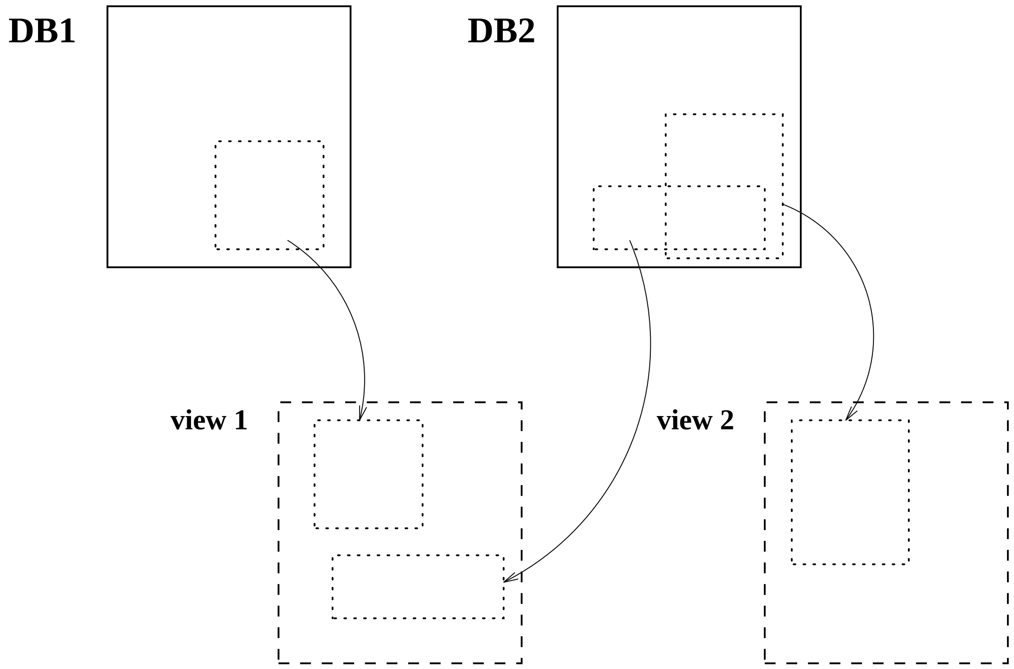


Figure 3.1: The Concept of a View in our Model

that the resulting view will also have a schema, like all databases. It will not, however, have any proper data of its own. Treating the view as another database makes it possible to define views on top of views, as we demonstrate in Chapter 4.

Some basic ideas underlying the design of the view mechanism, taken from [AB91], are summarized bellow:

1. A view should be treated as a database, which implies that virtual classes should be useable as base ones.

2. View construction should be a simple procedure, even for the inexperienced user.

3. Cumbersome tasks should be automated whenever possible, and not assigned to the user.

4. The view should provide both power and flexibility.

The simplicity of the view specification mechanism is demonstrated in the following section. The derivation of virtual classes is performed with a declarative and simple query-like language, appropriate even for non-database experts. Moreover, we provide an automated procedure for integrating the derived virtual classes into the hierarchy, in order to relieve the user from tedious and mundane tasks. Not only is this convenient, but it also ensures the consistency of the resulting view schemata, by avoiding errors that could occur if the classes were to be classified manually by the user. Last but not least, behavioral and parameterized ways of deriving virtual classes, and generation of imaginary objects set the basis for a powerful and, at the same time, flexible view mechanism.

3.3 The View Definition Language

The view definition language, presented in this section, can be used by a programmer to specify views within the context of our model. The BNF syntax of our view definition language is presented in Figure 3.2. The language has two groups of operators: those which initiate or terminate a transaction on a particular view schema, and those which account for the derivation of the view classes, i.e., the object algebra.

The *create-view* command initiates the whole view definition process by assigning a unique object identifier to the new view <*view*-name>. Each *create-view* statement is accompanied by an *end-view* statement, denoting the end of the view definition. In other words, *end-view* signifies the completion of the view creation transaction.

In the second group of view definition commands, we find all those commands which actually derive the virtual classes for a particular view. These commands are the <*view-operations*> of Figure 3.2. One can divide these operations into those that manipulate the content of classes, and those that manipulate the type description of classes. For example, the *hide* operator is only type manipulating, while specializing the set membership of a class according to a predicate (*specialization*), is content manipulating. Some operations are both type- and content-manipulating, such as the intersection operation, which produces a virtual class with both a set membership and a type description that

<*view*-creation>::=

create-view <*view*-name> *as*

{ <*view-operations*> }

end-view

<*view-operations*>::=

| *import class* <*class*-name> *from* <*database*-name>;

| *hide property* <*property*-name> *in* <*class*-name>;

| *class* <*virtual*-class> *includes* <*inclusion*-predicate>;

| *class* <*virtual*-class> *intersection of* <$class_1$>...<$class_n$>;

| *class* <*virtual*-class(<*param*>)> *includes* <*select*-query>;

<*inclusion*-predicate>::=

| <*select*-query>;

| <$class_1$>...<$class_n$>;

| *like* <*class*>;

<*select*-query>::=

| *select* <*variable*> *from* <*source*-class> *where* <*predicate*>;

Figure 3.2: The BNF Notation of the View Definition Language

create-view AbstractFemaleMngr as

import class Manager from Company;

hide property Salary in Manager

class FemaleManager includes

(*select M from Manager where M.sex* = "Female")

save-view

end-view.

Figure 3.3: An Example View Definition

are different from those of the source classes. An example of a view definition is given in Figure 3.3. The new view is created by importing class *Manager* from the database *Company*. Then the *Salary* attribute is hidden in the view, and a new class of female managers is derived. The view is then saved, and the definition is concluded by the *end-view* command.

In what follows, we give a detailed description of each of the <*view-operations*> in Figure 3.2. We also give examples that illustrate the source classes (arguments of each operation), the derived classes (results of the operation), and the resulting relationships between the source and the derived classes.

3.3.1 Importing

Importing is the first operation in a view definition. The view can be thought of as initially being empty, and then by using import statements, the programmer specifies what portions of information from other databases should be made visible. The syntax of the import statement is:

import class <*class*-name> *from* <*database*-name>;

Note that since views are themselves considered to be databases, a <*database*-name> in the above statement can also refer to a view. Semantically, when <*class*-name> is imported, it becomes visible together with all its subclasses. Therefore, an import statement does not only account for the import of a single class into the view, but it also accounts for the import of a whole class hierarchy, rooted at <*class*-name>. This eliminates the need to provide separate import statements for each and every single class being imported.

One more issue, which arises when importing classes from different databases into a single view, concerns the relationships between the newly imported classes within the context of the view. For instance, assume two databases (or two views) which both have a class called *seniorMngr* denoting senior managers of a company. Importing both of these classes into the same view would make them both visible, along with their subclasses. However, the relationship between the two classes in the view is not quite clear. In general such classes seem to be distinct, since they will possibly have different attributes and methods applicable to their object sets. We do not address this matter any further here, but in the sequel, we will assume that classes imported from different databases (or views) are distinct, even if they bear the same name. Such a distinction can be made at the implementation level, as we present in Chapter 4. Without loss of generality, we focus primarily on importing from a single base schema, in order to clarify our approach while keeping the presentation simple.

3.3.2 Hiding

After importing class hierarchies into a view, the programmer can perform additional restructurings on these hierarchies. One of the most common of restructurings is to hide attributes or methods. Hiding is a type manipulation operation and does not in any way affect the set content of a class.

As first pointed out in [AB91], in object-oriented databases, the hiding of a property must be performed explicitly, in contrast to hiding in relational databases, where it is done in an implicit manner. For instance, consider a *Student* relation with attributes $[Name, StudentNo, GPA]$. If we need to hide the *Name* of each student this has to be done implicitly by projecting onto all attributes except *Name*. The following select query accounts for this projection:

$$
\begin{array}{l}
\textit{create-view AbstractStudent} = \\
\qquad \textit{select } [S.StudentNo, S.GPA] \textit{ from } S \textit{ in Student}.
\end{array}
$$

Besides being burdensome, this approach also depends on the schema of the *Student* relation. Whenever this schema changes, the definition of the view must also change. For example, if a new attribute *Major*, denoting a student's major is added to the *Student* relation, then *Major* must also be added to the list of attributes selected by the previous query.

Another problem that this approach presents, is its inability to be applied to object-oriented views. Such an implicit way of hiding is simply not correct for the object-oriented case. To see this, consider a class *Student* and its subclass *TA*, denoting the class of teaching assistants. According to this *is-a* relationship, *TA* will inherit all properties of *Student* and may have additional properties of its own, e.g., an attribute *Tutors* denoting the course that each *TA* tutors. If we attempt to hide attribute *Name* in class *Student* by using a query similar to the one above, then not only will the *Name* of each student be hidden, but the *Tutors* attribute of students who are teaching assistants will also be projected away. Therefore, in the object-oriented case, the hiding of a property must be performed explicitly.

In the model we present here, the hiding operation has the following syntax:

hide <*property*-name> *in class* <*class*-name>.

Semantically, when a property <*property*-name> is hidden in class <*class*-name>, then it is also hidden in all subclasses of <*class*-name> too. This approach to hiding is more general than approaches taken for hiding properties in other view models. To our knowledge, all other approaches hide a property in a single class only. They create a new class, which is similar to the source class, except that it does not contain the hidden property. It is clear that this kind of approach is less general than the one we follow here. While in our approach, hiding in a class and all its subclasses is performed with only one statement, the other approaches need one statement for each and every class in which the property is to be hidden. Besides being less general, the latter can also be quite cumbersome.

3.3.3 Specializing

After importing class hierarchies and hiding properties, the view definer can proceed with the derivation of new view classes from the existing ones. These new classes are populated with objects existing in the view and are called *virtual* classes.

One such way to derive virtual classes is the *specialization* operation. Specialization creates a new view class and populates it with objects returned by a database query. The syntax of the corresponding statement is:

class <*virtual*-class> *includes*
(*select* <*var*> *from* <*source*-class> *where* <*predicate*>),

where <*source*-class> is some already existing view class and <*predicate*> is the con-

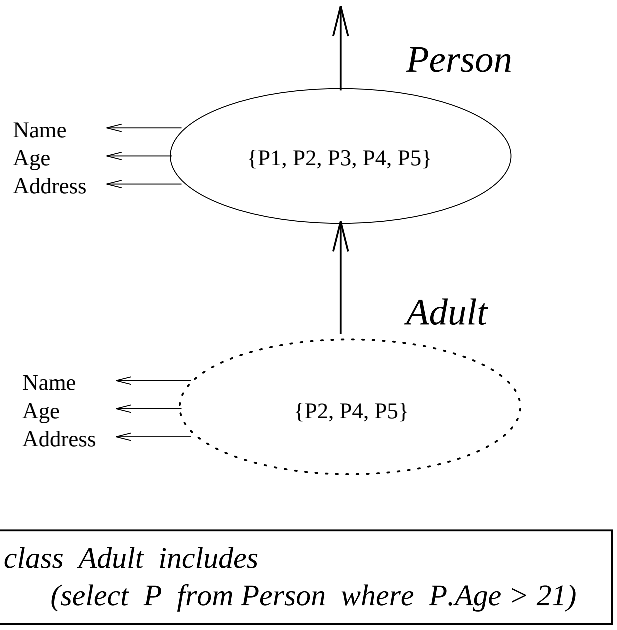


Figure 3.4: A Specialization Example

dition that objects in <*source*-class> have to satisfy in order to be selected to populate the new class. The resulting virtual class, <*virtual*-class>, is a subclass of the <*source*-class>. Note how specialization deals only with the extent and not with the type of the <*source*-class>. The extent of the virtual class is a subset of the extent of the <*source*-class>, while the type remains the same. Figure 3.4 shows a virtual class, *Adult*, derived as a specialization of a source class, *Person*. *Adult* is populated with all the objects in the extent of *Person* with an attribute *Age* that is greater than 21. These are the objects returned by the select query. Virtual class *Adult* is a subclass of class *Person*, and all properties applicable to *Person* objects apply to *Adult* objects, as well.

3.3.4 Generalizing

Generalization can be thought of as an operation that creates virtual classes in a fashion opposite to that of specialization. More specifically, while specialization creates virtual classes in a top-down fashion, generalization works bottom-up.

A virtual class is defined to be a generalization of source classes by the following

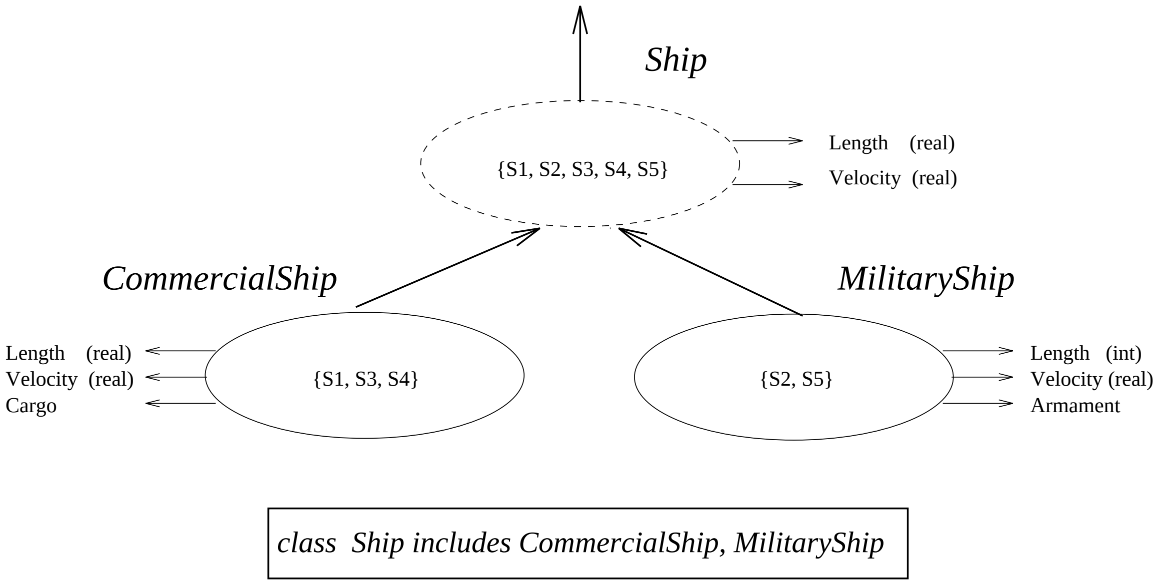


Figure 3.5: An Example of Generalization

command:

$$class <virtual\text{-class}> includes <class_1>...<class_n>.$$

That is <*virtual*-class> is the union of all source classes $<class_i>$, and each of the latter is a subclass of the new class. The extent of the virtual class includes every object instance belonging in the extent of at least one of the source classes, while the type of the new class includes properties common to all source classes. If every source class $<class_i>$ has a property A of type T_i, then virtual class <*virtual*-class> will have a property A of type T, where T is the least upper bound of all the T_i's. If there is no such least upper bound, then A is undefined in the virtual class.

Figure 3.5 presents an example of a virtual class $Ship$, which is defined as a generalization of source classes $MilitaryShip$ and and $CommercialShip$. The new class will include attribute $Velocity$ of type $real$, which is common to both subclasses. It will also include attribute $Length$ with a type which is the least upper bound of $real$ and int, i.e., the type of $Length$ will be $real$ in the new class. There is no other property applicable

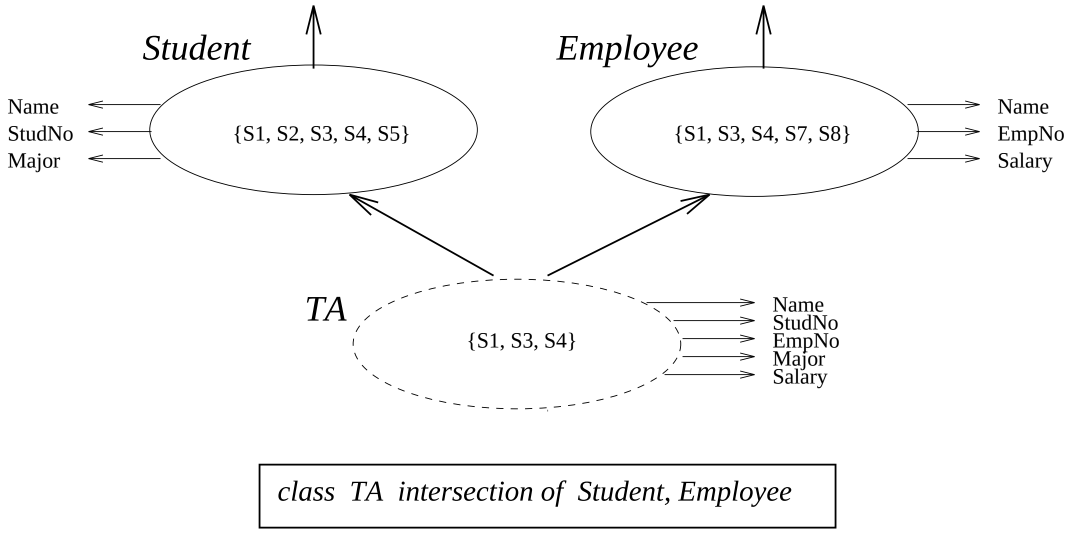


Figure 3.6: An Intersection Example

to objects of both classes; therefore, no other property will apply to objects of the new class *Ship*.

3.3.5 Intersecting

The intersection operation manipulates both the type description and the set membership of the source classes on which it operates. To denote that a virtual class is an intersection of several source classes, we use the following syntax:

$$class <virtual\text{-class}> intersection\ of <class_1>...<class_n>.$$

The virtual class is populated with those objects that belong to all source classes $<class_i>$. The type of $<virtual$-class$>$ is the greatest common subtype of the types of the source classes, as defined in section 3.1. These type and set relationships denote that the virtual class is a subset of all source classes.

In Figure 3.6, we present an example of how the intersection operator can be used to derive a virtual class. Given the two source classes *Student* and *Employee*, the virtual class TA, denoting the class of teaching assistants, is derived by intersecting them. Objects which are both students and employees are also TA's, so they populate the new class. The properties that will apply to these objects as members of the new class are all those applicable to objects in the source classes. Therefore, the type of class TA will include properties *Name*, *StudNo*, *EmpNo*, *Salary*, and *Major*. The domain of attribute *Name* in TA is the intersection of the domains of *Name* in the source classes *Employee* and *Student*.

3.3.6 Increasing the Flexibility

In the previous sections we saw how the view definition process is initiated and how virtual class hierarchies are built in both bottom-up and top-dow fashions. In this section we present two ways in which the flexibility of the view mechanism can be increased. In particular we supply the programmer with a construct that groups classes according to their properties, and with a construct that groups classes according to some parameter. The former, called *behavioral generalization*, is used to derive *behavioral classes*, while the latter, referred to as *parameterization*, is used for *parameterized class* derivation.

Behavioral Generalization

Behavioral generalization increases the flexibility of bottom-up constructions by introducing an additional way to group classes. As we saw earlier, generalization groups classes according to their names. That is, it creates a virtual class that includes all the classes explicitly specified by the view definer. On the other hand, behavioral generalization enables the user to group all classes, with a specific behavior. The syntax for the corresponding command is:

class <*virtual*-class> *includes like* <*class*>.

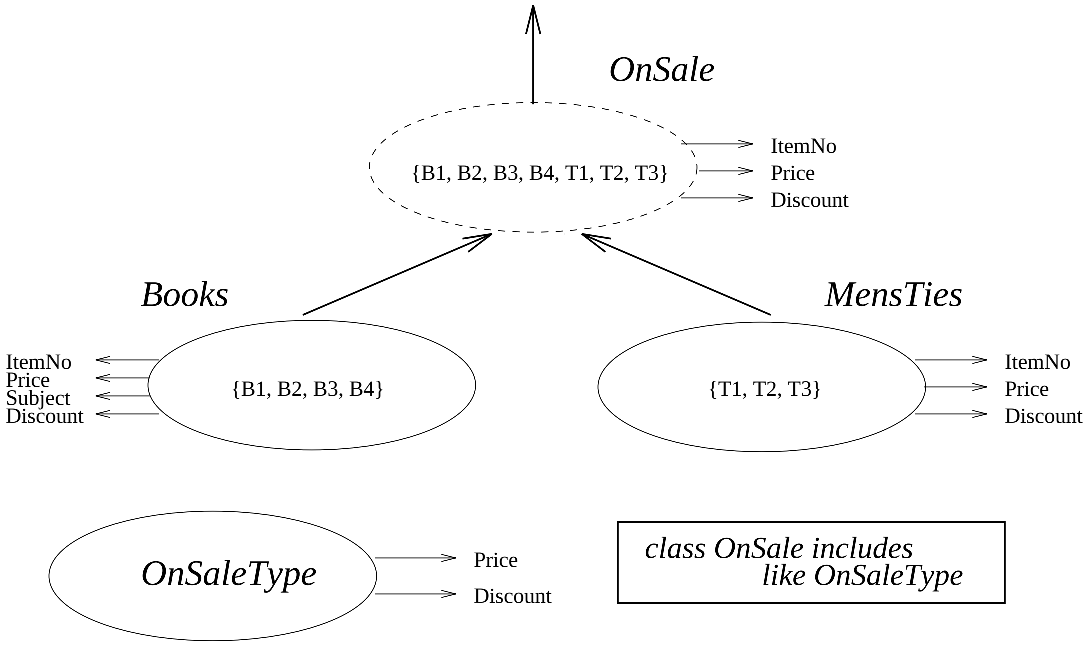


Figure 3.7: An Example of Behavioral Generalization

This means that virtual class <*virtual*-class> includes all those classes which have a type description at least as specific as the one of <*class*>. If <*class*> has, for instance, two properties p_1 and p_2, then <*virtual*-class> will be the union of all classes which have both p_1 and p_2, possibly along with other properties. This is illustrated in the example of Figure 3.7.

Class *OnSale* is defined to include all classes that are at least as specific as *OnSaleType*. Therefore, *OnSale* includes classes *Books* and *MensTies*. The latter classes both have the required properties, *Price* and *Discount*. Note that they also have additional properties like *ItemNo* and *Subject*. It is easy to see the convenience that this sort of definition provides. If *OnSale* was to be defined as a simple generalization, e.g.,

class OnSale includes MensTies, Books

then we would have to change the definition of *OnSale*, every time a new class of objects was to be put on sale. For instance, to put $Shoes$ on sale we would have to write:

$class\ OnSale\ includes\ MensTies, Books, Shoes$

Similarly, when objects are no longer on sale. This would be cumbersome in a department store environment, where hundreds of items are put on and off sale every few weeks. On the other hand, when using behavioral generalization we do not have to explicitly mention the classes of objects to be put on sale, and moreover, we do not have to change the definition of $OnSale$ for each change made. When a class has the attributes $Price$ and $Discount$, it will automatically be included in $OnSale$, while hiding either of these properties from some class will remove the objects of that class from $OnSale$.

However, the semantics for behavioral generalization specified above can dynamically modify the type of the derived class. This is because the derived class <*virtual*-class> is defined to include not only the properties of <*class*>, but also all other properties common to the source classes, as is done in generalization. For example, in Figure 3.7, the derived class $OnSale$ not only has properties $Price$ and $Discount$ of class $OnSaleType$, but also has property $ItemNo$ that is in both the source classes, $Books$ and $MensTies$. If a new class $Shoes$, with properties $Price$ and $Discount$ but without property $ItemNo$, is created, then the resulting type of class $OnSale$ will only include properties $Price$ and $Discount$, and not property $ItemNo$, since the latter will not be a property in all source classes, $Books$, $MensTies$ and $Shoes$. Hence, the type of $OnSale$ might change, depending on the underlying source classes.

This becomes a major problem once we consider placing the derived class into the hierarchy. Since the type of the virtual class is variable, depending on the types of the underlying source classes, any addition/removal of a source class to/from the set of source classes grouped by behavioral generalization can produce a new type for the virtual class. This means that the virtual class will probably have to be removed from its previous place in the generalization hierarchy, and then be reclassified. However, invoking the classification algorithm every time the type of the virtual class changes is clearly an unacceptable overhead.

A solution to this problem is to define the virtual class that is derived by behavioral generalization to have a standard type which does not depend on the types of the underlying source classes. Specifically, the virtual class is defined to have a type description that includes only those properties that are guaranteed to be common to all source classes at any moment in time. This is just the type of the class $<class>$ in terms of which $<virtual\text{-class}>$ is defined. Therefore, in our example, the virtual class $OnSale$ will have the same type as the class $OnSaleType$, which it is defined to resemble. Thus, the the type of $OnSale$ has only the properties $Price$ and $Discount$, regardless of any additions or removals of source classes in the grouping.

Parameterization

While behavioral classes enhance the flexibility of bottom-up constructions, parameterized classes increase the flexibility of top-down constructions. Parameterization can be thought of as an operation that partitions a source class into subclasses according to a parameter. It derives not only a single virtual class, but a *group* of parameterized virtual classes. The syntax of a parameterization statement has the following form:

$$class\ <virtual\text{-class}(<param>)>\ includes$$
$$(select\ <variable>\ from\ <source\text{-class}>\ where\ <predicate>);$$

This results in a set of virtual classes $<virtual\text{-class}(<val_i>)>$, where val_i is a value of the parameter $<param>$ used by the selection query. If n is the number of *distinct* values that the selection query returns for $<param>$, then the number of the virtual classes derived will also be n.

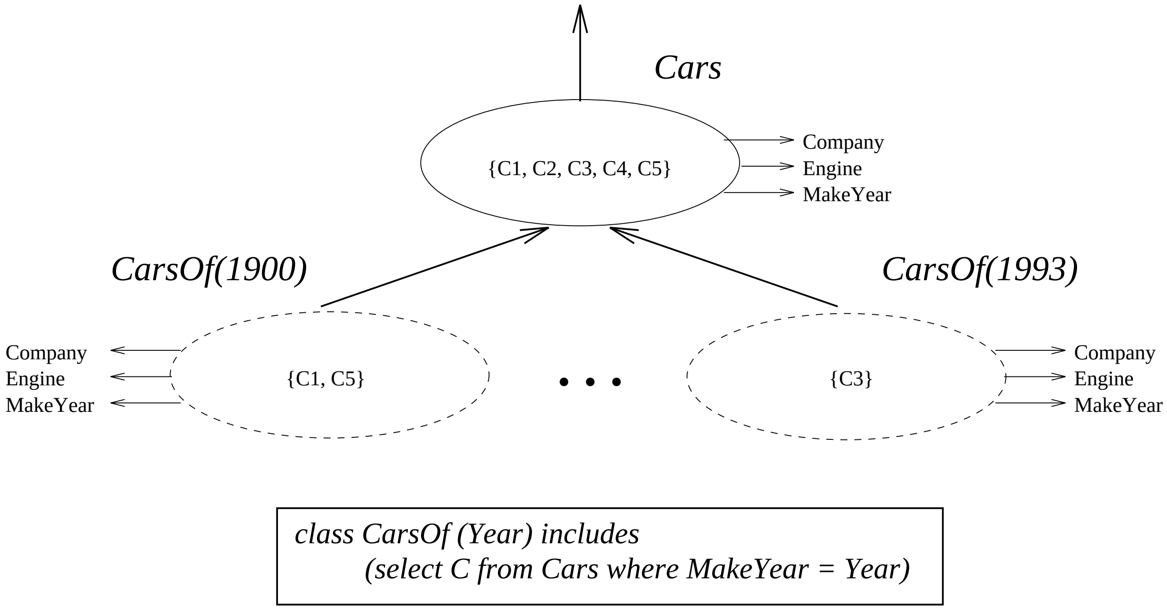


Figure 3.8: An Example of Parameterized Class Derivation

Figure 3.8 gives an example where we group cars according to the year they were built. This is done by parameterizing the values of the attribute $MakeYear$. The result is a group of classes of the form $CarsOf(Year)$, where $Year$ ranges over the set of values of attribute $MakeYear$. All the derived classes are subclasses of the source class Car and, moreover, no one of them has objects in common with any of the others.

This is more convenient than providing a separate class declaration for each year. Furthermore, as cars with new values for their $MakeYear$ attribute are added to the database, they will automatically be grouped into the corresponding new parameterized class. Hence, when new 1994 models are added to the database, they will automatically form the class $CarsOf(1994)$.

3.3.7 Imaginary Objects

All the operations presented up to now are object preserving. They populate the virtual classes with database objects. This section, on the other hand, presents ways to populate virtual classes with generated objects, that is, objects not previously existing in the database. Such objects are called *imaginary objects* and are considered in a number of papers (e.g., [AK89, Kim89, AB91, KLW93]]). The main objective behind using such objects is to increase the power of the view model. Applications that imaginary objects have, include combining small objects into larger aggregate ones, the decomposing large objects into smaller ones, and performing sophisticated restructurings that turn values into objects, and vice versa. Imaginary objects can also be used to create an object-oriented view of a relational database. This can be done by assigning object identifiers to the tuples of the relational database.

Although imaginary objects turn out to play an important role in increasing the flexibility of the model, there are certain problems related to using them. These problems include updating views that contain imaginary objects, specifying their structure, and guaranteeing that each imaginary object is assigned the same object identifier every time it is used. We will look more closely into these problems after describing how imaginary objects are created in our model.

Creating Imaginary Objects

In order to generate new objects, we explicitly denote their structure and specify the values of their attributes. The syntax for the object creation command is the one presented bellow:

class <*imaginary*-class>
    includes imaginary (*select* $[attr_1 : V_1, attr_2 : V_2, ..., attr_n : V_n]$
        from <*var*> *in* <*source*-class>
        where <*predicate*>)

The selection query included within the parentheses returns a set of tuples, where each tuple represents the attributes of some object. After the tuples are generated, object identifiers are assigned to them to produce objects. Each tuple is assigned a different and unique object identifier. The new objects are then used to populate the virtual class <*imaginary*-class>. We call this class *imaginary*, since it includes only imaginary objects. The attributes of the new objects are those defined within the brackets and are called *core* attributes. Imaginary classes can be treated like any other class. Other attributes can, therefore, be defined for <*imaginary*-class>. These attributes are *virtual*.

As an example of object creation we consider grouping person objects into larger, family objects [AB91]. Each family object will have two *core* attributes *Husband* and *Wife*. This restructuring is done by the following rule:

$$
\begin{array}{l}
\textit{class Family} \\
\quad \textit{includes imaginary (select } [Husband : H, Wife : W] \\
\qquad\qquad\qquad \textit{from } P \textit{ in } Person \\
\qquad\qquad\qquad \textit{where } H.Sex = \text{“}male\text{”} \\
\qquad\qquad\qquad\quad \textit{and } H.Spouse = W)
\end{array}
$$

With this definition, the user is able to view the data not just as a collection of people, but as a collection of families, each consisting of a husband and a wife. Children can also be added to the family as a virtual attribute *Children* in the following manner:

$$
\begin{array}{l}
\textit{attribute Children in class Family} \\
\quad \textit{has value (select } P \textit{ from } Person \\
\qquad\qquad \textit{where } P \textit{ in } Husband.Children \\
\qquad\qquad\quad \textit{or } P \textit{ in } Wife.Children)
\end{array}
$$

However, deciding whether an attribute should be *virtual* or *core* is not always easy, as we shall see when we consider updates.

Besides composing/decomposing large objects from/into smaller ones, imaginary objects can also transform values into objects. If, for example, the user wants to view a person's address not as a collection of different attributes, e.g., *City*, *Street*, *No*, but as a single attribute *Address*, we can group the values of the former three attributes, in order to create new address objects:

$$
\begin{array}{l}
class\ Address \\
\quad includes\ imaginary\ (select\ [City : P.City, Street : P.Street, No : P.No] \\
\qquad\qquad\qquad\qquad from\ P\ in\ Person).
\end{array}
$$

The new objects populate the imaginary class *Address*, and can be used to define a new *Address* attribute in class *Person*, as follows:

$$
\begin{array}{l}
attribute\ Address\ in\ class\ Person \\
\quad has\ value\ (select\ the\ A\ from\ Address \\
\qquad\qquad where\ A.City = City \\
\qquad\qquad\quad and\ A.Street = Street \\
\qquad\qquad\quad and\ A.No = No).
\end{array}
$$

In this way, the programmer can transform values/attributes into imaginary objects, adapting the database to the users' needs.

Issues Arising From Object Creation

The previous examples clearly demonstrate the power and flexibility that object creation adds to a view mechanism. However, object generation raises certain important issues, which we now discuss.

The first issue, concerns the identity of imaginary objects and how this identity remains consistent throughout the lifetime of the imaginary object. As we shall see in the next chapter, the object of imaginary objects are computed and recomputed on demand. Each time a query that generates new objects is posed to the system, it generates a set of tuples and assigns an object identifier to each tuple, in order to create a new object. However, this approach does not guarantee that at each invocation a specific object, O, will be assigned the same identifier. This problem has implications for query processing. For example, the query:

$$
\begin{array}{l}
select\ F\ from\ Family \\
where\ F.Husband = \text{“}Bill\text{”} \\
\quad and\ F\ in\ (select\ F\ from\ Family \\
\qquad\qquad where\ F.Age > 22)
\end{array}
$$

uses two invocations of $Family$ objects (i.e., $select\ F\ from\ Family$) and then intersects the objects produced by each invocation. There is no guarantee, however, that an object, O, will have the same object id in both invocations. Therefore, the intersection might be performed incorrectly. In fact, if the second invocation generates new object identifiers, then the set of objects resulting from the above query will be empty. On the other hand, rewriting the query as:

$$
\begin{array}{l}
select\ F\ from\ Family \\
where\ (F.Husband = \text{“}Bill\text{”}\ and\ F.Age > 22)
\end{array}
$$

would generate one object for each family satisfying the where clause.

A second issue related to imaginary objects concerns view updates. In the context of views, there are two directions in which updates can propagate [SLT91]:

- updates on base objects should propagate to the corresponding view objects, and

- updates on view objects should propagate to the underlying base objects, whenever possible.

The latter is difficult to accomplish in the relational context. Since, "object identity" corresponds to key attribute values, only views containing a key may be updated [SLW88]. On the other hand, in the object-oriented case, when only object-preserving operations are used, then updates seem to propagate in a natural way [SLT91]. However, updating in the presence of imaginary objects is not quite so straightforward.

As we have already seen, to create imaginary objects, we assign object identifiers to tuples of values. That is, we map tuples to object identifiers. From an implementation point of view, we could represent this mapping as a table, where each distinct tuple corresponds to a different object identifier. In this way, each tuple would have the same object identifier, throughout its life as an imaginary object. The problem of updating the database then becomes the problem of updating this mapping.

The way in which updates on database objects are reflected onto imaginary view objects is affected in large by the choice of the core attributes. In fact, the core attributes play a central role, because when the value of a core attribute is modified, then the value of the corresponding tuple in the tuple-oid mapping table changes. Hence, the system assumes that it is dealing with a new object. For example, consider a database for the Ministry of Transportation. To represent vehicle ownerships, the database has a class called $Ownership$ with attributes such as $Ownership\_No$, $Vehicle\_Model$, $Vehicle\_Color$, $Vehicle\_LicenceNo$, $Owner\_Name$, $Owner\_Address$, and $Owner\_DateOfBirth$. We define a view in which owners are represented as objects. Thus, we create a virtual class called $Owner$ with attributes $Name$, $Address$, $DateOfBirth$ and $Ownership$. Then in class $Ownership$, the attributes $Owner\_Name$, $Owner\_Address$, and $Owner\_DateOfBirth$ are replaced by a single attribute $Owner$. This view is defined by the following code:

$create$-view $Vehicle\_Owners$

$import\ class\ Ownership\ from\ database\ MinOfTransportation$

$class\ Owner\ includes\ imaginary$

$(select\ [Name : X.Owner\_Name,$

$Address : X.Owner\_Address,$

$DateOfBirth : X.Owner\_DateOfBirth,$

$Ownership : X]$

$from\ X\ in\ Ownership);$

$attribute\ Owner\ in\ class\ Ownership\ has\ value$

$(select\ the\ O\ from\ Owner\ where\ O.Ownership = self);$

$hide\ attributes\ Owner\_Name, Owner\_Address, Owner\_DateOfBirth$

$in\ class\ Ownership;$

end-view

Now, if Bob is an owner and his address is updated, then a new object with a new identifier will be created in the view. Bob's ownership will refer to this new object. In other words, the system will see Bob before moving and after moving as two different owners. The reason is that *Address* is defined as a core attribute, meaning that *Address* is crucial to the identity of an owner. It would be better to define *Address* as a virtual attribute, and therefore, declare it as not crucial to the identity of the owners.

The previous example clearly demonstrates the need to choose the core attributes carefully, since updates can affect object identity. Indeed, the core attributes should be thought of as being somewhat immutable. Attributes of an imaginary object which might be affected by updating this object, should be defined as virtual attributes, rather than as core attributes.

Chapter 4

The Underlying F-logic Implementation

4.1 Getting to Know *F-logic*

Frame Logic (abbr. *F-logic*) [KLW93] is a formalism which accounts in a clean, declarative fashion for most of the structural aspects of object-oriented languages. These features include object identity, complex objects, inheritance, polymorphic types, methods, encapsulation and others. Moreover, *F-logic* combines the deductive and object-oriented paradigms into one coherent framework, overcoming the so-called *impedance mismatch* problem [Ban88]. At the same time it provides logical semantics for the object-oriented model and allows the definition and manipulation of database schema and types. *F-logic* has a model-theoretic semantics and a sound and complete resolution-based proof procedure. It represents object-oriented concepts, both semantically and syntactically. In this section, we give a general and informal overview of *F-logic*, while in Section 4.2 we use *F-logic* as the foundation for our view mechanism.

4.1.1 Preliminaries

F-logic organizes classes and individual objects into. Such hierarchies are comprised of subclass as well as class membership relationships. Classes are *reified*, meaning that there is no distinction between classes and objects. This gives *F-logic* a great deal of uniformity, making it possible to manipulate classes and objects in the same language.

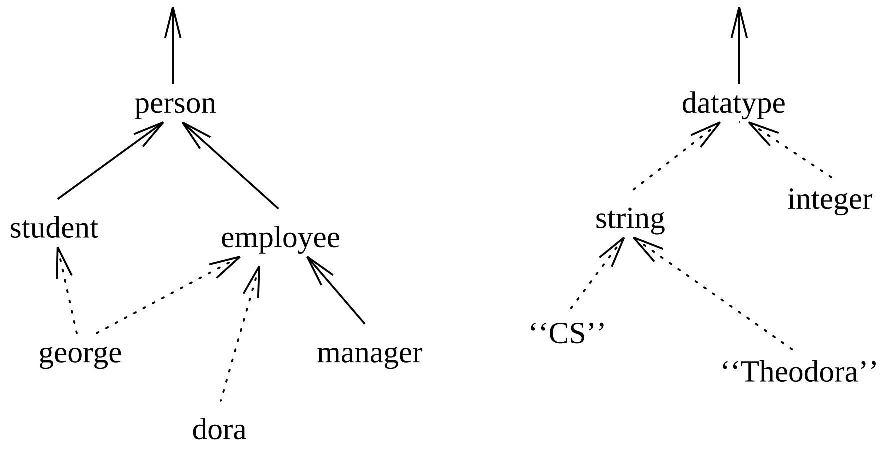


Figure 4.1: Example *F-logic* Is-A Hierarchies

Classes can therefore be members of other classes, by viewing classes as objects. Examples of *F-logic* hierarchies are shown pictorially in Figure 4.1.

Syntactically, a subclass relationship is denoted by '::' (and a solid arc pictorially), while a class membership relationships is denoted by ':' (and a dotted arrow pictorially). For instance, the fact that $employee$ is a subclass of $person$ is represented by the expression $employee :: person$, while the fact that $string$ is a member of class $datatype$ is represented by $string : datatype$. Note how $string$ is a class itself, since it includes individual objects, like "CS" and "$Theodora$".

4.1.2 An Example

An example of an *F-logic* database is given in Figure 4.2. It consists of database facts, general class information, deductive rules, and queries.

Facts are statements that provide information about specific objects of the *F-logic* database. The second fact, for example, means that $dora$ is an $employee$, whose name is "$Theodora$". This is because along with attributes $name$ and $hobbies$, applicable to $person$ objects (Figure 4.2, (iii)), $dora$ has a method $worksFor$. This method applies to $employee$ objects. Therefore, $dora$ is a $person$, and more specifically, an $employee$. The

Database Facts :

$(i)\ george[name \rightarrow \text{“}George\text{”}; inst \rightarrow \text{“}UofT\text{”}; advisor \rightarrow tony[name \rightarrow \text{“}Tony\text{”}]]$

$(ii)\ dora[name \rightarrow \text{“}Theodora\text{”}; hobbies \rightarrow \{\ partying, fashion\ \}\ ;$
$\quad worksFor \rightarrow ai[name \Rightarrow \text{“}Artisti\text{”};\ dealsWith \rightarrow \text{“}FashionClothes\text{”}]]$

General Class Information :

$(iii)\ person[name \Rightarrow string; hobbies \Rightarrow\!\!\!> hobby; children \Rightarrow\!\!\!> child(person)]$

$(iv)\ employee[worksFor \Rightarrow company; salary \Rightarrow integer]$

$(v)\ student[inst \Rightarrow institution; advisor \Rightarrow faculty;\ project@person \Rightarrow\!\!\!> report]$

$(vi)\ company[name \Rightarrow string; location \Rightarrow country; dealsWith \rightarrow merchandise]$

Deductive Rules :

$(vii)\ E : employedStudent\ \leftarrow\ E : employee\ \wedge\ E : student$

$(viii)\ X[specialPrj \twoheadrightarrow Y]\ \leftarrow\ X : student\ \wedge\ X[advisor \rightarrow A; project@A \twoheadrightarrow Y]$

Queries :

$(ix)\ ?-\quad dora[salary \rightarrow X]$

$(x)\ ?-\quad X : student\ \wedge\ X[inst \rightarrow \text{“}UofT\text{”}; project@tony \twoheadrightarrow Y]$

Figure 4.2: An Example *F-logic* Database

term *dora* is a logical id of an object that represents a person. However, "*Theodora*" is a member of the class *string*; it represents the value of one of *dora*'s attributes, her *name*. Additional information provided by this fact is that *dora* works for a company with logical id *ai*. The name of the company is denoted by object id "*Artisti*", and it deals with "*FashionClothes*".

Statements (iii) to (v) give a general description of the *F-logic* classes. Each class has a *signature* which specifies the names, the types of the arguments, and the types of the results of each method applicable to the class (attributes are considered to be methods with no arguments). For instance, statement (v) says that for every object in class *student*, attributes *studNo*, *instit* and *advisor* must be of type *integer*, *institution* and *faculty* respectively. The expression $project@person \Rrightarrow report$ describes a method *project*, that expects one proper argument of type *person* and returns a set of elements of type *report*. Attributes and methods are also viewed as objects. As a result of this, names can be returned as query answers. Hence, schema information is transformed into data so that it can be manipulated in the same language.

Double-headed arrows, $\twoheadrightarrow$ and $\Rrightarrow$, are used for set-valued attributes, while single headed ones, $\rightarrow$ and $\Rightarrow$, are used to signify scalar attributes. Also, double-shafted arrows, $\Rightarrow$ and $\Rrightarrow$, specify types, while single shafted ones specify values. If a double-shafted arrow is not given for some method, then the corresponding single shafted one cannot be used. For each statement of the type $obj[attr \rightarrow ...]$, where $obj : class$, there must be a statement of the form $class[attr \Rightarrow ...]$. Moreover, given $class[attr \Rightarrow ...]$, one cannot use $obj[attr \twoheadrightarrow ...]$, where $obj : class$. A statement of the type $class[attr \Rightarrow (a, b, c)]$ means that attribute *attr* when applied to objects of class *class* has a type, which is the conjunction of types a, b, and c in the parentheses.

The term $children \Rrightarrow child(person)$ in statement (iii) specifies a type constraint for the attribute *children*. Here, *child* is a unary function symbol and *person* is a constant denoting a class. The term *child(person)* is the logical id of a new class created by application of the function symbol *children*( ) on *person*. Hence, function symbols are used as object and class id's constructors. As we will see later on, function symbols play a major part in the (implementation) of our view mechanism.

Statements (*vii*) and (*viii*) are *F-logic* deductive rules, which derive additional information from already existing database information. For instance, the first of these rules creates a new class with object id *employedStudent* and populates it with objects belonging in both *employee* and *student* classes. Note that this rule does not mention anything about the properties that the new class *employedStudent* will have. However, this can be taken care of by explicitly defining *employedStudent* to be a subclass of both *employee* and *student*, i.e.,

$$employedStudent :: employee \ \wedge \ employedStudent :: student.$$

Then *employedStudent* will inherit the properties of both *employee* and *student*. Another way for dealing with this situation will be presented in Section 4.2.5, where we derive virtual classes as intersections of source classes. The second deductive rule, *specialPrj* for student objects, which returns a set of *reports* written in common with the *advisor* of the student. Query (*ix*) asks for the salary of *dora*, while query (*x*) inquires about those "*UofT*" students who have completed a project of type *report* with *tony*.

In *F-logic* the notions of well-typing and type error have a precise *model-theoretic* meaning. We say that a signature atom $c[m@b_1, ..., b_k \Rightarrow ...]$ *covers* a data atom $o[m@a_1, ..., a_k \rightarrow v]$ if for each $i = 1, ..., k$, we have $o : c$ and $a_i : b_i$. The same is true for set-valued methods (replacing $\Rightarrow$ and $\rightarrow$, by $\Rightarrow\!\!\!\!\Rightarrow$ and $\twoheadrightarrow$, above). Every data expression, either stored or derived, must be covered by a signature. Moreover, every expression must satisfy all the constraints imposed by the signatures that cover that expression. Another aspect of the typing system is that signature declarations are enforced. Only methods which are defined explicitly within a class, or that are inherited from superclasses of the class, can be applied to objects in the class. For instance, the only methods applicable to objects in class *employee* in Figure 4.2 are *name*, *hobbies*, and *children* inherited from class *person*, and *worksFor* and *salary*, which are explicitly defined in *employee*.

Predicates can be represented in *F-logic* in the following manner. To encode a n-ary predicate symbol, p, a new class is introduced. This new class will be denoted by the same symbol, i.e. p, for convenience. Let *p-tuple* be a new n-ary function symbol. Then $(\forall X_1 ... \forall X_n)(p\text{-}tuple(X_1, ..., X_n) : p)$ is asserted, and classical atoms of the form $p(T_1, ..., T_n)$ are represented as molecules of the form:

$$(1) \qquad p\text{-}tuple(T_1, ..., T_n)[arg_1 \rightarrow T_1; ... : arg_n \rightarrow T_n].$$

In the above, $p\text{-}tuple(T_1, ..., T_n)$ is value-based, that is, it is fully dependent on the value of its attributes. The term $p\text{-}tuple(T_1, ..., T_n)$ can be seen as an object id of the object to be used to represent a p-relationship among $T_1, ..., T_n$, if such a relationship exists. Statement (1) above *asserts* that there is an actual p-relationship among $T_1, ..., T_n$.

4.1.3 The Syntax

If $\mathcal{L}$ is an *F-logic* language, then its alphabet consists of:

- a set of *object constructors*, $\mathcal{F}$;
- an infinite set of *variables*, $\mathcal{V}$;
- auxiliary symbols: $(,), \rightarrow, \twoheadrightarrow, \Rightarrow, \Rrightarrow$, etc.; and
- logical connectives and quantifiers: $\vee, \wedge, \neg, \leftarrow, \forall, \exists$.

Object constructors are treated as function symbols. The *arity* of a function symbol is a nonnegative integer that determines how many arguments the symbol can have. Symbols of arity 0 denote constants, while those with arity ≥ 1 are used to construct larger terms out of simpler ones. An *id-term* is a first-order term composed of function symbols and variables, as in predicate calculus. We use $U(\mathcal{F})$ to denote the set of all *ground* (i.e. variable-free) id-terms. $U(\mathcal{F})$ is called the *Herbrand Universe*.

At the conceptual level, ground id-terms are used to denote *logical object-ids*. A logical object-id is a logical abstraction of the concept of *physical object identity* [KC86, AK89]. Hence, for example, *person* is the *oid* of the class denoting all people. Henceforth, a symbol that begins with a lower-case letter denotes a ground id-term, while one beginning with a capital letter denotes an id-term that may be non-ground.

An *F-logic* language consists of a set of formulae constructed out of the alphabet symbols. Formulas are built out of simpler formulas by using logical connectives and quantifiers. The simplest kind of formulas are called *molecular* F-formulas (abbr. *F-molecules*, *molecules*).

Definition4.1(*Molecular Formulas*)
A *molecule* in *F-logic* is one of the following statements:

(i) An *is-a assertion* of the form $C :: D$ or $O : C$, where C, D, and O are id-terms.

(ii) An *object molecule* of the form O[list of *method expressions*].

A *method expression* can be either a *non-inheritable data expression*, or a *signature expression*.

- A *non-inheritable data expression* is either:

A non-inheritable *scalar* expression ($k \geq 0$):

$$ScalarMethod@Q_1, ..., Q_k \rightarrow T$$

A non-inheritable *set-valued* expression ($l, m \geq 0$):

$$SetMethod@R_1, ..., R_l \twoheadrightarrow \{S_1,\ ...,\ S_m\}$$

– A *signature expression* is either:

A *scalar* signature expression ($n, r \geq 0$):

$$ScalarMethod@V_1, ..., V_k \Rightarrow (A_1, ..., A_r)$$

A *set-valued* signature expression ($s, t \geq 0$):

$$SetMethod@W_1, ..., W_s \Rrightarrow \{B_1,\ ...,\ B_m\}$$

In (i), $C :: D$, states that C is a *nonstrict* subclass of D, which implies that $C :: C$ is a valid is-a assertion. On the other hand, $O : C$, states that O is a member of class C.

In (ii), O is an id-term that denotes an object. $ScalarMethod$ and $SetMethod$ are also id-terms, respectively indicating scalar and set-valued methods invoked on O. Double-headed arrows $\twoheadrightarrow$ and $\Rrightarrow$, indicate that $SetMethod$ is a set-valued function, while single headed ones, $\rightarrow$ and $\Rightarrow$, indicate $ScalarMethod$ to be scalar. In data expressions, the terms T and S_i are id-terms that represent output of the respective methods, when invoked on O with attributes $Q_1, ..., Q_k$ and $R_1, ..., R_l$, respectively. The arguments are id-terms too. In signature expressions, A_i and B_j are id-terms that represent *types* of the results returned by the method. The notation $(A_1, ..., A_n)$ denotes that the output of the method must belong to *all* the classes listed within the parentheses.

The order of data and signature expressions is immaterial. Furthermore, "@" will be omitted whenever the corresponding method does not expect any arguments (i.e. it is an attribute). For instance, $P[Mthd@\!\!\rightarrow Val]$ will be written $P[Mthd \rightarrow Val]$. The '{}' will be omitted whenever they contain only one element, and the same applies for '( )'.

Is-a assertions are *atomic* and cannot be decomposed further into simple formulas. On the other hand object molecules are not always atomic, so they can be decomposed into a conjunction of their *atoms*. For instance, $X[attr_1 \rightarrow a; meth@arg \rightarrow Y]$ is equivalent to $X[attr_1 \rightarrow a] \wedge X[meth@arg \rightarrow Y]$.

F-formulae are built up from simpler F-formulae by means of logical connectives and quantifiers:

- molecular formulae are F-formulas;
- $\alpha \vee \beta, \alpha \wedge \beta, \neg\alpha$ are F-formulae, if α and β are;
- $\forall X\ \alpha, \exists Y\ \beta$ are are F-formulae, if α and β are, and X, Y are variables.

A *literal* is either a molecular formula or a negated molecular formula.

In the sequel we will be combining molecules in order to achieve compaction. For instance,

$$Q : P \ \wedge\ Q[meth_1@X \rightarrow Y] \ \wedge\ Y : S \ \wedge\ Q[meth_2@W, Y \Rightarrow (Z, T)] \ \wedge\ Z : R$$

will be written as

$$Q : P[meth_1@X \rightarrow (Y : S); meth_2@W, Y \Rightarrow (Z : R, T)]$$

and

$$P[meth_1@X \rightarrow Q[meth_2@W \twoheadrightarrow \{T, S\}]]$$

will account for

$$P[meth_1@X \rightarrow Q] \ \wedge\ Q[meth_2@W \twoheadrightarrow \{T, S\}].$$

4.2 View Creation Using *F-logic*

In the previous chapter we demonstrated how views are created within the context of our model. We presented the syntax and semantics of all the operations that account for view creation. Each of these operations can be translated into one or more *F-logic* rules that perform the same derivation. We can therefore use the higher level view generation query language of the previous chapter as a user interface, and use *F-logic* as the underlying implementation.

4.2.1 Importing

As we saw in the previous chapter, a view is thought of as a database that imports all its data from other databases. This is performed by using an import mechanism that imports specific classes, along with all of their subclasses, into the view. In this section we describe how such a mechanism can be simulated in *F-logic*. More specifically we will present *F-logic* rules that perform imports and discuss the issues that arise.

In chapter 3, creating a view v by importing a class $<class>$ from the base schema BS was performed by the following (higher level) command:

$$\begin{array}{l} \textit{create-view } v \textit{ as} \\ \quad \textit{import class } <class> \textit{ from } BS \end{array}$$

The result of executing this command is shown in Figure 4.3. Base class, $<class>$, is imported into the view, v, along with all its subclasses. In the context of *F-logic* this import is translated into the following rules:

(1) $v(C1) :: v(C2) \leftarrow C1 :: C2 \wedge\ C1 :: <class> \wedge C2 :: <class>$

(2) $v(O) : v(C)\ \leftarrow\ O : C\ \wedge\ C :: <class>$

(3) $v(C)[X \Rightarrow T]\ \leftarrow\ C[X \Rightarrow T]\ \wedge\ C :: <class>$

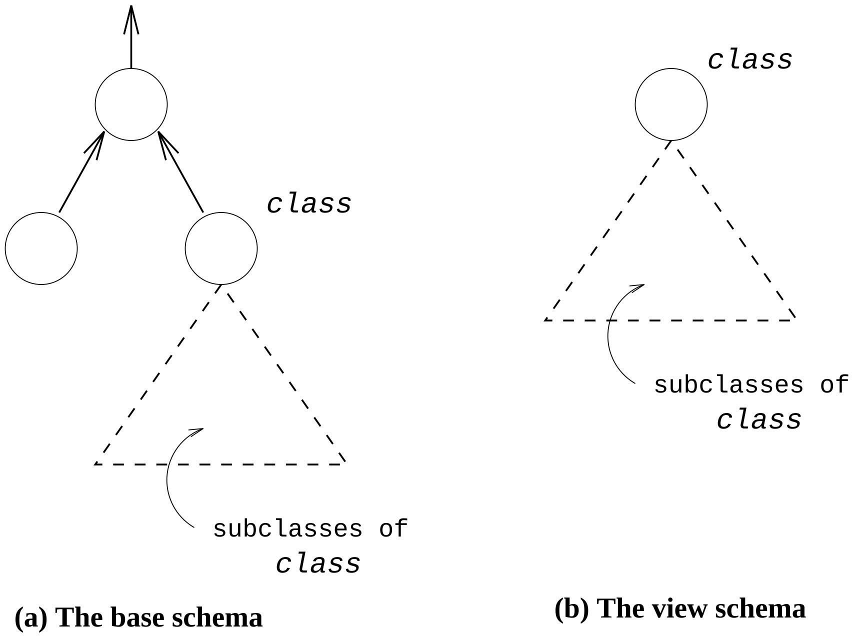


Figure 4.3: Importing Base Class <*class*>

$$(\mathbf{4}) \qquad v(C)[X \Rightarrow\!\!\!\Rightarrow T] \;\leftarrow\; C[X \Rightarrow\!\!\!\Rightarrow T] \;\wedge\; C :: <\!class\!>$$

$$(\mathbf{5}) \qquad v(O)[X \rightarrow V] \;\leftarrow\; O : C[X \rightarrow V] \;\wedge\; C :: <\!class\!>$$

$$(\mathbf{6}) \qquad v(O)[X \twoheadrightarrow V] \;\leftarrow\; O : C[X \twoheadrightarrow V] \;\wedge\; C :: <\!class\!>$$

Rule (1) starts the procedure by importing <*class*> and all its subclasses into the view, while at the same time retaining the formation of the hierarchy. This is done by using the object constructor function v, which will create *object-ids* for the new view classes. Note that this object id creation is performed dynamically, every time the rule is executed. Each class, $v(C)$, in the view schema is *linked* to class, C, of the base schema. Moreover, if class $C1$ is a subclass of class $C2$ in the base schema, then in the view schema, class $v(C1)$ will be a subclass of $v(C2)$. This feature ensures that the hierarchy is correctly preserved when importing the subclasses of <*class*> in the view. Note the use of the :: <*class*> part in the body of the rule. This works as an upper bound, in order to prevent the insertion of any other superclasses of, or classes not related to <*class*>. By simply changing <*class*>, we can dynamically change the hierarchy that is being imported.

The second rule takes care of the population of the newly imported classes. This is in the same flavor as the first rule. The v function is used again to assign object identifiers to the new view objects. If an object, O, is in base class C, then the corresponding view object, $v(O)$, will be in view class $v(C)$. Again the conjunct $C :: <class>$ is used to assure that only classes belonging in the subgraph of the schema graph rooted at class $<class>$ will be considered.

Rules 3 and 4 are the typing rules. Once we have imported the desirable class hierarchy and populated the view classes with objects, we want to be able to apply attributes and methods to them. More specifically, object $v(O)$ in view class $v(C)$ corresponds to object O in class C. Therefore, $v(O)$ must have the same set of attributes and methods as those of O. Hence, if class C has a scalar or set-valued attribute (method), then by rules 3 and 4 we infer that this attribute (method) must also be defined in the class $v(C)$. Note that these rules are required in order to talk about values of the attributes (or methods) of the imported classes.

Finally, rules 5 and 6 account for importing the values of the properties of the corresponding view objects. If object O has property X with value V, then the corresponding view object $v(O)$ will have the same property X with the same value V applied to it as an object of class $v(C)$. Rule 5 accounts for the scalar properties, while rule 6 is used for the set-valued ones.

One might think that using all the above rules for a single import of a base class into the view is quite cumbersome. However, one should keep in mind that whenever importing a class into the view, all its subclasses are imported as well. Hence, the aforementioned rules can account for a large number of classes to be imported, i.e., $<class>$ and all its subclasses. Moreover, to import a superclass $<supclass>$ of $<class>$ at some later moment, we simply need to replace $<class>$ with $<supclass>$ in all the *F-logic* import rules above. By changing <*class*> to <*supclass*> in the above rules, <*supclass*> will be the base class to be imported along with all its subclasses (<*class*> included).

In the sequel, we will use the term <*view*-name>(<*class*>) to denote the class in view <*view*-name>, resulting from importing class <*class*> from some database. On the other hand, when a new class, <*newClass*>, is derived within the view, it will be prefixed with the view's name, i.e., <*view*-name>_<*newClass*>. For example, importing class *employee* from a database into view *view*1 will produce class *view*1(*employee*), whereas selecting all female employees within the view will result in a class *view*1_*femaleEmployee*. Prefixing classes with the name of the view in which they are derived, ensures that a different object id will be assigned to them so that they remain distinct. Therefore, *femaleEmployee* in *view*1 will be a different class from *femaleEmployee* in *view*2. A further discussion on the issue is presented in Section 4.2.3.

4.2.2 Hiding

According to the semantics of the *hide* operator presented in Section 3.3.2, hiding a property from a class is performed explicitly. The property is hidden in the class and in all its subclasses. To our knowledge, other approaches to object-oriented views [SLT91, Run92b], when hiding a property in a class, create a new virtual class identical to the source class, but without the hidden property. In these approaches, hiding a property in a class and all its subclasses requires a separate hiding command for every class in which the property is to be hidden. Clearly this is a less general and more cumbersome approach than the one we follow here.

The way we perform our hiding technique in *F-logic* is by not importing into the view all those properties that are to be hidden. This is done by adding the appropriate constraints to the import rules at view creation time. Suppose we wanted to hide the *salary* of all *employee*s. In general we do not want to just hide *salary* in the class of *employees*, but we also want to hide the salaries of objects in subclasses of *employee*, e.g., *managers* and *technicians*. This can be performed by importing class *employee* and blocking at the same time the import of the attribute *salary*. All that is needed is the addition of a constraint $X \neq salary$ to import rules 3 and 5, of Section 4.2.1, as shown bellow:

$$(\mathbf{3}) \qquad v(C)[X \Rightarrow T] \;\leftarrow\; C[X \Rightarrow T] \;\wedge\; C :: \text{<}class\text{>} \wedge X \neq salary$$

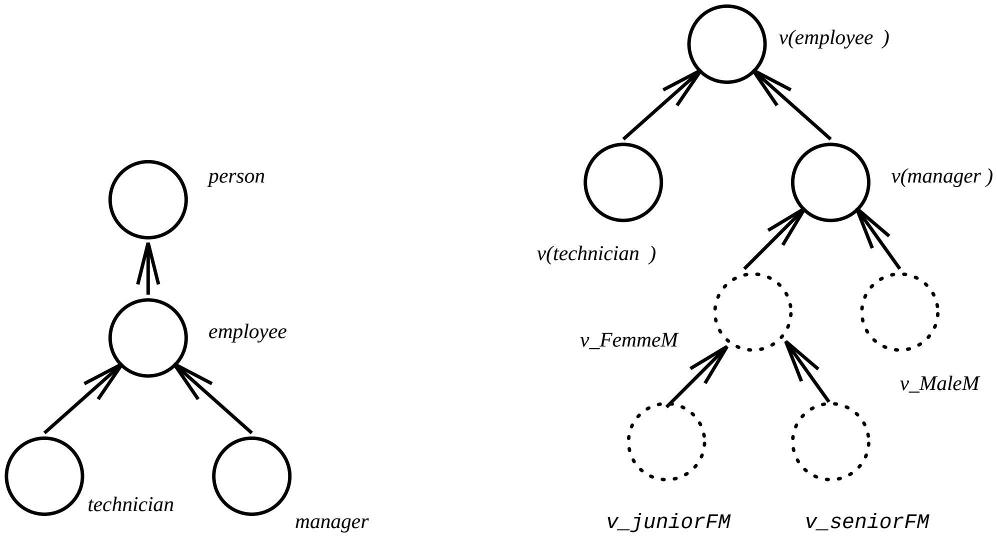
$$(\mathbf{5}) \qquad v(O)[X \rightarrow V] \;\leftarrow\; O : C[X \rightarrow V] \;\wedge\; C :: \text{<}class\text{>} \wedge X \neq salary$$

When *employee* is imported, its subclasses are also (automatically) imported. The above two rules hide the attribute *salary* from the imported classes. Rule (3) passes to view class, $v(C)$, the type definition of every property of base class C, except for *salary*. On the other hand, rule (5) prevents the values of attribute salary of the base objects from being imported into the view. Therefore, *salary* as defined in the database is no longer visible from within the view.

Once we have hidden the *salary* attribute by blocking its import, we can proceed to redefine *salary* with a different type within the view. This kind of overriding is a required feature for object-oriented models. It allows a great deal of functionality, since not only can we hide an attribute from a group of users, but we can also present different interpretations of it to different users. To demonstrate this functionality, consider the case, in which we would like a specific group of users to view *salary* as an abstract level denoted by some letter. Then, if salaries over 60,000 are to be denoted by "*A*", this could be done by importing *employee*, blocking *salary* as above and then defining *salary* within the view as follows:

$$v(employee)[salary \Rightarrow letter] \;\leftarrow\; employee[salary \Rightarrow integer]$$

$$v(O) : v(employee)[salary \rightarrow \text{``}A\text{''}] \;\leftarrow\; O : employee[salary \rightarrow S] \wedge S > 60,000.$$



(a) The base schema BS

(b) Create view v1 as
import class employee from BS
{ additional class derivations }

Figure 4.4: Specializing Within the View

This definition, according to *F-logic* type inheritance, is inherited by all subclasses of $v(employee)$, in which the original definition of salary is also hidden.

4.2.3 Specialization

Specialization is a way of deriving view classes by populating them with objects of source classes. It is, more specifically, a set manipulation operator that selects a set of objects from a class. It resembles the selection operation of relational algebra [Ull88, Dat90]. The view classes created by specialization can be further specialized.

Figure 4.4 shows a view created by initially importing class *employee* from the base schema. All subclasses of *employee* are imported as well. We then go on to derive a new view class v_FemmeM by populating it with all the object instances in view class $v(manager)$, whose *sex* attribute has a specific value, namely, "Femme". In the same manner v_MaleM will be populated with those (view) objects with an attribute sex of value "Male". The *F-logic* rules which perform these specializations are presented bellow:

$$O : v\_FemmeM \leftarrow O : v(manager)[sex \rightarrow \text{“}Femme\text{”}]$$
$$O : v\_MaleM \leftarrow O : v(manager)[sex \rightarrow \text{“}Male\text{”}]$$

We use a prefix, $v\_$, for each class derived within the view v in order to differentiate from classes having the same name but derived in other views. Such classes need not have the same type and/or the same extent. Class $FemmeM$ derived in view v_1 may be different than class $FemmeM$ derived in view v_2. Prefixing each of these classes with their view name ensures that a different object id will be assigned to each so that they remain distinct.

Once v_FemmeM is derived, it can be treated as any other view class so we can perform further specialization. Grouping female managers into two subclasses according to their age could be done by the following specialization rules:

$$O : v\_juniorFM \leftarrow O : v\_FemmeM[age \rightarrow V : over40]$$
$$O : v\_seniorFM \leftarrow O : v\_FemmeM[age \rightarrow V : under41]$$

where $over40$ and $under41$ are classes of integer objects ranging respectively from 41 to some reasonable upper bound, e.g., 80, and from 40 to some reasonable lower bound, e.g., 16. Note how the value V of attribute age is not specific, but falls within a range of different values. Classes $over40$ and $under41$ can be base classes, already populated with all the (integer) objects that they describe. That is, class $over40$ would be populated with all the integer numbers in the domain (40,80), and class $under40$ with all integers in (16,40).

The above discussion indicates that terms like $V : over40$ and $V : under41$ are a convenient device for simulating arithmetic comparison operators within the *F-logic* context. For instance, the term $V : over40$ in the previous example actually accounts for subgoals $V > 40$ and $V < 80$. Hence, the first of the above two rules could also be

$$O : v\_juniorFM \;\leftarrow\; O : v\_FemmeM[age \rightarrow V] \wedge V > 40 \wedge V < 80.$$

However, using the term $V : over40$ instead of $V > 40 \wedge V < 80$, provides a greater degree of uniformity, since in this case we do not have to extend *F-logic* to accommodate arithmetic comparison operators.

The class resulting as a specialization of some source class is a subclass of the source class. Its extent is therefore a subset of the extent of the source class, while its type is the same as the type of the source class, since no properties are hidden or added.

4.2.4 Generalization

Generalization combines two or more classes into a superclass. Every object in the extent of any of the source classes becomes a member of the extent of the new superclass. The type of the new class contains only those properties common to all source classes. The following rules show an example of generalization in the context of *F-logic*.

$$S : v\_ship \;\leftarrow\; S : v(tanker)$$
$$S : v\_ship \;\leftarrow\; S : v(frigate)$$
$$v\_ship[X \Rightarrow T] \;\leftarrow\; v(tanker)[X \Rightarrow T_1] \wedge v(frigate)[X \Rightarrow T_2] \;\wedge\; lub(T_1, T_2, T)$$

The first two rules populate the new class with all objects contained in the source classes. Objects belonging in either of the two classes $v(tanker)$ and $v(frigate)$ will also belong to v_ship. For instance, the object $ionianSea$ of class $v(tanker)$ will also be an object of the new class, v_ship, even if it is not a member of class $v(frigate)$. The third rule ensures correct typing. The new view class *v_ship* has a property X only if both source classes have X. The conjunct $lub(T_1, T_2, T)$ is a predicate that given types T_1 and T_2 calculates their least upper bound T, which has to be in the database. If T_1 and T_2 do not have a least upper bound, then property X is undefined in *v_ship*. For instance *ionianSea* will have an attribute *length* applied to it only if *length* is applicable in both $v(tanker)$ and $v(frigate)$. If the type of *length* is *real* in one source class and *integer* in the other, then the type of *length* in the new class *v_ship* will be *real*. Attributes not applicable to both source classes, e.g., attribute *armament* of class $v(frigate)$, will not be passed over to the new class *v_ship*.

Note that the view class need not be a union of only two source classes. We can extend the above definition in order to produce a class which will be the generalization (union) of n classes:

$$
\begin{aligned}
S : union \;&\leftarrow\; S : v(class_1)\\
S : union \;&\leftarrow\; S : v(class_2)\\
\vdots \quad &\qquad \vdots\\
S : union \;&\leftarrow\; S : v(class_n)\\
union[X \Rightarrow T] \;&\leftarrow\; v(class_1)[X \Rightarrow T_1] \;\wedge\; ... \wedge v(class_n)[X \Rightarrow T_n]\\
&\qquad\qquad\qquad\qquad \wedge\; lub(T_1, ..., T_n, T).
\end{aligned}
$$

We use one population rule for each source class comprising the union, but only one typing rule overall, with one subgoal for each source class. Additionally, the latter rule has a predicate *lub* which returns T as the least upper bound of T_1, ..., T_n.

4.2.5 Intersection

Intersecting two or more classes produces a new view class, containing only those objects that belong in the extents of all the source classes. The new class has all the properties present in any source class. For instance, to intersect two source classes, *class*1 and *class*2, we use the following rules:

$$O : intersect \leftarrow O : class1 \wedge O : class2$$
$$intersect[X \Rightarrow T_1] \leftarrow class1[X \Rightarrow T_1] \wedge \neg\, class2[X \Rightarrow (\ )]$$
$$intersect[X \Rightarrow T_2] \leftarrow class2[X \Rightarrow T_2] \wedge \neg\, class1[X \Rightarrow (\ )]$$
$$intersect[X \Rightarrow (T_1, T_2)] \leftarrow class1[X \Rightarrow T_1] \wedge class2[X \Rightarrow T_2]$$

The first rule populates the new class with the objects common to the two source classes. The three other rules account for correct typing. By rule 2, every property in *class*1 but not in *class*2 is defined to apply in class *intersect*, too. Rule 3 accounts for the properties in *class*2 but not in *class*1. Finally, the fourth rule takes care of those properties that are common to the two source classes, and ensures that the domain of each property in the new class is the intersection of the domains that this property has in the two source classes.

Again, as with generalization, a virtual class can be an intersection of more than two classes. We have only one population rule, but we have multiple typing rules which would have to account for all the possible combinations of properties common to one, two, three, and up to n classes, where n is the number of the source classes being intersected. When $n = 3$, for example, to define the intersection of three classes $c1$, $c2$, and $c3$ we will need eight rules:

$$O : intersect \leftarrow O : c1 \wedge O : c2 \wedge O : c3$$
$$intersect[X \Rightarrow T_1] \leftarrow c1[X \Rightarrow T_1] \wedge \neg\, c2[X \Rightarrow (\ )] \wedge \neg\, c3[X \Rightarrow (\ )]$$
$$intersect[X \Rightarrow T_2] \leftarrow \neg c1[X \Rightarrow (\ )] \wedge c2[X \Rightarrow T_2] \wedge \neg\, c3[X \Rightarrow (\ )]$$
$$intersect[X \Rightarrow T_3] \leftarrow \neg c1[X \Rightarrow (\ )] \wedge \neg\, c2[X \Rightarrow (\ )] \wedge c3[X \Rightarrow T_3]$$
$$intersect[X \Rightarrow (T_1, T_2)] \leftarrow c1[X \Rightarrow T_1] \wedge c2[X \Rightarrow T_2] \wedge \neg\, c3[X \Rightarrow (\ )]$$
$$intersect[X \Rightarrow (T_1, T_3)] \leftarrow c1[X \Rightarrow T_1] \wedge \neg\, c2[X \Rightarrow (\ )] \wedge c3[X \Rightarrow T_3]$$

$$intersect[X \Rightarrow (T_2, T_3)] \leftarrow \neg c1[X \Rightarrow (\ )] \wedge c2[X \Rightarrow T_2] \wedge c3[X \Rightarrow T_3]$$
$$intersect[X \Rightarrow (T_1, T_2, T_3)] \leftarrow c1[X \Rightarrow T_1] \wedge c2[X \Rightarrow T_2] \wedge c3[X \Rightarrow T_3].$$

4.2.6 Increasing the Flexibility

In the previous chapter, we mentioned two features that increase the flexibility of the generalization and specialization constructions, *behavioral generalization* and *parameterized classes*. This section shows how these two features can be implemented in *F-logic* in a natural way.

Behavioral Generalization

Behavioral generalization is a way of populating a virtual class with objects that share common properties. In other words, the extent of a virtual class derived by behavioral generalization includes all those objects which satisfy some specific properties. This is naturally represented in *F-logic* by the following rule:

$$(1) \qquad O : v\_viewClass \leftarrow O : X[p_1 \Rightarrow (\ );\ ...;\ p_n \Rightarrow (\ )]$$

That is, if object O in class X has properties $p_1, ..., p_n$, then O is in the extent of the new class $v\_viewClass$. The *F-logic* notation $O : X[p \Rightarrow (\ )]$ means that object O in class X has property p. If class X is not given in the rule (i.e., we use $O[p \Rightarrow (\ )]$), then the new class will not only include object instances with property p, but it will also include *classes* with property p. This is due to the uniform treatment of classes and object instances by *F-logic*: everything is considered to be an object.

Another issue concerns the types of the properties. Rule (1) does not specify the types of the properties of an object. This can lead to inconsistencies and undesirable results. Consider a situation similar to that in Section 4.2.2, where *salary* of type *integer* is redefined in some classes to be of type *letter*. The behavioral generalization rule

$$(2) \qquad O : v\_viewClass \;\leftarrow\; O[salary \Rightarrow (\;)]$$

populates $v\_viewClass$ with objects having an integer salary *and* with objects having a letter salary, which may not be the user's intention.

One way around this is to assume that each property has only one definition, and therefore, only one type. This is a common assumption in other models, including [SLT91, Run92b]. By using this assumption it would not be possible for us to have a *salary* of type *integer* and a *salary* of type *letter*. We would have to use a different name for defining an abstract salary, e.g., $salaryLevel \Rightarrow letter$. Then, if we wanted to group those objects with a salary of type *integer*, we could use rule (2). On the other hand, to group objects with an abstract salary we could use the following rule:

$$(3) \qquad O : v\_viewClass \;\leftarrow\; O[salaryLevel \Rightarrow (\;)]$$

A second solution is to use an explicit type definition in rule (1) to distinguish between properties. For instance, if we only wanted those objects for which *salary* of type *letter* applies we could state this explicitly, as:

$$O : v\_viewClass \;\leftarrow\; O : X[salary \Rightarrow letter],$$

and avoid any undesirable results.

Parameterized Classes

Besides behavioral generalization, we also employ *parameterized classes* as shown in Section 3.3.6, to enhance the functionality of our model. In that section, we used a single derivation query to partition a class into subclasses according to a parameter. In *F-logic* this sort of derivation can be performed by a rule with syntax:

$$O : <newClass>(<par>) \leftarrow O : <viewClass>[<property>\rightarrow <par>].$$

Such a rule partitions class, $<viewClass>$ into subclasses of the form $<newClass>(<par>)$. For instance, to group $Ford$ cars by the year they were made, we could use the following *F-logic* rule:

$$C : fordCars(Year) \leftarrow C : car[makeYear \rightarrow Year;\ manuf \rightarrow \text{“}Ford\text{”}].$$

This defines a family of derived classes parameterized by year. The extent of each class contains those objects that represent $Ford$ cars made in year $Year$. For instance, $fordCars(1993)$ includes all those $Ford$ cars made in 1993. When the new 1994 cars are produced, they will automatically be grouped into class $fordCars(1994)$ as soon as their $makeYear$ is specified.

Note how easy it is to derive a family of parameterized classes according to more than one parameters. For example, if we wanted to group cars according to both the country and the year they were built in, we could add one more parameter to the rule and get:

$$C : cars(Year, Cntry) \leftarrow C : car[makeYear \rightarrow Year;\ madeIn \rightarrow Cntry].$$

Such a rule would then create classes like $(1992, Canada)$, inhabited by objects representing cars made in Canada in 1992.

Both the above examples demonstrate the power and flexibility, that parameterized classes provide to the view mechanism. By using a single rule, the programmer can create a group of classes that otherwise would require many separate definitions. Furthermore, parameterized classes are *automatically* created (or deleted), when objects are added to (removed from) the database. For example, if cars from 1952 do not exist in the database anymore, then class $Cars(1952)$ is automatically removed from the group of parameterized classes. In a similar fashion, the addition of the new cars of 1995, would automatically group these cars into a newly created class $Cars(1995)$.

4.2.7 Imaginary Objects

Finally we consider view definition commands that generate new objects. The generated objects are said to be *imaginary*, since they did not previously exist in the database [AB91]. As demonstrated in Chapter 3, such objects increase the flexibility of the view mechanism, by allowing sophisticated restructurings that turn objects into values and vice versa. Furthermore, small objects can be combined into larger ones, while large objects can be decomposed into smaller ones.

In our *F-logic* implementation, however, the aforementioned notion of imaginary objects has to be slightly modified. We can no longer characterize all newly generated objects as imaginary. This means that we could have objects not previously existing in the database, which actually do not represent imaginary objects. This is because of the way we create our views, and more specifically, because of the way we implement our import mechanism in *F-logic*.

Recall, that when we import a class C from the base schema, we use a unary function symbol, v, to create a view class, $v(C)$, where v is the view's name. The view class is imaginary in the sense that it did not previously exist in the database. After $v(C)$ is generated, it is populated with objects of the form $v(O)$, where O is an object in base class C. The properties and values of objects in base class C are passed to the corresponding objects in view class $v(C)$. Strictly speaking, both $v(C)$ and its objects are imaginary, since they have new object ids, created by the function symbol v. However, a closer look shows that such classes and objects resemble in an exact way the base classes and base objects from which they are derived. We therefore consider the imported classes and objects to be *replicas* of the corresponding base ones. They have the same state and structure and, moreover, they behave in the same way. In this manner, the import rules of section 4.2.1 can be thought of as "*links*" between the view objects and the corresponding base objects. Any change in the state or behavior of the base object propagates directly to the corresponding view object.

It therefore becomes clear that such objects, though not previously existing in the database, do not actually correspond to the notion of imaginary objects as developed in [AB91]. Consequently, we will modify this notion, and use the term imaginary objects to characterize those objects generated within the boundaries of the view, which normally have a structure different from all other view objects. In the sequel, we will demonstrate how such imaginary objects can be created within our view model. We do this by means of examples, which further clarify the difference between the two types of generated objects in our model.

Imaginary objects are created by defining their type and structure. After they are created, they can be used to populate imaginary classes. For example, combining *person* objects to create *family* objects (Example in section 3.3.7) can be implemented by the following *F-logic* rule:

$$O : v\_family[husb \rightarrow H; wife \rightarrow W] \\ \qquad \leftarrow \; H : v(person)[sex \rightarrow \text{“}Male\text{”}; spouse \rightarrow W]$$

This rule defines a view class v_family populated with objects of the form $O[husb \rightarrow H; wife \rightarrow W]$, where H is a $Male$ $person$, whose $spouse$ is W. Such objects are imaginary, since they did not previously exist in the **view**. It is essential to note that the structure of the new v_family objects does not resemble the structure of any of the other objects previously existing in the view. Well-typing conditions should be ensured by a typing rule covering the new attributes, i.e. $v\_family[husb \Rightarrow person; wife \Rightarrow person]$.

In addition to combining small objects into larger ones, as demonstrated in the previous example, we can also transform values into objects. Referring to the example in Section 3.3.7, we could write in *F-logic*:

$$dora[name \rightarrow \text{``}Dora\text{''}; address \rightarrow [city \rightarrow \text{``}Patras\text{''}; street \rightarrow \text{``}Kalavriton\text{''}]]$$

The value of address, i.e. $[city \rightarrow \text{``}Patras\text{''}; street \rightarrow \text{``}Kalavriton\text{''}]$, is a *complex value* representing the address of the person named "*Dora*". In practice, complex values are used in fairly restricted ways; primarily as *weak entities* in the Entity-Relationship approach [KLW93]. It does not seem reasonable for the user to have to assign id's to objects that are not going to be accessed through these id's. For this reason, *F-logic* permits the use of "don't care" object id's. The above example can then be written as:

$$dora[name \rightarrow \text{``}Dora\text{''}; address \rightarrow \_[city \rightarrow \text{``}Patras\text{''}; street \rightarrow \text{``}Kalavriton\text{''}]],$$

where the underscore denotes a "don't care" object id. The exact semantics for this don't care symbol is as follows: If

$$T[ScalM@S_1, ...S_n \rightarrow \_[...]]$$

is a term with a don't care symbol, then it's meaning is given by the following encoding:

$$T[ScalM@S_1, ...S_n \rightarrow val_n(T, ScalM, S_1, ...S_n)[...]]$$

where val_n is a new $(n+2)$-ary function symbol, chosen specifically for this encoding. In the previous example, the value of attribute *address* is

$$dora[address \rightarrow val_n(dora, address)[city \rightarrow \text{``}Patras\text{''}; street \rightarrow \text{``}Kalavriton\text{''}]].$$

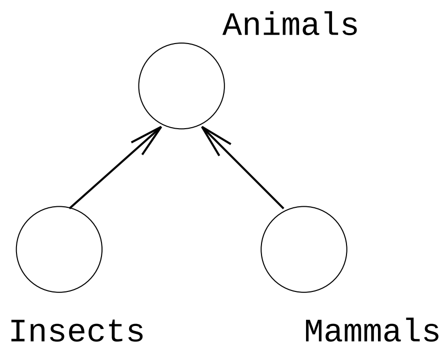


Figure 4.5: The Base Schema of the Animal Database

In conclusion, we should mention that the flexibility of imaginary objects, does not come free (much like everything else in this world). The price to pay, is the difficulty of maintaining imaginary objects and updating views that create such objects, as discussed earlier in Chapter 3. We should note however, that the view mechanism we present here does not depend on imaginary objects and can work without them. In fact, many other approaches [Run92b, SLT91, Ber92] do not even consider such objects. In our case, imaginary objects are an additional feature, that increases the flexibility and the users' convenience.

4.2.8 An Example View Definition in *F-logic*

In this section, we give an example showing how a view definition in the SQL-like notation of Chapter 3, is translated into an equivalent definition in *F-logic* rules.

The initial database is the one in Figure 4.5. It consists of a class *Animals* of animals and two subclasses *Insects* and *Mammals*. We want to create a view of this database for an educational game, where animal descriptions will be given and the user will try to guess the names of the described animals. Obviously we need to hide the attribute *Name* of *Animals* in the view, and moreover, to perform some additional restructuring of data. We create the view *Animal_Game* by using the following SQL-like commands:

create-view *Animal_Game as*

(1) *import class Animals from Database*;

hide attribute Name in class Animals;

(2) *class Carnivore includes*

$(select\ M\ from\ Mammals$

$where\ M.Eats =$"Meat");

(3) *class Vegetarian includes*

$(select\ M\ from\ Mammals$

$where\ M.Eats =$"Veggies");

(4) *class VeggieAndCarnivore intersection of Carnivore, Vegetarian*;

(5) *class* $InsectsOf(X)$ *includes*

$(select\ I\ from\ Insects$

$where\ I.Found\_In = X)$;

(6) *class InsectsOfAsiaOrAmerica*

includes $InsectsOf(Asia), InsectsOf(America)$;

end-view

The resulting view is depicted in Figure 4.6. The class of mammals is specialized into two subclasses *Carnivore* and *Vegetarian* depending on the eating 'habits' of the mammals. The latter two classes are then intersected to group those mammals that eat both meat and veggies (e.g., people). On the other hand, the class of insects is parameterized according to the continent on which each insect is found. That is, subclasses of *Insects* of the form $InsectsOf(Asia)$, $InsectsOf(Europe)$, etc., will be created. We also de-

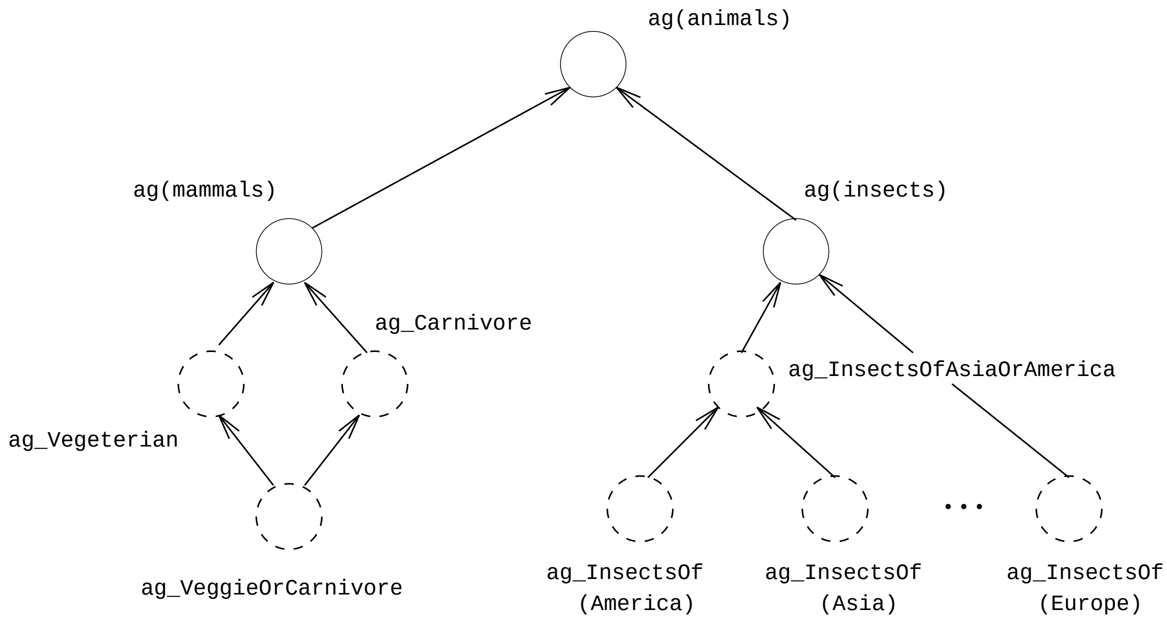


Figure 4.6: The View Schema of the *Animal_Game* View

fine the generalization of two of these parametric classes, namely $InsectsOf(Asia)$ and $InsectsOf(America)$, and call it $InsectsOfAsiaOrAmerica$. The above definition is then translated into the following *F-logic* rules (we assume the view is called ag, for $Animal\_Game$):

(1) $ag(C1) :: ag(C2) \leftarrow C1 :: C2 \wedge \ C1 :: animals \ \wedge \ C2 :: animals$

$ag(O) : ag(C) \ \leftarrow \ O : C \ \wedge \ C :: animals$

$ag(C)[X \Rightarrow T] \ \leftarrow \ C[X \Rightarrow T] \ \wedge \ C :: animals \wedge X \neq name$

$ag(C)[X \Rrightarrow T] \ \leftarrow \ C[X \Rrightarrow T] \ \wedge \ C :: animals$

$ag(O)[X \ \rightarrow \ V] \ \leftarrow \ O \ : \ C[X \ \rightarrow \ V] \ \wedge \ C \ :: \ animals \wedge X \ \neq \ name$

$ag(O)[X \twoheadrightarrow V] \ \leftarrow \ O : C[X \twoheadrightarrow V] \ \wedge \ C :: animals$

(2) $M : ag\_Carnivore \ \leftarrow \ M : ag(mammal)[eats \rightarrow$ "Meat"$]$

(3) $M : ag\_Vegetarian \ \leftarrow \ M : ag(mammal)[eats \rightarrow$ "Veggies"$]$

(4) $O : ag\_VeggieAndCarnivore \ \leftarrow \ O : ag\_Carnivore \ \wedge \ O : ag\_Vegetarian$

$$
\begin{aligned}
ag\_VeggieAndCarnivore[X \Rightarrow T_1] \;&\leftarrow\; ag\_Carnivore[X \Rightarrow T_1] \\
&\qquad \wedge \neg\, ag\_Vegetarian[X \Rightarrow (\;)] \\
ag\_VeggieAndCarnivore[X \Rightarrow T_2] \;&\leftarrow\; ag\_Vegetarian[X \Rightarrow T_2] \\
&\qquad \wedge \neg\, ag\_Carnivore[X \Rightarrow (\;)] \\
ag\_VeggieAndCarnivore[X \Rightarrow (T_1, T_2)] \;&\leftarrow\; ag\_Carnivore[X \Rightarrow T_1] \\
&\qquad \wedge\, ag\_Vegetarian[X \Rightarrow T_2]
\end{aligned}
$$

(5) $I : ag\_InsectsOf(X) \;\leftarrow\; I : ag(insects)[foundIn \rightarrow X]$

(6) $I : ag\_InsectsOfAsiaOrAmerica \;\leftarrow\; I : ag\_InsectsOf(Asia)$

$$
\begin{aligned}
I : ag\_InsectsOfAsiaOrAmerica \;&\leftarrow\; I : ag\_InsectsOf(America) \\
ag\_InsectsOfAsiaOrAmerica[X \Rightarrow T] \;&\leftarrow\; ag\_InsectsOf(Asia)[X \Rightarrow T_1] \\
&\qquad \wedge\, ag\_InsectsOf(America)[X \Rightarrow T_2] \\
&\qquad \wedge\, lub(T_1, T_2, T)
\end{aligned}
$$

□

4.3 Special Issues

4.3.1 View Independence

The concept *view independence*, introduced in [Run92b], is intended to complement the earlier notion of *data independence*, developed for the relational model. A system is said to provide logical data independence if it supports a view mechanism which allows the user to support his own view schema on top of the common logical schema. However, data independence does not account for the possibility of existing views having to be redefined when the underlying data model is extended or reorganized. Such redefinition of a view's specification and/or semantics is an unacceptable overhead. For eliminating the possibility of such a redefinition being needed, we can employ the concept of view independence. A database is said to provide *view independence* if "the specification and the semantics of existing view schemata are not affected by the definition of new view schemata" [Run92b]. Therefore, proving a database to be view independent ensures that the view schema definition and semantics will be immune to changes of the underlying global schema.

Within our model, however, the concept of view independence does not seem to have any significance. Since our approach consists of importing the desired portion of the base schema and then, within the view, performing any additional restructurings, the base schema is not modified. All new class derivations and classifications occur within the view boundaries, so the definition of new views does not have any affect on the underlying base schema. Therefore, our approach is by default *view independent*.

4.3.2 Views on top of Views

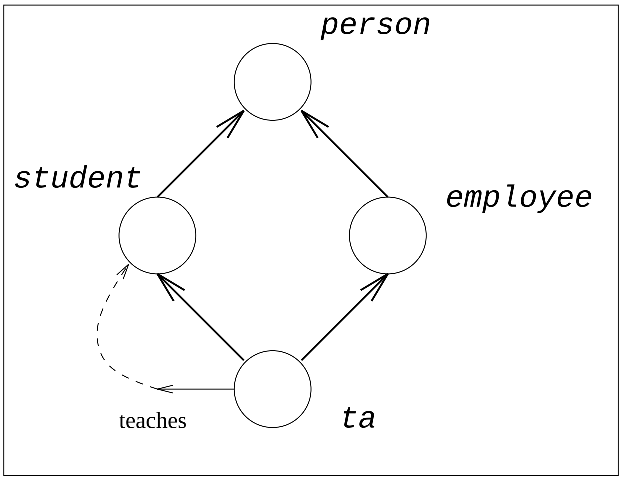
In our model, we consider a view to be a database that imports data from other databases, and restructures this data in a suitable, for the user, way. Imported data need not come only from a base schema. Since a view is treated as a database, we can import data from other views. Therefore,

create-view v as

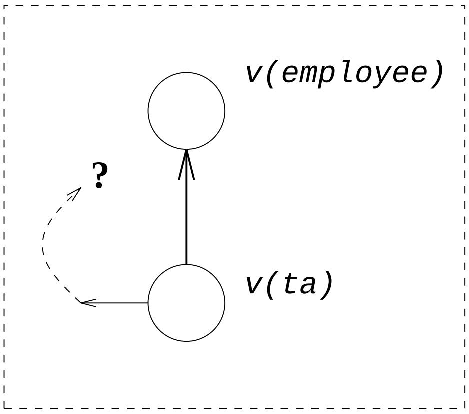
import class <*class*> *from* v_i

creates a view v on top of a predefined view, v_i. The whole procedure is the same as the one we follow when importing from a base schema.

The ability to define views on top of views, adds quite a lot to the flexibility. If a programmer wishes to create a new view that is slightly different from an existing view, then he need not follow the whole procedure right from scratch by importing from the base schema. Instead, he can import classes and objects from the existing view schema and perform slight restructurings, avoiding potentially large overhead. A more detailed



(a) The base schema



(b) The view schema

Figure 4.7: The Need to Generate Closed Views

discussion of this issue is given later on, in the next chapter.

4.3.3 View Closure

View closure [TYI88, HZ90, Run92b], also known as *interface coherence* [AB91], is an issue concerning the property decomposition hierarchy. A view is called *closed* (or *type-closed*) when for every property visible in the view, its arguments and results are also visible in the same view. More specifically, if p is a property of view class $C1$ with $domain_p(C1) := C2$, then class $C2$ must also be in the view. This is depicted in Figure 4.7.

When base class *employee* is imported into the view, its subclass, *ta*, of teaching assistants is automatically imported too. However, *ta* has a property *teaches* with $domain_{teaches}(ta) := student$, which suggests that *student* should also be imported into the view, in order to avoid inconsistencies.

In [Run92b] a simple algorithm that, given a view schema, calculates the minimal set of classes that should be added to it to make it closed, is presented. For every view class C_i, all its properties are checked in the following way. If $p \in properties_{C_i}$ and $domain_(C_i) = C_j$, then class C_j must also be in the view. If C_j is not in the view, then it is appended to the set of classes that need to be imported, in order to make the view closed. These classes are considered view classes and are also checked by the algorithm. The algorithm is of linear complexity $O(min(|V|, |A|))$, where $|V|$ is the number of classes in the global schema, and $|A|$ the number of arcs in the property decomposition hierarchy. The algorithm terminates when the view is type-closed.

The above algorithm was devised based on the assumption that none of the initially selected view classes could be dropped in order to make the view schema closed. It would also be interesting to develop tools that from the initial non-closed view schema would infer the smallest coherent portion contained in this view schema. Furthermore, in our case, another possibility would be to filter out from the view those properties that cause the problem, as for instance *teaches* in the example of Figure 4.7. However, these issues are not considered here.

Chapter 5

Classifying the Virtual Classes

In the previous chapter, we saw ways of defining virtual classes from base ones, in order to satisfy the needs of the user. These classes were derived by stored *F-logic* rules corresponding to virtual class derivation queries, and were interrelated with other classes via diverse relationships. In this chapter, we argue that it is quite advantageous to integrate the derived classes into the *is-a* hierarchy. The necessity of classifying the virtual classes has been indicated in many other approaches [TYI88, AB91, SLT91, Run92a], in contrast to earlier approaches, in which the virtual classes are treated as "stand-alone"objects [HZ90] or are attached as direct subclasses to the schema root class [Kim89].

5.1 Classification in General

Class classification, also called *class integration*, is the process of taking a new class and placing it in the appropriate position in the class hierarchy. In the relational model this process simply corresponds to the addition of the new virtual relations to the list of existing relations. Each relation is independent of all other relations, so there exists no need to explicitly define the interrelationships between the new relations and the old ones. In the context of object-orientation, however, classification is a difficult and complex task. This is because the placement of the new classes into the class hierarchy has to maintain both the consistency of the existing is-a *and* property decomposition hierarchies. This restriction complicates the whole process by giving rise to a number of non-trivial problems. However, the confrontation and solution of these problems will result in a number of significant advantages.

First of all, the integration of virtual classes in the global schema supports type inheritance between base and virtual classes. Virtual classes are defined in terms of existing classes, and therefore often use attributes, methods, object instances, or other kind of (possibly) complex functions of the base classes. So instead of all these '*properties*' being unnecessarily duplicated in the virtual classes, class integration would allow them to be inherited via type inheritance.

A more pragmatic advantage of class integration is a possible optimization of the system's performance. Capturing explicit relationships between stored classes and the newly derived ones may turn into a valuable information source for the query optimizer during query processing. For instance, the union of a class and its subclass is just the former class. So the query optimizer, before processing a query which requires the union of two classes, can just check for the existence of a superclass-subclass relationship between the corresponding two classes, avoiding this way expensive query processing. This can only be done if all such relations are explicitly captured, which is the case with class classification.

Furthermore, if we are to consider dealing with update operations, classification of the derived virtual classes turns out to be a wise choice. The more complete the class hierarchy is, the faster consistent processing of update operations becomes. Inserting into a subclass, for example, automatically inserts into all superclasses too. If classification is not used, we will also have to check the class predicates of all other classes, when updating a single virtual class.

To sum up, we should mention that when a view is considered to be a virtual *schema* rather than a virtual *class*, class integration becomes a mandatory rather than an optional task. If the virtual classes are not classified after being derived, then the view will most likely correspond to a collection of possibly unrelated classes rather than to a generalization schema graph.

5.2 To Classify Manually or Automatically?

There are two ways one could follow for the integration of the derived virtual classes: *manual* integration or *automatic* integration.

Manual integration refers to the task by which the view definer explicitly places the derived class in a certain position in the global schema. He defines the direct superclasses of the virtual class, by inserting directed is-a arcs from the virtual class to the direct superclasses. Analogously, is-a arcs from other classes to the virtual class define the former to be direct subclasses of the later. However, manual classification has two major disadvantages.

To begin with, the programmer must know about all classes in the global schema and their interrelationships. Since the global schema in many applications can be large, and consistently growing as more virtual classes are integrated, manual integration can become a difficult task. If however, the classification procedure is automated, then the view definer's job is significantly simplified.

The other drawback of manual classification concerns the human factor, which can lead to all kinds of inconsistencies. For instance, the view definer could, by mistake, insert incorrect generalization arcs into the schema. The result would be an incorrect *is-a* relationship between two or more classes. This automatically translates into inconsistent subset and subtype relationships between classes and, moreover, to incorrect inheritance of properties. It could also be the case that edges were omitted, or even that redundant edges were added by mistake. As a result inheritance of properties would not be performed properly, redundancy would appear, and therefore, the resulting global schema would become confusing for the users. Once more, in this case, automatic classification of the virtual classes proves to be a better solution. It can guarantee a correct and complete placement of the derived classes in the schema and wipe out the possibility of an inconsistent database system. Algorithms can be devised to handle correct placement of virtual classes, insertion of all required and deletion of all redundant or inconsistent *is-a* arcs [RB92].

Consequently, we choose to automate the classification procedure and gain the aforementioned advantages. However, the view definer is not prevented from classifying the virtual classes manually, if he desires to do so. *F-logic* allows for the insertion of arbitrary *is-a* arcs by the view definer, in which case the classification algorithm can be used as an advisory tool, as well as a consistency checker.

5.3 Classifying Within the Views

As discussed in the previous chapter, our approach towards defining views over a base schema could be described as *importing* all the required classes and then performing operations on the imported classes to derive extra classes. This dictates, in some way, that the virtual classes derived within some view need to be classified inside that view. Therefore, choosing to integrate each class within the view in which it belongs is a natural decision. There are, however, some potential problems that one could see in this kind of treatment of class integration.

To begin with, a problem seems to arise when two or more views are initiated in a similar fashion (by importing the same classes) and then, up to some point, the same set of virtual classes is derived in both. Such a situation is depicted in Figure 5.1. Views $v1$ and $v2$ are identical, except for the class $v2\_MaleT$ derived in $v2$. This means that up to the point where $v2\_MaleT$ is derived, the two views will be derived by similar definitions

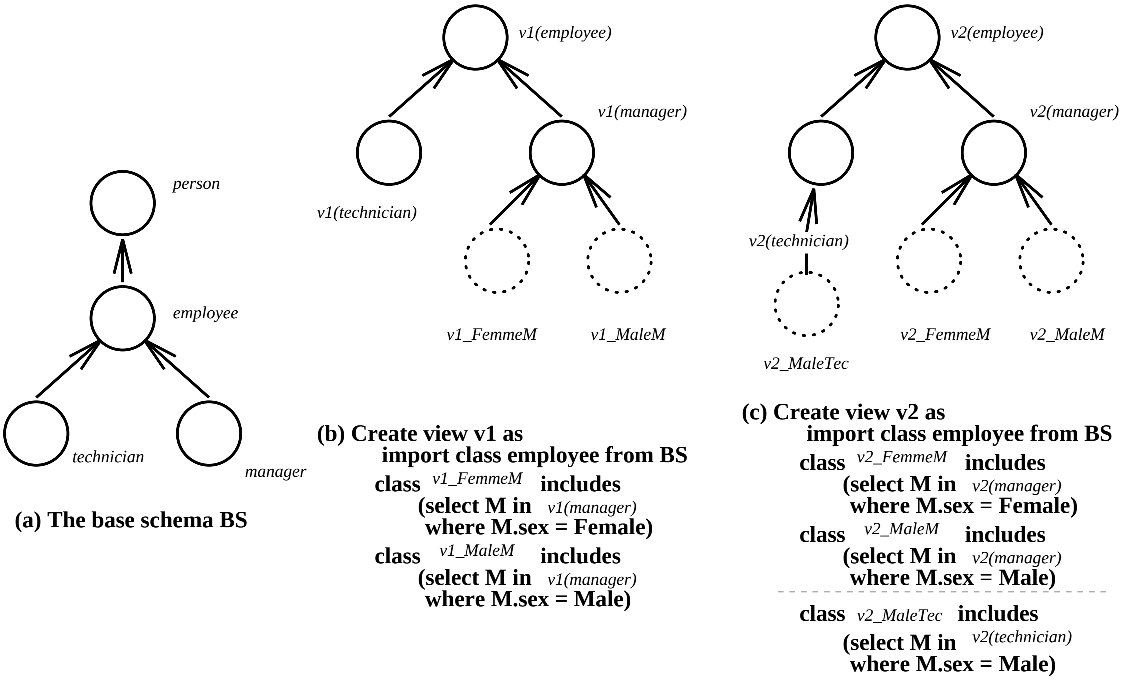


Figure 5.1: A Potential Source of Duplication

(Figure 5.1(a) and Figure 5.1(b)). The redundancy in the form of duplicated view definition commands is obvious. Apart from this, we will also have multiple invocations of the classification algorithm for the same class. For instance, the class of female managers will be classified once as $v1\_FemmeM$ in view $v1$, and once as $v2\_FemmeM$ in view $v2$. The same goes for the class of male managers. The redundancy in both of the above two forms would rise with the number of 'look-alike' views, as well as with the number of virtual classes derived in the same way in each of these views.

However, it is quite unlike that there will be a great number of views that "look alike" and have the same set of virtual classes derived in them. But even if this is the case, the problem can still be solved. We can perform the classifications in one view and then define the other views on top of this one view, instead of defining them on top of the global schema. For instance in the above example, we could first define $v1$ as in Figure 5.1, and then go on to define $v2$ on top of $v1$ as:

create-view v2 as

$$import\ class\ v1(employee)\ from\ v1$$

This eliminates the need for deriving and classifying classes $FemmeM$ and $MaleM$ in $v2$, since importing $v1(employee)$ will also import all its subclasses including the aforementioned ones. One should note that even if we choose to classify within one global schema, we still have to eventually select the classes that comprise each view and form them into a view schema, as is done in [RB92]. There, an algorithm with complexity $O(|GS|^3)$, where $|GS|$ is the number of classes in the global schema, is used to construct a view schema by choosing the appropriate classes from the global schema. This algorithm is executed one time for each view that is created. When only single inheritance is allowed, the complexity drops to $O(|GS|)$. In our case, however, classifying within the view eliminates the need of such a procedure and, as shown above, does not result in significant redundancy.

A second potential problem that could result from our approach concerns object and property sharing. One of the major reasons for integrating the derived virtual classes is that of property sharing via inheritance and object sharing, so that unnecessary duplication is avoided. However, since we integrate virtual classes into the view schema in which they are derived rather than into a global schema, one might wonder whether properties and objects are still shared in a correct way. As we show in the sequel, both the above issues are dealt with properly.

As shown earlier in Section 4.2.1, importing a class <*class*> from the base schema BS or from a view schema VS into a view v is translated into the following *F-logic* rules:

(1) $v(C1) :: v(C2) \leftarrow C1 :: C2 \ \wedge \ C1 :: <class> \wedge C2 :: <class>$

(2) $v(O) : v(C) \ \leftarrow \ O : C \ \wedge \ C :: <class>$

(3) $v(C)[X \Rightarrow T] \ \leftarrow \ C[X \Rightarrow T] \ \wedge \ C :: <class>$

(4) $v(C)[X \Rightarrow\!\!\!\!\!\!> T] \ \leftarrow \ C[X \Rightarrow\!\!\!\!\!\!> T] \ \wedge \ C :: <class>$

$$(\mathbf{5}) \qquad v(O)[X \rightarrow V] \;\leftarrow\; O : C[X \rightarrow V] \;\wedge\; C :: <class>$$

$$(\mathbf{6}) \qquad v(O)[X \twoheadrightarrow V] \;\leftarrow\; O : C[X \twoheadrightarrow V] \;\wedge\; C :: <class>$$

From the "typing rules" (rules 3,4) it is easy to see that any property of class C, for instance, is also a property of the view class $v(C)$. Therefore, properties do not have to be redefined in the view class, which means that no duplication takes place. The same goes for the other view classes too. Moreover, note the property inheritance taking place within the view itself. By the ":: rule" (rule 1), since $C1 :: C2$, it will also be the case that $v(C1) :: v(C2)$, so $v(C1)$ inherits all properties of $v(C2)$. For object instances, the situation is similar. Rule 2 guarantees that if an object O is in class C, then $v(O)$ will be in class $v(C)$. The only form of redundancy that occurs is that of more than one object ids for the same object. In the previous case O and $v(O)$ are two different object ids, but according to rule 2 these essentially refer to the same object. However, this redundancy is not a serious problem since the object ids for objects within the view (e.g., $v(O)$) are not stored but only computed on demand by executing the respective rule.

In conclusion, the decision to classify virtual classes within the view in which they are derived, does not fall short in any of the advantages of integrating into a global schema as described at the beginning of this chapter. This is done with no unnecessary duplication or noticeable redundancy. The aforementioned approach to classification is a natural consequent of our choice to separately defining each view (by importing base classes) and then work on it independently, as opposed to constructing a complex global schema and then forming view schemata by selecting the appropriate classes from the global schema.

5.4 The Class Classification Algorithm

Once a virtual class has been derived, we proceed with its classification. By doing so we get to enjoy all the previously described advantages of classification. To classify a

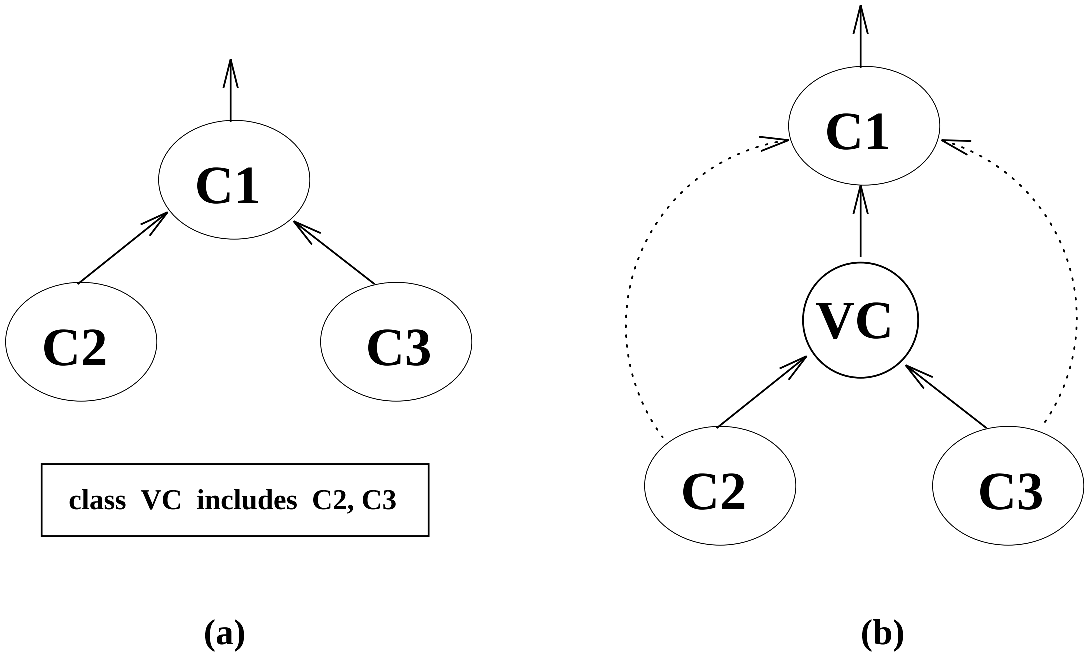


Figure 5.2: Propelling a Virtual Class in the Middle of the Hierarchy

virtual class, essentially means to take the class and place it where it belongs in the class hierarchy [SL83]. In general, assume a virtual class VC defined as including classes C_1 ... C_k as well as objects from classes C_{k+1} ... C_n. Then the superclasses and subclasses of VC are defined as follows [AB91] :

1. If D is a superclass of C_1 ... C_n, then D is also a superclass of virtual class C,
2. Each C_i is a subclass of C for $1 \leq i \leq k$.

Note how these definitions can insert a virtual class right in the middle of the hierarchy. For example, if the existing hierarchy is the one defined in Figure 5.2(a) and the virtual class VC is defined to include classes B and C, then Figure 5.2(b) shows the hierarchy after the classification of VC has taken place. The initially existing *is-a* arcs from classes B and C to class A have to be removed, since they are redundant. The *is-a* relationships between the two classes and A are now implicitly modeled via the inserted virtual class VC.

A virtual class is correctly inserted, if it is placed bellow all classes that *subsume* it (i.e., its superclasses) and above all classes that it *subsumes* (i.e., its subclasses). *Subsumption* is a relation between classes, such that class A subsumes class B only if the $extent(A)$ includes the $extent(B)$. Subsumption is also an issue in the context of Concept [NS90], Frame-Based [BL84] and Semantic-Based Network [SL83] languages. Testing for subsumption can be done by using a function $Subsumes()$ defined as follows:

$$Subsumes(A, B) = \begin{cases} true, & \text{if } B \text{ } \textit{is-a} \text{ } A \\ fail, & \text{otherwise} \end{cases}$$

In the next chapter, we will elaborate on the subsumption function and on the issues involved in the problem of determining subsumption relationships between classes. We will also present two approaches to solving the subsumption problem within our model, based on work previously done on the matter, in the field of concept languages, as well as in the query optimization field.

Once the $Subsumes()$ function has been developed, it becomes a major component of the classifier. After a virtual class VC is derived, the classifier uses the $Subsumes()$ function to determine the subsumption relationships between VC and all other classes in the global schema. Of course, since we have chosen to classify the virtual classes within the view in which they are derived, we only need the subsumption relationships between VC and the other view classes in VC's view. After the relationships are acquired, VC is placed above all its direct subclasses and bellow all its direct superclasses. In [Run92a] it is claimed that such a classification algorithm, which is repeatedly advocated in the literature [SL83, TYI88, AB91], does not correctly account for the type inheritance underlying the schema graph. This is because of two problems that this algorithm does not address properly. These are the *type inheritance mismatch* and the *is-a incompatibility* problems. In the following sections, however, we demonstrate how our (different) approach to classification (classifying within the view rather than within the global schema) and the semantics for our virtual class derivation operators (especially the way we perform *hiding*) can be used to overcome the aforementioned problems. We then close the chapter with the presentation of the classification algorithm as a whole.

5.5 Two Problems of Class Classification

There are two problems with the classification procedure described in the previous chapter. These problems are closely related to object-oriented data models and, therefore, are not handled appropriately by earlier work on classification. The first problem is the inheritance mismatch in the type hierarchy [SLT91, Run92a], and the other is composing *is-a* incompatible subset and subtype hierarchies into one consistent class hierarchy [Run92a]. Although in [Run92a] it is claimed that a simple classification algorithm like the one described in the previous section does not overcome these problems, I will demonstrate in the following how these can be dealt with in the context of our semantics.

5.5.1 The Type Inheritance Mismatch Problem

This problem arises when the type of a virtual class does not allow for the class to be properly integrated into the global (or in our case the view) schema. In other words, there is no place in which the virtual class can be placed, in order to preserve correct type inheritance. This is demonstrated in Figure 5.3

Suppose we have a virtual (derived) class VC, obtained by hiding property b from $C3$ (Figure 5.3, above). Then the type and content of VC are the ones presented in the box. It is clear that there is no proper place in the existing hierarchy for VC to be inserted. VC is a superclass of $C3$, since it has the same content and all its properties apply to $C3$ as well. Therefore VC must be placed above $C3$ in the class hierarchy. But then we cannot derive any *is-a* relationships between VC and the two remaining classes. This is because there exist properties (e.g., d) defined in VC but not applicable to $C2$ and vice versa. The same goes for VC and $C1$. This means that the virtual class cannot be properly inserted in the hierarchy. The type inheritance problem does not depend on the extents of the classes, but only on their types.

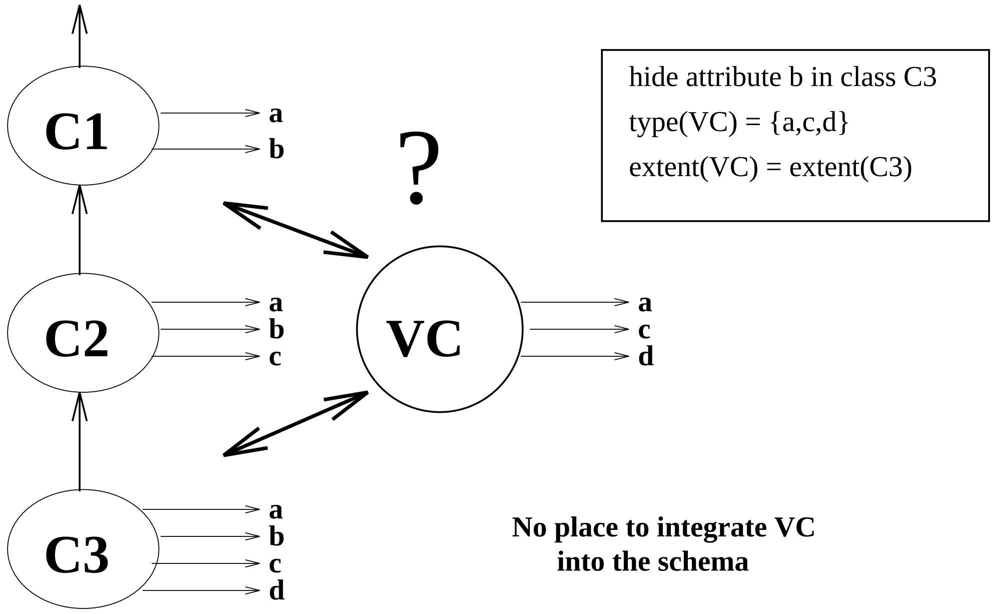


Figure 5.3: The Type Inheritance Mismatch Problem

5.5.2 The *is-a* Incompatibility Problem

This problem is caused by conflicts between the subset and subtype relationships, for the position of the virtual class. Such conflicts can arise because the class hierarchy combines both the subset and subtype relationships between classes into one single *is-a* hierarchy. So, if the type of the virtual class is lower in the type hierarchy than the position of the same class's extent in the set hierarchy, then a conflict will arise when trying to compose the *is-a* hierarchy from the other two hierarchies.

As can be seen from Figure 5.4, VC is a subtype of both $C2$ and $C3$, since every property defined in the latter two classes is also applicable to VC. This means that VC must be placed *under* these two classes in the hierarchy. On the other hand, the content of VC is the union of the objects belonging to $C2$ and $C3$, so $C2$ and $C3$ are clearly subsets of VC. This dictates the placement of VC *above* the two classes in the hierarchy. Consequently there is no proper location for the virtual class to be inserted into the hierarchy.

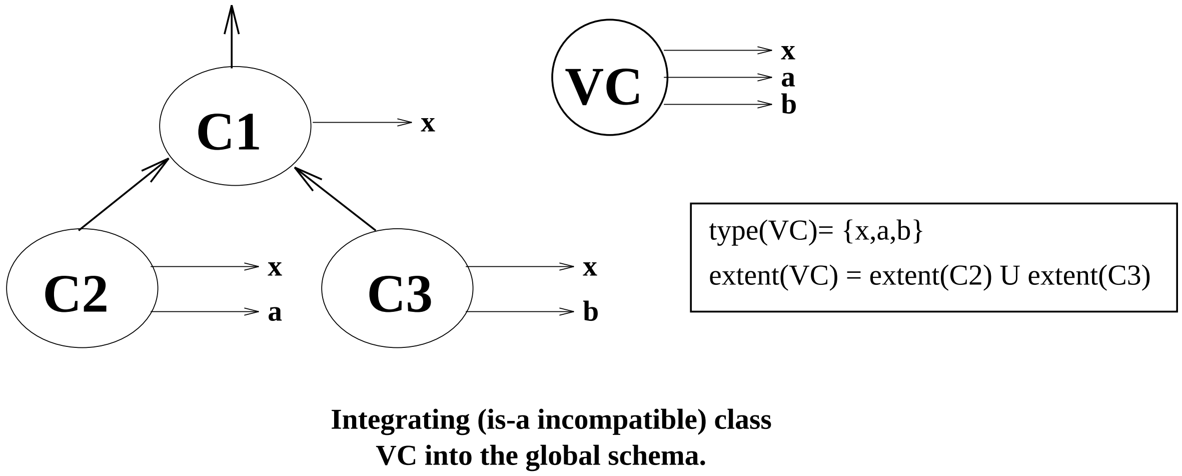
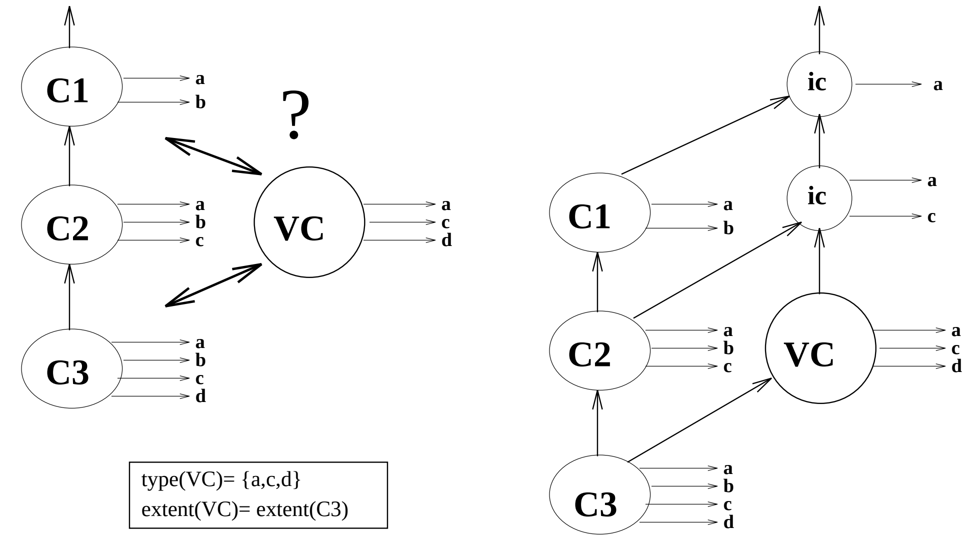


Figure 5.4: The *is-a* Incompatibility Problem

5.6 Overcoming the Type Inheritance Mismatch Problem

A solution to problem is presented in [MS89, Run92a]. The latter is more general since it deals with the type and set content of the class, while Missikoff and Scholl focus only on type classification. The general idea can be described as integrating intermediate classes into the class hierarchy in order to make the insertion of the virtual class possible. This is demonstrated in Figure 5.5. The insertion of the two intermediate classes ic is done so that the properties common to VC and the two base classes are filtered out. After this, VC can be correctly inserted into the schema. The algorithm that [Run92a] presents reorganizes the schema graph in order to produce a class hierarchy closed under $\sqcap$, where $\sqcap$ denotes the least common supertype of two types. That is, for any two classes C_1 and C_2, there must exist a third class C_3 in the graph with $type(C_3) = type(C_2) \sqcap type(C_1)$ and $extent(C_3) = extent(C_1) \cup extent(C_2)$. This is done by generating new intermediate classes and connecting them with the existing ones, while preserving the correct type inheritance. This type preparation algorithm has quadratic complexity $O(m^2)$, where m is the number of edges in the global schema.

In our model, the alternative approach we take for view creation and our semantics for the *hide* operator make the difference. These two factors guarantee a correct solution



(a) No place to integrate VC into the schema **(b) Create intermediate classes to preserve type inheritance**

Figure 5.5: A Solution to the Type Inheritance Mismatch Problem

to the type inheritance mismatch problem without a type preparation algorithm similar to the one presented above. Since such an algorithm has complexity quadratic in the number of edges in the global schema, and since the global schema in most applications is quite big and constantly growing, and since the algorithm has to be used every time a specific type of class derivation (e.g., hiding) occurs, it is easy to see how our approach avoids the aforementioned overhead.

Recall that in our approach, virtual classes are derived and classified within view schemas. It would, therefore, be meaningless to hide a property in one class while at the same time retaining it in some superclasses. For instance, assume that a view v_1 is created by importing class *employee* from the base schema. This would result in automatic importing of all subclasses of employee, as depicted in Figure 5.6. The *F-logic* rules for this view creation would be:

$$\textbf{(1)} \qquad v(C1) :: v(C2) \leftarrow C1 :: C2 \;\wedge\; C1 :: <class> \wedge C2 :: <employee>$$

$$\textbf{(2)} \qquad v(O) : v(C) \;\leftarrow\; O : C \;\wedge\; C :: <employee>$$

$$\textbf{(3)} \qquad v(C)[X \Rightarrow T] \;\leftarrow\; C[X \Rightarrow T] \;\wedge\; C :: <employee>$$

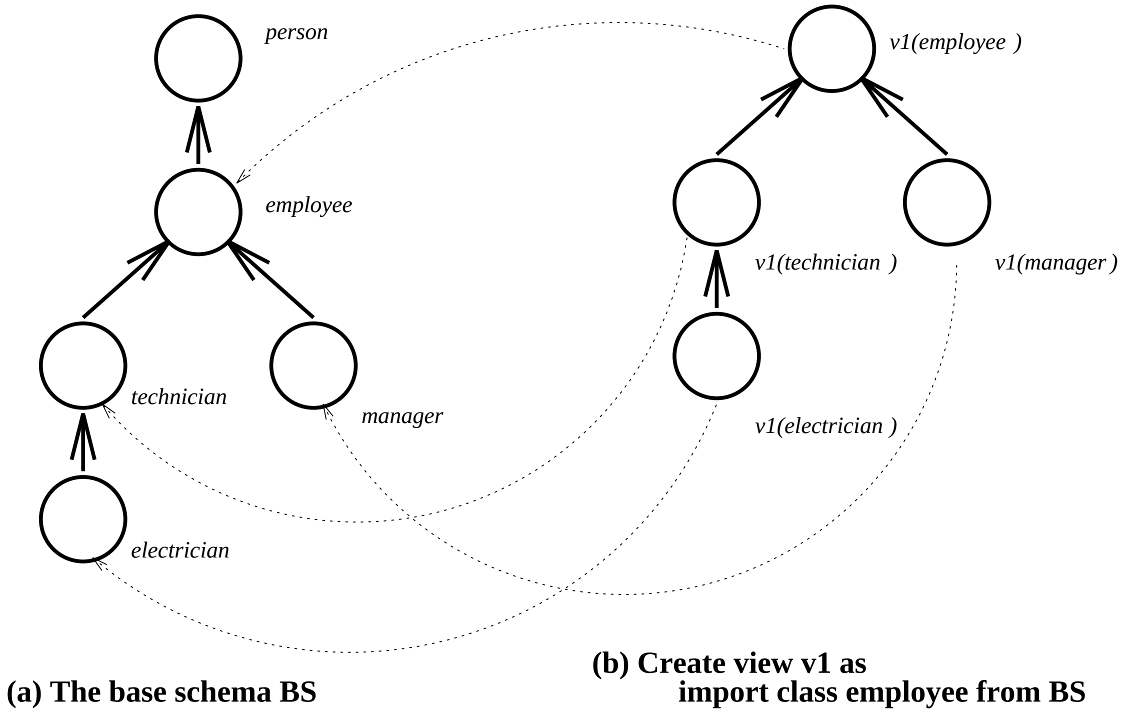


Figure 5.6: Importing Class Employee from the Database

$$(\mathbf{4}) \qquad v(C)[X \Rrightarrow T] \;\leftarrow\; C[X \Rrightarrow T] \;\wedge\; C :: <employee>$$

$$(\mathbf{5}) \qquad v(O)[X \rightarrow V] \;\leftarrow\; O : C[X \rightarrow V] \;\wedge\; C :: <employee>$$

$$(\mathbf{6}) \qquad v(O)[X \twoheadrightarrow V] \;\leftarrow\; O : C[X \twoheadrightarrow V] \;\wedge\; C :: <employee>$$

Assume a scalar attribute *salary* defined in base class *employee*. This is then inherited via type inheritance by base class *manager* and imported by rule 3 (above) from *manager* into $v_1(manager)$. However, via the same rule *salary* is imported from *employee* to $v_1(employee)$. Therefore, any attempt to hide *salary* only in view class $v_1(manager)$ would not have the expected results: since $v_1(manager)$ is a subclass of $v_1(employee)$, every object $v_1(O)$ in the former class also belongs in the extent of the latter class. This means that the salary of $v_1(O)$ will still be visible in class $v_1(employee)$. Therefore, hiding is not performed correctly. Consequently, this justifies the approach we take to hiding a property function not only in one class (e.g., $v_1(manager)$), but also in every other class of the view, by blocking its import. We can perform this sort of blocking by adding a simple constraint to the import rules. Specifically, the conjunct

$(X \neq salary)$ is added to rules 3 and 4, which results in blocking the import of the attribute *salary* in the view.

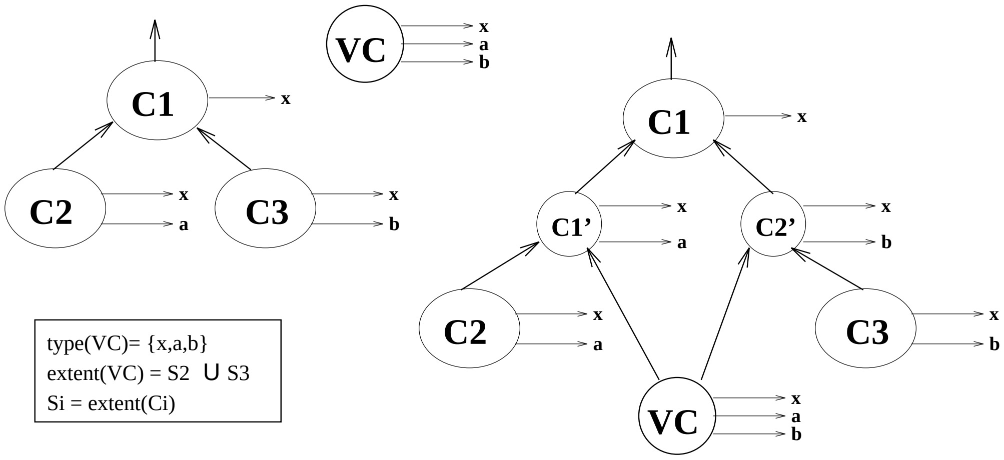
Such an approach seems to be more natural than just creating a virtual class VC with $extent(VC) = extent(v_1(manager))$ and $type(VC) = type(v_1(manager)) - \{salary\}$. Furthermore, it is definitely more general since hiding in a class and all its subclasses is done with one simple operation. To our knowledge, all other object-oriented view models can perform hiding in only one class at a time by projecting away the specific property. By following this approach to property hiding, it is easy to see that situations similar to the one presented in Figure 5.3 can not possibly occur within the context of our model. In our case, hiding property b from class $C3$ would also hide b from $C2$ without creating a virtual class. Therefore no *is-a* hierarchy mismatch problem would occur.

Hence, we have demonstrated the way in which our semantics successfully overcomes the type inheritance problem. This is done without using a type preparation algorithm, as in [Run92a], which would result in a quadratic (with respect to the number of *is-a* arcs in the global schema) complexity overhead every time a specific type of virtual class is derived.

5.7 Overcoming the *is-a* Incompatibility Problem

A general solution to this problem is presented in [Run92a]. It is based on creating additional intermediate classes, so that the virtual class can be correctly integrated. Figure 5.7 shows how. The hierarchy is reorganized by the creation and insertion of classes $C2'$ and $C3'$ (Figure 5.7(b)). Their types are the types of their respective source classes ($C2$ and $C3$), and their set contents equal to the content of VC. Now VC can be placed under them without violating the hierarchy constraints.

Note how this solution keeps the hierarchy closed under $\sqcap$. That is, for any two types in the hierarchy, their least common supertype is also in the hierarchy. The virtual



(c) Integrating (is-a incompatible) class VC into the global schema.

(d) Create intermediate classes to solve the is-a incompatibility problem

Figure 5.7: A Solution to the *is-a* Incompatibility Problem

class VC could have also been placed as a direct subclass of $C1$. In this case, however, the resulting hierarchy would not be closed under $\sqcap$.

However, Rundensteiner does not actually use this solution. The object algebra operators she uses for deriving virtual classes generate classes which either are both a superset and subset of the source class, or are both a subset and subtype of the source class. This means that no *is-a* incompatibility can occur. The only exception is her **hide** operator, which from a source class C generates a virtual class VC with $type(C) \prec type(VC)$ and $extent(C) = extent(VC)$. Figure 5.8(a) shows how a problem can arise when deriving a virtual class VC by hiding attribute c from $C3$. Under the assumption that $extent(C1) > extent(C2) > extent(C3)$, the above hide operation results in the type and set hierarchies presented in Figure 5.8(b). It is clear that while $type(VC)$ dictates that VC is placed directly above $C2$ and directly under $C1$, $extent(VC)$ requires that VC is placed directly under $C2$. These conflicting dictations of where the virtual class should be placed comprise the *is-a* incompatibility problem. However, this problem is solved in [Run92a] when using the type preparation algorithm to solve the previously mentioned type inheritance problem. Therefore, no extra solution is provided.

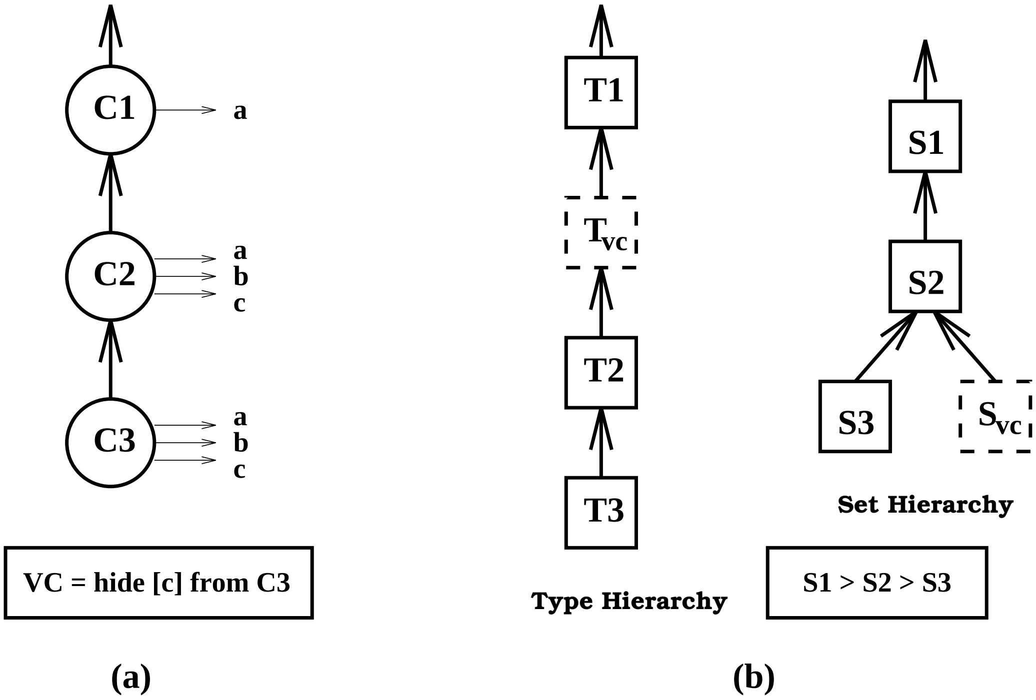


Figure 5.8: Problem When Hiding Within Single Class

In our case, the situation is similar. All the operators used for deriving virtual classes are not subject to the subclass/subtype incompatibility problem. The derived classes are all compatible with the existing *is-a* hierarchy. The difference in our case, though, is that hiding does not constitute an exception. The way our hiding mechanism works guarantees that no inconsistencies similar to those occurring in Figure 5.8(a) will ever occur. Our approach would not create a virtual class by hiding c from only one source class, but it would rather block the import of property c into the entire view. Hence, c would no longer be visible to any of the view classes. The resulting class hierarchy in our view, would be similar to that of Figure 5.8(a) but with property c hidden from both $C2$ and $C3$. Clearly there would be no disagreement between the type and set hierarchies and therefore, no incompatibility problem would occur.

5.8 Placing the Derived Classes in the Hierarchy

Once the *Subsumes*() function is defined and the subsumption relationships between classes can be derived, we follow a common approach for the actual placement of the derived classes that is used in many of the proposed object-oriented view models [SL83, TYI88, AB91, Run92a]. This approach consists of finding all the direct super- and sub-classes of the virtual class and placing it under the former and above the latter. A very simple example of this technique was presented in Figure 5.2. After the placement of the new class in the hierarchy takes place, all redundant *is-a* arcs (denoted as dotted arcs in the example) should be eliminated. This can easily be performed by removing all direct *is-a* arcs between any direct superclass and any direct subclass of the newly placed class (e.g., between C_1 and C_2). All the *is-a* relationships that these arcs used to denote, are now implicitly modeled via the new class (e.g., C_2 *is-a* VC, and VC *is-a* C_1).

Specific algorithms for finding the direct superclasses and direct subclasses of the virtual class to be integrated can be found in [Run92a]. The set of direct superclasses can be found by a depth-first search of the global schema (in our case the view schema). This can be performed with linear complexity in the number of *is-a* arcs in the schema. Note that once the direct superclasses are found, no direct subclass can be found above any direct superclass. Therefore, the search for the set of direct subclasses of the virtual class is performed only in those subgraphs of the schema, which are rooted at some direct superclass. The algorithm for finding the set of direct subclasses is also of linear complexity $O(|E|)$, where $|E|$ is the number of *is-a* arcs in the schema graph. These two algorithms can be combined to give an overall complexity of $O(|E| + |E|) = O(|E|)$. In our case, since classification is performed within the view, the resulting complexity turns out to be linear in the number of edges in the view schema $O(|VE|)$.

It should be noted, however, that the aforementioned complexity corresponds to the case in which the subsumption function is of constant complexity. If the complexity of the *Subsumes*() function is linear, then the complete classification algorithm is of quadratic complexity, and so on. This fact clearly demonstrates the significance of subsumption in the whole classification procedure. The $Subsumes()$ function is a major part of the classifier and its complexity affects the overall complexity of the classification procedure in a major way. It is, therefore, essential to develop an efficient $Subsumes()$ function and to discover special cases for which the subsumption problem can be solved with low complexity.

In the following chapter we elaborate on the definition of the $Subsumes()$ function and on the issues arising when testing subsumption relationships between classes. We also present two approaches to subsumption testing within the context of our model.

Chapter 6

Subsuming the Class Relationships

In the previous chapter, we praised the use of an automatic classifier, which would account for the placement of the newly derived virtual classes into the view schema. As mentioned there, the major part of this classifier is the $Subsumes()$ function which, applied on two classes, tests for the existence of an is-a relationship between them. In this chapter, we start out with a discussion on the subsumption problem in general, and then proceed with the presentation of techniques for solving this problem in the context of our model.

6.1 Subsumption in General

The main objective of the subsumption procedure is to discover the relationships holding between a newly derived virtual class and the other existing classes. This output is then used to determine the correct location in the schema, for the virtual class to be placed. A virtual class is correctly integrated into a schema, if it is placed bellow all classes that *subsume* it (i.e., its superclasses) and above all classes that it *subsumes* (i.e., its subclasses). Therefore *subsumption* is a relation between classes, such that class A subsumes class B only if the $extent(A)$ includes the $extent(B)$. Subsumption has also been an issue in the context of Concept [NS90], Frame-Based [BL84] and Semantic-Based Network [SL83] languages. Testing for subsumption can be done by using a function $Subsumes()$ defined as:

$$Subsumes(A, B) = \begin{cases} true, & \text{if } B \textit{ is-a } A \\ fail, & \text{otherwise} \end{cases}$$

The general features of the $Subsumes()$ function are described in [SL83]. In the context of object-oriented data models [Run92a], $Subsumes(A, B)$ is true if and only if:

1. For each property p_i of A, there is an equivalent property p_j defined for B.

2. The domain of each p_i in A, is a superset of the domain of the equivalent property p_j in B.

3. The constraints defined for p_i in A, include those defined for the equivalent property p_j in B.

4. Membership predicates restricting the set content of A, imply the corresponding predicates for B.

It is well known that the subsumption problem is undecidable in general. This is due to the comparison of arbitrary methods (e.g., in 1, above) or predicates (e.g., in 4, above) that has to take place when checking for a subsumption relationship between two classes. Comparing actual function bodies to determine equivalence of properties in condition 1 above is a potentially undecidable problem. Similarly, comparing arbitrary expressions that denote membership predicates, for instance, when considering condition 4 above, is also undecidable in general [Run92a]. We could, however, assume that each property function (method) has a unique property identifier. Then two property functions are equivalent if they have the same property identifier. By shifting the property function comparison problem to the simple test of identifier equivalence, we overcome the need to compare actual function bodies, which is a potentially undecidable problem. However, it may be that the semantics of the two property functions are the same even though these functions do not have the same property identifier. Therefore, some *is-a* relationships might not be discovered and the virtual class might not be placed as low in the hierarchy as possible. This would result in a less informative, but still correct, schema.

The other source of undecidability is the comparison of membership predicates. Membership predicates are predicates that define the population of the class, such as subset predicates for base classes and derivation queries for virtual classes. To overcome this sort of undecidability we could follow one of either two approaches. The first one is to permit only decidable predicates. The other is to implement an incomplete decision procedure, by considering during classification, only those parts of the class predicates that are known to be decidable.

The $Subsumes()$ function is *sound* and $total$, but not *complete*, due to the undecidability of the classification problem. This means that, if $Subsumes()$ returns $true$ for two classes, then these classes are necessarily *is-a* related. Furthermore, totality guarantees that $Subsumes()$ will always terminate and return either $true$ or $fail$. However, because of not being complete, $Subsumes()$ is not guaranteed to find all existing *is-a* relationships. If some subsumption relationships are not discovered, the virtual class is not placed as low in the hierarchy as would be possible. This placement, however, is still correct.

As mentioned in Chapter 5, the complexity of the subsumption method has a great impact on the overall complexity of the whole classification procedure. Due to this, it is essential to develop an efficient $Subsumes()$ function and even find special cases, for which the complexity of subsumption remains low. Moving in this direction, we present in the sequel two approaches to subsumption testing within the context of our model. The first approach is based on earlier work done in the field of concept languages [DLNN91], while the second is based on containment of conjunctive queries in the context of query optimization [Ull89].

6.2 Subsumption Within our Model

When a virtual class C is derived, the classifier will use the subsumption procedure to derive any relationships holding between the new class and the other existing view classes C_i. Here, we will concentrate only on the $Subsumes()$ function itself. That is, given two classes C_1 and C_2, we will discuss how the result of $Subsumes(C_1, C_2)$ is extracted. We will assume that the conditions concerning properties, domains and constraints (i.e., conditions 1, 2 and 3 in Section 6.1) have already been examined and proved to hold. Thus, in the rest of this section, we will concentrate on the fourth and most interesting condition, namely the one concerning membership predicates. That is, given that $type(C_2) \preceq type(C_1)$, we will be testing if $extent(C_2) \subseteq extent(C_1)$. If the latter holds, then $Subsumes(C_1, C_2)$ will return $true$, while in the opposite case it will return $fail$. If, however, $type(C_2) \not\preceq type(C_1)$, then $Subsumes(C_1, C_2)$ will return $fail$, and there is no need to test the membership predicates of the corresponding classes.

6.2.1 A Polynomial Approach to Subsumption

Our first approach is based on work done by Donini et al. [DLNN91] in the context of concept languages. There, a concept language, namely $\mathcal{PL}_1$, for which subsumption can be checked in polynomial time, is presented. We demonstrate how this technique can be adapted for our model and we present the sort of *F-logic* queries the technique can (or cannot) account for. The technique falls into the category of those permitting only decidable membership predicates.

Basic Notions

Concept languages provide a means for expressing knowledge about hierarchies of concepts, i.e. classes of objects with common properties [DLNN91]. They are given a Tarski style declarative semantics that allows them to be conceived as sublanguages of predicate logic [BL84]. The basic reasoning tasks on concepts are unsatisfiability and subsumption testing. We say that a concept is unsatisfiable if it always denotes an empty set. A concept C is subsumed by a concept D if C always denotes a subset of D.

$\mathcal{PL}_1$ is an extension of the polynomial language $\mathcal{FL}^-$ [BL84]. The later includes conjunction of concepts $(C \sqcap D)$, universal role quantification $(\forall R.C)$ and unqualified existential role quantification $(\exists R)$, where C and D are concepts and R is a role (a role corresponds to the notion of an attribute). Concepts are built up according to the following rule:

$$C, D \longrightarrow A \mid C \sqcap D \mid \forall R.C \mid \exists R$$

An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ consists of a domain $\Delta^{\mathcal{I}}$ and a function $.^{\mathcal{I}}$ that maps every concept to a subset of $\Delta^{\mathcal{I}}$ and every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ such that the following equations are satisfied:

$$\begin{aligned}
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} | \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists R)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} | \exists b.(a,b) \in R^{\mathcal{I}}\}.
\end{aligned}$$

The class of objects denoted by the *conjunction* $(C \sqcap D)$ of two concepts in the first equation, is equal to the intersection of the set contents of the classes denoted by each concept. The *universal role quantification* $(\forall R.C)$ construct in the second equation, denotes the class of objects for which each role, R, is a concept C (e.g., each *child* is a *doctor*). And finally, the *existential role quantification* $(\exists R)$ construct denotes the class of objects that have at least one role, R (e.g., at least one *child*).

An interpretation $\mathcal{I}$ is a *model* for a concept C if $C^{\mathcal{I}}$ is nonempty; C is subsumed by D if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$. A concept is *satisfiable* if it has a model and *unsatisfiable* otherwise. Other constructs which have also been considered in the literature include:

- the empty $(\bot^{\mathcal{I}} = \emptyset)$ and universal $(\top^{\mathcal{I}} = \Delta^{\mathcal{I}})$ concepts;

- negation of concepts: $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}}/C^{\mathcal{I}}$;
- qualified existential role quantification $(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | \exists (a,b) \in R^{\mathcal{I}}.b \in C^{\mathcal{I}}\}$;
- number restrictions: $(\geq nR)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} |\ |\{b \in \Delta^{\mathcal{I}} | (a,b) \in R^{\mathcal{I}}\}| \geq n\}$, and similarly for $\leq$;
- inverse of roles: $(R^{-1})^{\mathcal{I}} = \{(a,b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} |\ (b,a) \in R^{\mathcal{I}}\}$;

$\mathcal{PL}_1$ enhances $\mathcal{FL}^-$ with the empty and universal concepts, with negation of primitive concepts, number restrictions and inverses of roles. $\mathcal{PL}_1$ has been proved to be maximally expressive with respect to subsumption, in the sense that none of the above concepts can be added without sacrificing tractability.

The Technique

The technique assumes an alphabet of variable symbols denoted x, y, z and syntactic objects of the form $x : C$ and xRy called *constraints*. An $\mathcal{I}$-assignment is a function a that maps every variable to an element of $\Delta^{\mathcal{I}}$. We say that a *satisfies* the constraint $x : C$ if $a(x) \in C^{\mathcal{I}}$ and a *satisfies* xRy if $(a(x), a(y)) \in R^{\mathcal{I}}$. A constraint c is *satisfiable* if there is an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment a such that a *satisfies* c. A *constraint system* S is a finite, nonempty set of constraints. An $\mathcal{I}$-assignment a *satisfies* S if a satisfies every constraint in S. S is *satisfiable* if there is an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment a such that a satisfies S. Therefore, it is easy to see that a concept C is satisfiable if and only if the constraint system $\{x : C\}$ is satisfiable.

To check whether a concept C is satisfiable, we start with the constraint system $S = \{x : C\}$ and in subsequent steps add constraints according to a set of propagation rules. This goes on until the system becomes *complete*, that is, until no more propagation rules are applicable. If at this point the constraint system contains a *clash* (i.e., contradiction) then the initial concept C is unsatisfiable. In the opposite case, C is satisfiable.

Subsumption now can be tested in terms of satisfiability. It is not hard for one to see that C is subsumed by D if and only if concept $C \sqcap \neg D$ is unsatisfiable (if the latter is true, C will be contained in D, or more formally $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$) . We rewrite D into an equivalent concept of the form $D_1 \sqcap D_2 \sqcap ... D_n$, where each D_i is conjunction-free (i.e., it does not contain the symbol $\sqcap$ applied to concepts). This can be done because the language contains no disjunction or qualified existential role quantification, by means of the following rewriting rule [DLNN91]:

$$\forall R.(C \sqcap D) \longrightarrow \forall R.C \ \sqcap \ \forall R.D \qquad (1)$$

Then C is subsumed by D if and only if for every conjunction-free component D_i of D, $C \sqcap \neg D_i$ is unsatisfiable. Although the concept $\neg D_i$ may contain the negation of a non-primitive concept, it can be rewritten in linear time into an equivalent *simple* concept (i.e. a concept where negation occurs only in front of primitive concepts) by using suitable rewriting rules, e.g.:

$$\neg \forall R.C = \exists . \neg C \qquad (2)$$

We say that xRy *holds* in a constraint system S if:

- R is a primitive role P and $xPy \in S$;
- R is P^{-1} and $yPx \in S$.

To check if a constraint system, S, is satisfiable we first turn S into a complete constraint system. As previously described, this is done by adding constraints to S according to a set of propagation rules until none of the propagation rules is applicable. The propagation rules determine the set constraints that are added to the initial constraint system, S, when some specific conditions hold. For instance, the first of the propagation rules presented below, states that if an object is in a concept denoting the conjunction of two other concepts (e.g., x : EMPLOYEE $\sqcap$ STUDENT), then it also has to be in each of these concepts separately (i.e., it is an EMPLOYEE and it is also a STUDENT). This information is then appended to the initial constraint system. Therefore, in a sense, the propagation rules are used to find all the information that can be used to describe the initial constraint system, that is, all the constraints that the initial constraint system implies.

The propagation rules for testing subsumption in $\mathcal{PL}_1$ are:

1. $S \ \rightarrow_{\sqcap} \ \{x : C_1, x : C_2\} \cup S$
   if $x : C_1 \sqcap C_2$ is in S and either $x : C_1$ or $x : C_2$ is not in S

2. $S \ \rightarrow_{\forall} \ \{y : C\} \cup S$
   if $x : \forall R.C$ is in S, xRy holds in S and $y : C$ is not in S

3. $S \ \rightarrow_{-1} \ \{yPx\} \cup S$
   if $xP^{-1}y$ is in S and yPx is not in S

4. $S \ \rightarrow_{\leq} [y/z]S$
   if $x : (\leq 1R)$ is in S, xRy and xRz hold in S with $z \neq y$, $[y/z]$S is obtained from S by replacing y with z

5. $S \ \rightarrow_{\leq} \ \{xRy\} \cup S$
   if $x : (\geq nR)$ is in S, with $n > 0$, y is a new variable, and there is no such z that xRz holds in S

6. $S \ \rightarrow_{\exists} \ \{xRy, y : C\} \cup S$
   if $x : \exists R.C$ is in S, y is a new variable, and there is no z such that both xRz holds in S and $z : C$ is in S

The above rules are continuously applied to the constraint system until no more rules can be applied. The constraint system is then complete. The next step is to check if the complete constraint system contains a clash. A clash is a set of constraints of one of the forms:

1. $\{x : \bot\}$;
2. $\{x : A, x : \neg A\}$;
3. $\{x : (\geq nR), x : (\leq mR)\}$, with $n > m$;
4. $\{x : (\leq 0P), xPy\}$;
5. $\{x : (\leq 0P^{-1}), xPy\}$.

If the complete constraint system contains a clash, then subsumption is true. If not, then subsumption is false. The computation of the completion of a constraint system $\{x : C \sqcap \neg D_i\}$ takes polynomial time [DLNN91] and, hence, subsumption in $\mathcal{PL}_1$ can be solved in polynomial time.

Adapting the Technique

In the following lines, we will consider the way in which the technique described above can be adapted and used to test subsumption in our model. We will be concerned with the sort of *F-logic* population queries (i.e., membership predicates) which the technique can or cannot account for. As will be demonstrated the technique accounts for a large number of population queries. We will name concepts after the elements of the set they denote. All concept names will be in upper-case letters, while roles will be capitalized. We will also use individual concepts in teletype letters to denote specific values. The usual notation will be used for *F-logic* queries.

From the interpretation of universal role quantification, we can see that this construct implies a value (or type) restriction on the role R of object a. That is, a is a $\forall R.C$ if and only if each R of a is a C. This resembles our specialization (selection) query derivation operation. To clarify this resemblance, lets attempt to translate a specialization into an equivalent concept language form. Suppose we are interested in all those people whose mother's occupation is a secretary. In *F-logic* this would be stated as:

$$p : person[mother \rightarrow M : person[occupation \rightarrow \text{"}secretary\text{"}]]$$

In $\mathcal{PL}_1$ it would be:

$$\text{PERSON} \sqcap \forall Mother.(\text{PERSON} \sqcap \forall Occupation.\texttt{SECRETARY})$$

where SECRETARY is an *individual* concept denoting a value (the string "secretary").

It is not difficult to see that $\mathcal{PL}_1$ can describe even more complex selection queries, and to our knowledge any arbitrary kind of complex queries, when only single valued attributes are used. When the specialization query also involves set valued attributes, $\mathcal{PL}_1$ can no longer be used. These type of attributes are described by qualified existential role quantification ($\exists R.C$), which does not constitute a $\mathcal{PL}_1$ construct. For instance, the set of people who own a BMW is described in *F-logic* as follows:

$$p : person[car \twoheadrightarrow \{\text{"}BMW\text{"}\}]$$

with *car* being set-valued, since we do not know how many cars this person owns, but just that one of his cars is a BMW. In concept language notation this would be expressed as:

$$\text{PERSON} \sqcap \exists Car.\texttt{BMW}$$

As previously stated, the $\exists Car.\texttt{BMW}$ construct does not belong in the language $\mathcal{PL}_1$, so we may not be able to test subsumption in polynomial time in the presence of set valued attributes. However, adding this qualified existential role quantification construct to $\mathcal{PL}_1$ would result in the $\mathcal{ALE}$ concept language [SSS88], for which satisfiability (and therefore subsumption) is proved to be NP-complete [DHL+90]. Hence, although we cannot test subsumption in polynomial time in the presence of set valued attributes, it can still be done in time exponential in the size of the query. This is not a major problem since view definition queries in general are quite short, comprising only a few conjuncts. Moreover, this is still an improvement of the initial undecidability.

Besides the selection operation, it is easy for one to see that the technique can also be used for the intersection operation. A virtual class derived as an intersection of two other classes coincides with the notion of conjunction. For example:

$$O : ta \ \leftarrow \ O : student \ \wedge \ O : employee$$

describes a new class *ta* of teaching assistants, containing all objects that belong both in *student* and *employee*. The respective concept language notation would be:

$$\text{TA} = \text{STUDENT} \sqcap \text{EMPLOYEE}.$$

The technique can, therefore, be adjusted for the intersection operation, without any additional problems.

However, the situation is not as simple as above in the case of the generalization operation. The membership predicate for a class resulting from the generalization operation was shown to be a union of *F-logic* population queries. For example, if class *ship* was defined as a generalization (union) of $militaryShip$ and $commercialShip$, then *ship* would be populated by the following *F-logic* queries:

$$O : ship \ \leftarrow \ O : militaryShip$$
$$O : ship \ \leftarrow \ O : commercialShip$$

which corresponds to a disjunction of concepts:

$$\text{SHIP} = \text{MILITARYSHIP} \sqcup \text{COMMERCIALSHIP}.$$

Unfortunately, $\mathcal{PL}_1$ accounts only for disjunction of *primitive* concepts. Therefore, the disjunction construct cannot be directly used as a translation of the union operation. However, there is a way around this problem. As we will see later on, a union, $\mathcal{Q}$, of conjunctive queries (i.e., rules that are conjunctions (logical AND) of their subgoals) is subsumed by another union of conjunctive queries, $\mathcal{R}$, if and only if every query in $\mathcal{Q}$ is subsumed by some query in $\mathcal{R}$. Since a view class derived by generalization can be thought of as being described by a union of conjunctive queries, we can use the previously described technique to accommodate generalization queries when testing for subsumption.

The additional constructions that we have provided for increasing the flexibility are also handled properly. Behavioral generalization can be thought of as a special case of generalization, while on the other hand, parameterized classes are just a special case of specialization, so they can be treated similarly. Last of all, population queries that derive imaginary classes are not accounted for since these classes stand unrelated to any other class in the view. Therefore, trying to find subsumption relationships between imaginary classes and the other classes in the view schema, would not really make sense.

An Example

An example demonstrating the application of the subsumption technique is presented bellow.

Assume a view in which *person* is a primitive class and *employee* is a subclass of *person* with additional attributes *occupation*, *salary* and *empNo*. We omit the function symbols for simplicity when referring to the view classes; e.g., we write *person* instead of $view(person)$. *Person* and *employee* are imported from the base schema. Suppose we want to create two additional view classes: one for those objects in *person* (i.e., people) whose *mother* is an *employee*, and the other for those objects in *person* whose *mother* is an *employee* and more specifically a secretary. In *F-logic* these two classes would be created by the following rules:

$$O : viewClass1 \leftarrow O : person[mother \rightarrow M : employee]$$
$$O : viewClass2 \leftarrow O : person[mother \rightarrow M : employee[occupation \rightarrow \text{``}secretary\text{''}]$$

Translating these two into $\mathcal{PL}_1$ notation we get:

$$\text{VIEWCLASS1} = \text{PERSON} \sqcap \forall Mother.\text{EMPLOYEE}$$
$$\text{VIEWCLASS2} = \text{PERSON} \sqcap \forall Mother.(\text{EMPLOYEE} \sqcap \forall Occupation.\mathtt{SECRETARY})$$

It is clear that intuitively $viewClass2$ is subsumed by $viewClass1$, since the former always denotes a subset of the latter. Denoting concept VIEWCLASS1 as D and concept VIEWCLASS2 as C, we will follow step by step the computations involved in computing the subsumption relationship between D and C. That is, we will demonstrate how the output of the $Subsumes(D, C)$ is calculated according to the previously described technique.

Initially D is broken up into its conjunction-free components $D = D_1 \sqcap D_2$, where D_1= PERSON and D_2= $\forall Mother$.EMPLOYEE. After this, the completion of each of the two constraint systems S_1=$\{x : C \sqcap \neg D_1\}$ and S'_1=$\{x : C \sqcap \neg D_2\}$ is calculated by adding constraints to it according to the propagation rules.

- Completion of S_1=$\{x : C \sqcap \neg D_1\}$

  - **step1**

    By using propagation rule 1, we add constraints $x : C$ and $x : \neg D_1$ to S_1 to get:

    $S_2 =$

$= S_1 \cup \{x : C,\ x : \neg D_1\}$

$= S_1 \cup \{x$: PERSON $\sqcap\ \forall Mother.$(EMPLOYEE $\sqcap \forall Occupation.$SECRETARY) , x: $\neg$ PERSON$\}$

- **step2**

  By rule 1 on $x : C$:

  $S_3 =$

  $S_2 \cup \{x$: PERSON, $x : \forall Mother.$(EMPLOYEE $\sqcap \forall Occupation.$SECRETARY)$\}$

- **step3**:

  Concept $\forall Mother.$(EMPLOYEE $\sqcap$ $\forall Occupation.$SECRETARY) can, according to rewriting rule (1), be rewritten as

  $\forall Mother.$EMPLOYEE $\sqcap$ $\forall Mother.\forall Occupation.$SECRETARY.

  Then by applying rule 1 again, we get:

  $S_4 = S_3 \cup \{x$: $\forall Mother.$EMPLOYEE, x: $\forall Mother.\forall Occupation.$SECRETARY$\}$

S_4 is the completion of the initial constraint system S_1, since no more propagation rules can be applied to it. It is easy to notice the existence of a clash produced by two conflicting constraints. These are $\{x$: PERSON$\}$, added in **step3**, and $\{x$: $\neg$PERSON$\}$, added in **step2**. Now we can proceed to calculate the completion of S_1'=$\{x : C \sqcap \neg D_2\}$. If this also contains a clash, then concept D will indeed subsume concept C. If the completion of the latter constraint system does not contain a clash then C will not be subsumed by D.

- Completion of S_1'=$\{x : C \sqcap \neg D_2\}$

  - **step1**

    By applying rule 1 on S_1' we get:

    $S_2' = S_1' \cup \{x : C,\ x : \neg D_2\}$

  - **step2**

    Concept $\neg D_2$ is rewritten by rewriting rule (2) as $\exists Mother.\neg$ EMPLOYEE. Then by applying rule 6 to $\{x : \neg D_2\}$ we obtain:

$S'_3 = S'_2 \cup \{x(Mother)y,\ y : \neg$ EMPLOYEE$\}$

- **step3**:

  Rule 1 on constraint $x : C$ gives:

  $S'_4 =$

  $S'_3 \cup \{x$: PERSON, $x : \forall Mother.$(EMPLOYEE $\sqcap\ \forall Occupation.\texttt{SECRETARY})\}$

- **step4**:

  Rewrite the last concept (as in previous case) and apply rule 1 to get:

  $S'_5 =$

  $S'_4 \cup \{x$: $\forall Mother.$EMPLOYEE, $x : \forall Mother.\forall Occupation.\texttt{SECRETARY}\}$

- **step5**:

  Use rule 3. x: $\forall Mother.$EMPLOYEE is in the constraint system, $x(Mother)y$ holds and $\{y$: EMPLOYEE$\}$ is not in the constraint system. This adds $\{y$: EMPLOYEE$\}$ resulting in:

  $S'_6 = S'_5 \cup \{y$: EMPLOYEE$\}$

- **step6**:

  Same as in **step5**, but for constraint $\{x$: $\forall Mother.\forall Occupation.\texttt{SECRETARY}\}$ this time:

  $S'_7 = S'_6 \cup \{y$: $\forall Occupation.\texttt{SECRETARY}\}$

S'_7 is the completion of the constraint system S'_1=$\{x : C \sqcap \neg D_2\}$, and contains a clash resulting from the simultaneous existence of two conflicting constraints: $\{y$: EMPLOYEE$\}$ of **step5** and $\{y : \neg$ EMPLOYEE$\}$ of **step2**. Since both constraint systems S_1=$\{x : C \sqcap \neg D_1\}$ and S'_1=$\{x : C \sqcap \neg D_2\}$ contain a clash, C is subsumed by D and the $Subsumes(D, C)$ function returns $true$.

It is interesting to note that while $Subsumes(D, C)$ returns $true$, $Subsumes(C, D)$ will return $fail$. This is because, by following the same steps as above, C will be broken into a conjunction of three conjunction-free concepts: C_1= PERSON, C_2= $\forall Mother$.EMPLOYEE and C_3= $\forall Mother.\forall Occupation$.SECRETARY. Considering that D= PERSON $\sqcap$ $\forall Mother$.EMPLOYEE, it is easy for one to see that, while there will be clashes in constraint systems $\{x : D \sqcap \neg C_1\}$ and $\{x : D \sqcap \neg C_2\}$, there will be no clash arising in $\{x : D \sqcap \neg C_3\}$. Therefore, according to the subsumption technique C does not subsume D and our $Subsumes(C, D)$ will return $fail$.

6.2.2 A Second Approach; Relaxing the Set of Permitted Predicates

In this section we describe yet another technique for deriving subsumption relationships between classes within the context of our model. This technique is based on earlier work done on optimization of conjunctive queries in the context of the relational model [CM77, SY81, ?, Ull89]. In that framework, testing for equivalence of conjunctive queries is translated into testing containment in both directions. One conjunctive query is contained in another in the sense that every answer returned by the former is also returned by the latter. As is easy to see, the notion of containment is identical to that of subsumption. We will, therefore, be using both terms interchangeably.

Basic Notions

We start out with some basic notions, and then go on to describe the technique as devised for the relational case [Ull89]. A *conjunctive query* is a single rule consisting of a conjunction (logical AND) of subgoals. For example,

a_1: `p(X,Y) :- e(Y,X) & e(X,Z)`

is a conjunctive query, since a_1 is a single rule and the body of a_1 is a conjunction of two subgoals, $e(Y, X)$ and $e(X, Z)$.

If Q_1 and Q_2 are two conjunctive queries then Q_1 is contained in Q_2, denoted $Q_1 \subseteq Q_2$, if for every EDB, the relation for the head predicate of Q_1 is a subset of the relation for the head predicate of Q_2. We initially consider ordinary subgoals with predicates representing EDB relations. Extending to built-in subgoals with predicates denoting arithmetic comparison operators is quite straightforward.

The key to the test for containment is the notion of a *symbol mapping*. A symbol mapping, h, substitutes for the variables appearing in a conjunctive query. The value of $h(X)$ can be an arbitrary term, while h is the identity on constants, function symbols and predicates, e.g. $h(a) = a$. If Q_1 and Q_2 are conjunctive queries:

$$Q_1: \quad I :- J_1,\ J_2,\ \dots\ J_l$$
$$Q_2: \quad H :- G_1,\ G_2,\ \dots\ G_k$$

then a symbol mapping, h, is called a *containment mapping* if h turns Q_2 into Q_1. That is, $h(H) = I$ and for each $i = 1, 2, \dots, k$, there is some j such that $h(G_i) = J_j$. The query $h(Q_2)$ resulting from the containment mapping will look like Q_1 but with some subgoals possibly missing.

Containment mappings are used to determine containment of conjunctive queries. More specifically, if Q_1 and Q_2 are as above, then Q_1 is contained in Q_2, if and only if there is a containment mapping from Q_2 to Q_1 [Ull89].

An example illustrating all the above is presented next. Consider the conjunctive queries a_1 and a_2:

a_1 : `p(X,Y) :- e(Y,X) & e(X,Z).`

a_2 : `p(X,Y) :- e(Y,X) & e(Z,X) & e(X,W).`

From a_1 to a_2 we may use the containment mapping $h(X) = X$, $h(Y) = h(Z) = Y$, and $h(W) = Z$. Then $h(a_2)$ is exactly a_1, and therefore a_1 is contained in a_2 (i.e., $a_1 \subseteq a_2$). In the opposite direction (i.e., for testing $a_2 \subseteq a_1$) we can find the containment mapping g defined by $g(X) = X$, $g(Y) = Y$, and $g(Z) = W$. Then $g(a_1)$ is the same as a_2 with the middle subgoal, $e(Z, X)$, missing. We therefore conclude that $a_2 \subseteq a_1$.

The above ideas can also be extended to unions of conjunctive queries. Let $\mathcal{Q} = Q_1 \cup Q_2 \cup ...Q_n$ and $\mathcal{R} = R_1 \cup R_2 \cup ...R_m$ be two unions of conjunctive queries, all of which have a common head predicate. Then $\mathcal{Q}$ is contained in $\mathcal{R}$, denoted $\mathcal{Q} \subseteq \mathcal{R}$, if for any EDB, the relation denoting the union of relations returned by the Q_i's is a subset of the corresponding relation returned by the R_i's. To formalize this, we need to define a *containment* among unions of conjunctive queries. A containment, α, from $\mathcal{Q}$ to $\mathcal{R}$ is a map from the conjunctive queries in $\mathcal{Q}$ to those in $\mathcal{R}$, such that if $\alpha(Q_i) = R_j$, then $Q_i \subseteq R_j$ [Ull89], where the latter is a containment between two conjunctive queries (as defined above). The key idea when unions of conjunctive queries are tested for containment is that one conjunctive query is contained in a union of two or more conjunctive queries only if the former query is contained in one of the queries that comprise the latter union.

Keeping the above ideas in mind, we proceed with the presentation of the technique for testing query containment, as devised for the relational case, and then demonstrate how this technique can be translated into a subsumption technique for our object-oriented view mechanism. We will discuss the problems that arise and devise appropriate solutions. All this is done in a descriptive manner accompanied by the presentation of illustrative examples.

The technique

The technique for testing subsumption in the relational model has as follows [Ull89]. To test whether $Q_1 \subseteq Q_2$ it suffices to test for an existing containment mapping from Q_2 to

Q_1. If Q_1 and Q_2 are defined as above:

$$Q_1: \qquad I :- J_1,\ J_2,\ \ldots\ J_l$$
$$Q_2: \qquad H :- G_1,\ G_2,\ \ldots\ G_k$$

then a straightforward approach for testing containment, is to consider all possible assignments from the G's to the J's. In the worst case, all predicates correspond to the same EDB relation, and therefore, there are l^k such assignments. Then, we can test each assignment A, to see if it is are explained by some symbol mapping. That is, if $A(i) = j$, then we treat the variables of J_j as constants and term-match G_i against J_j. If the term-matching is possible, it will produce a value of $h(X)$ for each variable X appearing in G_i. If we perform the matching for all i, and get no conflicting definitions of $h(X)$ for any X, then the corresponding symbol mapping h is a containment mapping, and thus $Q_1 \subseteq Q_2$. If on the other hand, all the possible assignments A are tested and no containment mapping is found, then $Q_1 \not\subseteq Q_2$. The complexity of the above method is exponential in the size of the input, that is the length of the conjunctive queries.

It is essential to note that if the conjunctive queries have predicates corresponding to more than one EDB relations, then there will be no attempt by an assignment A to match predicates of different relations. That is, each predicate of one query is mapped only to those predicates of the other query that correspond to the same EDB relation as the former does. Assume, for example, that we have the following queries:

a_3: `q(X,Y) :- p(X,Z) & e(Z,W)`

a_4: `q(X,W) :- p(X,Y) & e(Z,Y) & e(Y,W)`

and we are searching for a containment mapping from a_4 to a_3. Then the first subgoal of a_4, $p(X,Y)$, can be term-matched only with the first subgoal of a_3, $p(X,Z)$. Similarly, the second and third subgoals of a_4, i.e. $e(Z,Y)$ and $e(Y,W)$, respectively, can be term-matched only with the second subgoal of a_3, $e(Z, W)$. No p predicate is term-matched against an e predicate and vice versa. This is due to the fact that p and e correspond to unrelated EDB relations. However, in the object-oriented case, the situation is more intricate. There, the predicates correspond to classes, which are related to one another via diverse relationships. This fact complicates the subsumption testing procedure and requires new techniques to be devised. This is what we present in the following section.

Translating to Our Model

Our main objective is to adapt the above technique for relational conjunctive query containment to conjunctive object-oriented *F-logic* queries. That is, given two *F-logic* population rules, we will try to find a mapping that will turn one rule into the other. Again we note that we are dealing only with population queries (i.e., queries that specify the population of the virtual classes), since we want to see if the *extent* of the class specified by one query is contained in the *extent* of the class specified by the other query.

In this technique, the head predicates of the *F-logic* rules play an important role, as opposed to the technique of Section 6.2.1, where only the bodies of the rules were considered in the subsumption procedure. To see the importance of the head predicates, consider the following example:

$$(R_1) \qquad P : vc_1 \leftarrow P : person \wedge P[father \rightarrow F] \wedge F[hobby \rightarrow \text{“}judo\text{”}]$$
$$(R_2) \qquad X : vc_2 \leftarrow X : person \wedge X[hobby \rightarrow \text{“}judo\text{”}]$$

are two *F-logic* derivation queries that create two virtual classes vc_1 and vc_2, respectively. Virtual class vc_1 contains all people whose fathers' hobby is judo, while vc_2 contains all people who have judo as a hobby. The technique of Section 6.2.1 would translate the *body* of the first rule into a concept $C1$ and the *body* of the second rule into a concept $C2$, where

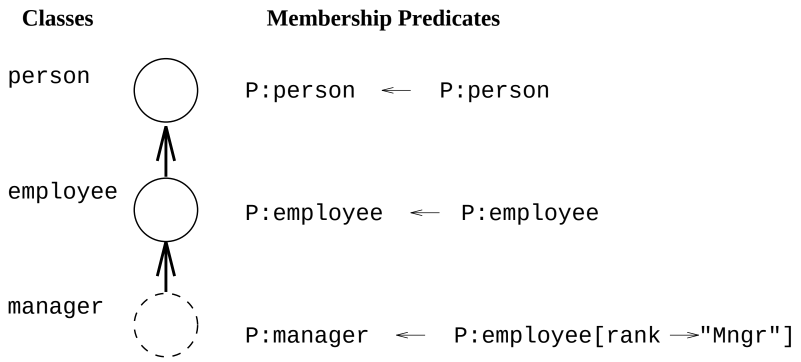


Figure 6.1: Hierarchy of Classes and Their Membership Predicates

$$\text{C1} = \text{PERSON} \sqcap \forall Father.(\text{PERSON} \sqcap \forall Hobby.\texttt{JUDO})$$

$$\text{C2} = \text{PERSON} \sqcap \forall Hobby.\texttt{JUDO}$$

It would then try to find a subsumption relationship between the two concepts; the answer would correctly be $fail$. However, when we use the containment mapping technique for subsumption, we can no longer consider only the bodies of the rule. Doing so would result in false output from the $Subsumes()$ function. In the above example, if we only consider the bodies of the rules, we can find a symbol mapping $h(X) = F$ which is a containment mapping from R_2 to R_1. Therefore, $Subsumes(R_2, R_1)$ would return $true$, which is, however, incorrect. This is because the head variables were not taken into account. If X is mapped to F in the body, then it also has to be mapped to F in the head. But then the head predicates do not match, and the technique fails. Thus, head variables are important and have to be considered when searching for containment mappings.

We are now ready to translate the technique presented for the relational case into a technique that will test subsumption of *F-logic* population queries by using containment mappings. As stated earlier, the situation is complicated by the diverse relationships among classes in an object-oriented model.

EXAMPLE 6.1: Consider the view of Figure 6.1, where view classes are presented in a hierarchy, along with their membership predicates. View classes *person* and *employee* are primitive classes imported from the base schema, while *manager* is a virtual class derived by the *F-logic* query:

$$P : manager \;\leftarrow\; P : employee[rank \rightarrow \text{“}Mngr\text{”}].$$

For primitive classes, we assume the membership predicates to be tautologies (facts), while for derived classes the membership predicates are the corresponding queries used for the classes' derivation. This is depicted in Figure 6.1. The membership predicates for primitive classes *person* and *employee* are facts, while the membership predicate for virtual class *manager* is the query used for its specification.

Assume we wish to group all female managers into one class. A natural approach would be to define a new class $femaleMngr$ as follows:

$$X : femaleMngr \;\leftarrow\; X : manager[sex \rightarrow \text{“}Female\text{”}].$$

In order to classify $femaleMngr$, we use subsumption to find all its direct superclasses and subclasses, and place $femaleMngr$ right in the middle. As described in Section 5.8, the search starts from the top of the schema, attempting to discover existing subsumption relationships, between the new class and the other existing classes. However, such an attempt with class *person* (or *employee*) in Figure 6.1 fails:

$$(Q_1) \qquad P : person \leftarrow P : person$$
$$(Q_2) \qquad X : femaleMngr \;\leftarrow\; X : manager[sex \rightarrow \text{“}Female\text{”}]$$

The above queries, Q_1 and Q_2, correspond respectively to the membership predicates of classes *person* and $femaleMngr$. Note that *person* and *manager* in the bodies of Q_1 and Q_2 respectively, refer to different classes. Therefore, no containment mapping can be found between the two queries, even though intuitively we know that $person$ actually subsumes $femaleMngr$. More surprisingly, we cannot even find a subsumption relationship between classes $femaleMngr$ and $manager$. To see this, consider the respective membership predicates:

$$(Q_2) \qquad X : femaleMngr \leftarrow X : manager[sex \rightarrow \text{“}Female\text{”}], \text{ and}$$
$$(Q_3) \qquad P : manager \leftarrow P : employee \wedge P[rank \rightarrow \text{“}Mngr\text{”}].$$

As we can see, the subgoals of the two rules refer to different classes, and therefore, no assignment from subgoals of one rule to subgoals of the other rule can be found. This means that we cannot find the subsumption relationship between these two classes, even though it actually exists.

□

A Naive Solution

An initial way of dealing with the problem described in Example 6.1, is to introduce the following two restrictions:

1. Specify each virtual class in terms of the lowest possible primitive class, and
2. Start the subsumption testing not from the top class of the schema, but from the source class of the derivation,

where by *source class* we mean the class used to specify the virtual class. In an *F-logic* rule, the source class is the class appearing in a subgoal X : <*source*-class> of the rule, where X is a variable also appearing in the head. For instance, the source class for $femaleMngr$ in rule (Q_2) of Example 6.1 is the class $manager$, since $X : manager$ is in the body of the rule and X also appears in the head of the rule.

The first restriction ensures that the virtual class is specified by means of the most descriptive rule possible. To see this, consider once again Example 6.1. Under restriction 1, class $femaleMngr$ is specified as

$$(Q_4) \qquad X : femaleMngr \ \leftarrow \ X : employee[rank \rightarrow \text{“}Mngr\text{”}; sex \rightarrow \text{“}Female\text{”}],$$

which is clearly more descriptive than specifying it as

$$(Q_2) \qquad X : femaleMngr \ \leftarrow \ X : manager[sex \rightarrow \text{“}Female\text{”}]$$

(i.e., using $manager$ as the source class). Another thing that this restriction ensures is that all kinds of employees (e.g. managers, technicians, maleTechnicians) will be specified in terms of the lowest possible primitive class (i.e., employee). This means that the specifications for all the above classes will have a subgoal of the form $X : employee$. Consequently, when testing the rules of such classes for subsumption, we will be able to find assignments from subgoals of one rule to subgoals of another. Recall that the absence of such assignments was the reason for the problematic situation described in Example 6.1.

Consider, for example, classes $manager$ and $femaleMngr$ specified from source class $employee$. Under restriction 1, these classes are specified by the following rules:

$$(Q_3) \qquad P : manager \ \leftarrow \ P : employee \ \wedge \ P[rank \rightarrow \text{“}Mngr\text{”}], \text{ and}$$

$$(Q_4) \qquad X : femaleMngr \ \leftarrow \ X : employee \ \wedge \ X[rank \rightarrow \text{“}Mngr\text{”}] \\ \wedge \ X[sex \rightarrow \text{“}Female\text{”}],$$

We can now find an assignment from the subgoals of Q_3 to those of Q_4

Rule	*Subgoal*	*Subgoal*
(Q_4)	$X : employee$	$X[rank \rightarrow \text{“}Mngr\text{”}]$
(Q_3)	$P : employee$	$P[rank \rightarrow \text{“}Mngr\text{”}]$

while this was not possible in Example 6.1. For the above assignment, we can find a symbol mapping $h(P) = X$, which turns out to be a containment mapping. Therefore, $Q_4 \subseteq Q_3$ and $Subsumes(Q_3, Q_4)$ returns true.

The above shows that for a virtual class, VC, derived from a source class, C, restriction 1 ensures that assignments of subgoals will be found, when testing the specification rule of VC and the specification rules of the subclasses of C (including C) for subsumption. However, restriction 1 does not guarantee that such assignments will be found when testing the specification rule of VC and the specification rules of the superclasses of C. For example, after $femaleMngr$ is derived (under restriction 1) as

$$(Q_4) \qquad X : femaleMngr \leftarrow X : employee \wedge X[rank \rightarrow \text{“}Mngr\text{”}] \\ \wedge X[sex \rightarrow \text{“}Female\text{”}],$$

we will start the subsumption testing from the topmost class in the schema. In Figure 6.1, this class is $person$ and it is defined as

$$(Q_1) \qquad P : person \leftarrow P : person.$$

Note, however, that we cannot find an assignment from subgoals of Q_1 to subgoals of Q_4, and therefore, we will not find that $Q_4 \subseteq Q_1$, although this is true. Consequently, $femaleMngr$ will not propagate down in the hierarchy, and therefore, it will not be classified correctly.

To deal with this situation we use restriction 2 which takes advantage of the fact that $employee$ is a subclass of $person$. Since (under restriction 1) $femaleMngr$ is derived from $employee$ (by using rule Q_4) with some additional conditions ($[sex \rightarrow \text{“}Female\text{”}] \wedge [rank \rightarrow \text{“}Mngr\text{”}]$), it will be a subclass of $employee$ and therefore, a subclass of $person$ too. This means that we do not have to start the subsumption test from the topmost class of the schema (i.e., *person*). It suffices only to start the test from the source class of the derivation (i.e., *employee*). But then, restriction 1 ensures that problems like those of Example 6.1 will not occur.

Hence, by deriving the virtual class from the primitive class, and then starting the subsumption testing from this (source) primitive class, we have found a solution to the subsumption problem in the context of our model.

However, this solution is unacceptable. It is cumbersome, restrictive and moreover, not in line with our notion of a virtual class. If we were to create a virtual class of female managers who drive BMWs', we should do this in terms of class *employee*. That is, the above solution requires the user to specify the new class as

$$\begin{aligned} O : fmDriveBMW \;\leftarrow\; & O : employee \;\wedge\; O[rank \rightarrow \text{“}Mngr\text{”}] \\ & \wedge\; O[sex \rightarrow \text{“}Female\text{”}] \;\wedge\; O[car \rightarrow \text{“}BMW\text{”}] \end{aligned}$$

instead of the more natural

$$O : fmDriveBMW \leftarrow O : femaleMngr \wedge O[car \rightarrow \text{“}BMW\text{”}].$$

Also note that the first of the two derivation queries has a greater number of F-molecules, and this number tends to grow as we move further down in the hierarchy specifying more and more virtual classes.

Even if we were to consider these problems as superficial, the previous solution is still unacceptable, due to the way it treats virtual classes. That is, virtual classes derived within the view, cannot be treated as any other class, since they cannot serve as source classes for the derivation of new virtual classes. All classes have to be derived from primitive classes, which restricts the power and flexibility of the view mechanism.

We must, therefore, devise a more efficient solution, which will not restrict the user nor will it limit the flexibility of our mechanism. We deal with this issue in the sequel.

An Efficient Solution

As previously mentioned, the problems encountered when testing subsumption in the object-oriented case, are caused by the existence of diverse relationships among classes. The previous technique introduced restrictions in order to capitalize from these relationships (e.g., restriction 2). However, these restrictions resulted in an unacceptable solution. Therefore, what is needed is a technique that will take advantage of the diverse relationships existing among classes, without restricting the flexibility of the view mechanism or the user's convenience.

In what follows, we present an efficient subsumption technique that, based on the class relationships, unfolds the definition of the virtual classes. That is, the definition of a virtual class is expanded to become more descriptive, and moreover, this is done in an efficient manner. The expansion of the definition of a virtual class is performed by adding additional membership predicates to the original membership predicates of this class. We call these new membership predicates *supplementary*. The virtual class will then be described by the **union** of its original and its supplementary membership predicates. All membership predicates in such a union are equivalent, in the sense that they describe the same group of objects.

EXAMPLE 6.2: The class $femaleMngr$ of female managers will now be described by the union of the following predicates:

$$p_1: \quad X : femaleMngr \leftarrow X : femaleMngr$$
$$p_2: \quad X : femaleMngr \leftarrow X : manager[sex \rightarrow \text{“}Female\text{”}]$$
$$p_3: \quad X : femaleMngr \leftarrow X : employee \wedge X[rank \rightarrow \text{“}Mngr\text{”}]$$

$$\wedge\ X[sex \rightarrow \text{“}Female\text{”}].$$

The new description for class $femaleMngr$ states that a female manager is a female manager, **and** a manager whose sex is “$Female$”, **and** an employee whose sex is “$Female$” and rank is “$Mngr$”.

□

As we will show, the addition of the supplementary predicates and consequently the expansion of a virtual class's description are the key points of our subsumption technique. The supplementary predicates for each class are derived by using the following two steps of the *supplementary predicate derivation procedure*:

1. Each view class VC must have one membership predicate consisting of a single *is-a assertion* of the form $O : VC$. In the case of primitive classes (e.g., $person$ in Figure 6.2) this is the original membership predicate. In the case of a virtual class (e.g., $manager$) this will be a supplementary predicate added to the class's definition.

2. For every membership predicate P_{VC} of a virtual class VC, we do the following:

    If P_{VC} has an *is-a assertion* of the form $O : C$ ($C \neq VC$), then, for each predicate P_C of C, we create a supplementary predicate P_{VC}' for VC that is similar to P_{VC}, but with $O : C$ replaced by O : (body of) P_C.

   This procedure continues until no new predicate for VC can be derived. Each is-a assertion $O : C$ will create at most $|\bigcup P_C|$ supplementary predicates for VC, where $\bigcup P_C$ is the union of membership predicates of class C.

We clarify the above procedure, by means of the following example.

EXAMPLE 6.3: In Figure 6.2 we have again our well-known classes, along with their membership predicates. No additional predicates can be derived for classes $person$ and

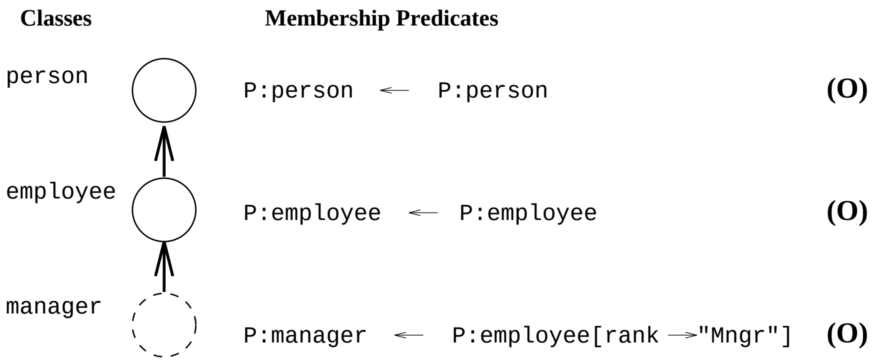


Figure 6.2: Hierarchy of Classes and Their Original Membership Predicates

employee. However, by step 1 above, we can derive an additional predicate $P : manager$ for class *manager*. This is shown in Figure 6.3. The new supplementary predicate for class *manager* is labeled with **(S)** (label **(O)** denotes original predicates). The description of *manager* now is the union of its two membership predicates.

Suppose we also derive a new virtual class $femaleMngr$ as follows:

$$P : femaleMngr \ \leftarrow \ P : manager[sex \rightarrow\text{“}Female\text{”}].$$

This definition will serve as the original membership predicate of $femaleMngr$ as depicted in Figure 6.3. The definition of $femaleMngr$ will then be expanded according to steps 1 and 2 of the supplementary predicates derivation procedure. The resulting expanded definition of $femaleMngr$ is presented in Figure 6.4. According to step 2, we replace the $P : manager$ subgoal that is in the body of the original membership predicate of $femaleMngr$ with other is-a assertions, in order to produce additional predicates. We first replace the $P : manager$ subgoal of $femaleMngr$'s original predicate with P : (body of) $PO_{manager}$, where $PO_{manager}$ is the original predicate of class manager. This creates a new membership predicate

$$P : femaleMngr \ \leftarrow \mathbf{P : employee[rank \rightarrow \text{“Mngr”}};sex \rightarrow\text{“}Female\text{”}]$$

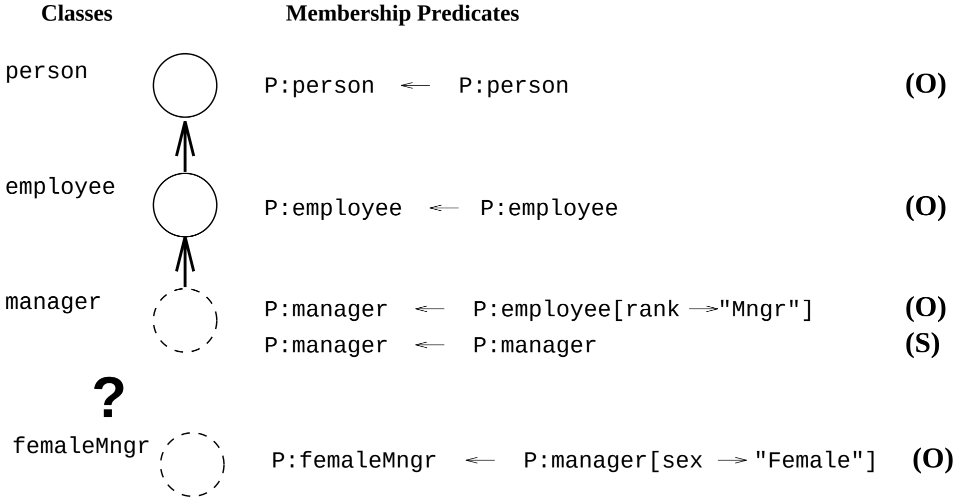


Figure 6.3: Expanding the Definition of Class *manager*

where the boldface portion is the P : (body of) $PO_{manager}$ part. Step 2 cannot create any more new membership predicates for $femaleMngr$. However, by step 1, predicate $P : femaleMngr$ is also added. Class $femaleMngr$ is now described by the union of its three membership predicates.

□

In the above fashion, after a virtual class is derived, its definition is extended by the addition of supplementary predicates. The extended definition will be the one used, when testing for subsumption between the virtual class and the other classes of the hierarchy.

The (single) predicate added by step 1, ensures that classes derived from a non-primitive view class, will be "comparable" to this class. For instance, consider class $femaleMngr$ derived as a specialization of class $manager$ in the following way:

$$p_1 : \qquad P : femaleMngr \leftarrow P : manager[sex \rightarrow \text{"}Female\text{"}].$$

If $manager$ was only described by its original predicate (i.e., $p : manager \leftarrow p : employee[rank \rightarrow \text{"}Mngr\text{"}]$), then subsumption testing would not be possible between

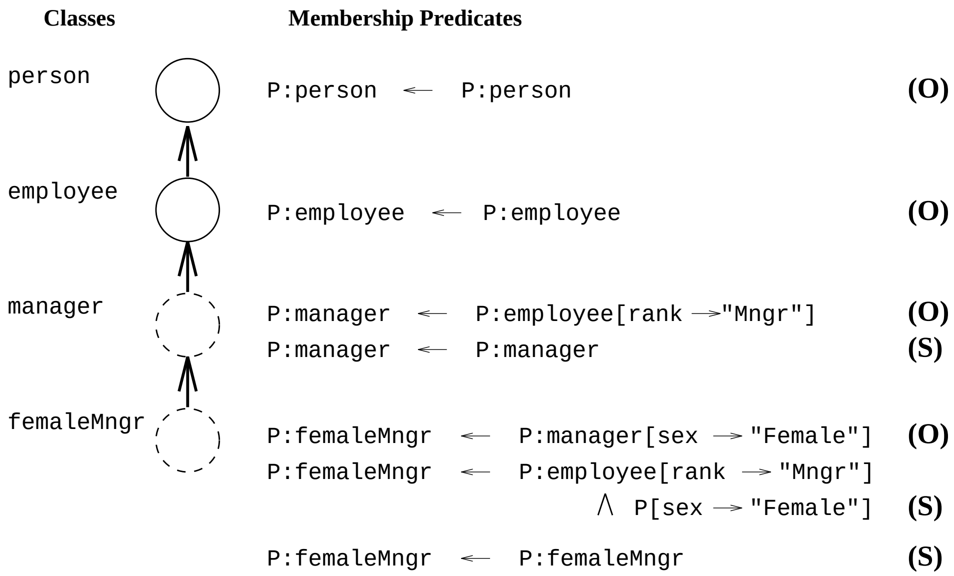


Figure 6.4: Expanding the Definition of Class $femaleMngr$

classes $manager$ and $femaleMngr$ (Example 6.1). On the other hand, using step 1 to expand the definition of class $manager$ with the additional predicate:

$$p_2: \qquad P : manager \;\leftarrow\; P : manager$$

makes subsumption testing possible. That is, we can find a containment mapping, $h(P) = P$, that turns p_2 into p_1 (with subgoal $P[sex \rightarrow$"$Female$"$]$ missing). Therefore, $p_1 \subseteq p_2$ and $Subsumes(p_2, p_1)$ returns $true$.

On the other hand, the supplementary predicates added by step 2, ensure that a virtual class derived from a primitive source class, will move as far down in the hierarchy as it should. For example, consider Figure 6.5, where only original predicates are used to describe the view classes. Specifying a new class $fmDriveBMW$ as

$$\begin{aligned} p_3: \qquad P : fmDriveBMW \;\leftarrow\;& P : employee \;\wedge\; P[rank \rightarrow \text{“}Mngr\text{”}] \\ &\wedge\; P[sex \rightarrow \text{“}Female\text{”}] \;\wedge\; P[car \rightarrow \text{“}BMW\text{”}] \end{aligned}$$

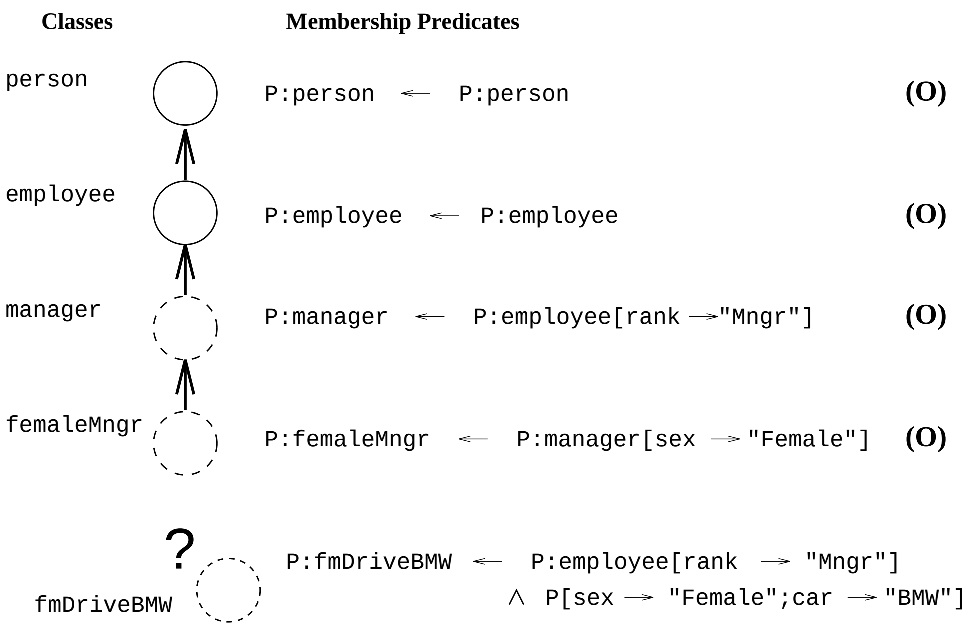


Figure 6.5: Deriving Class $fmDriveBMW$

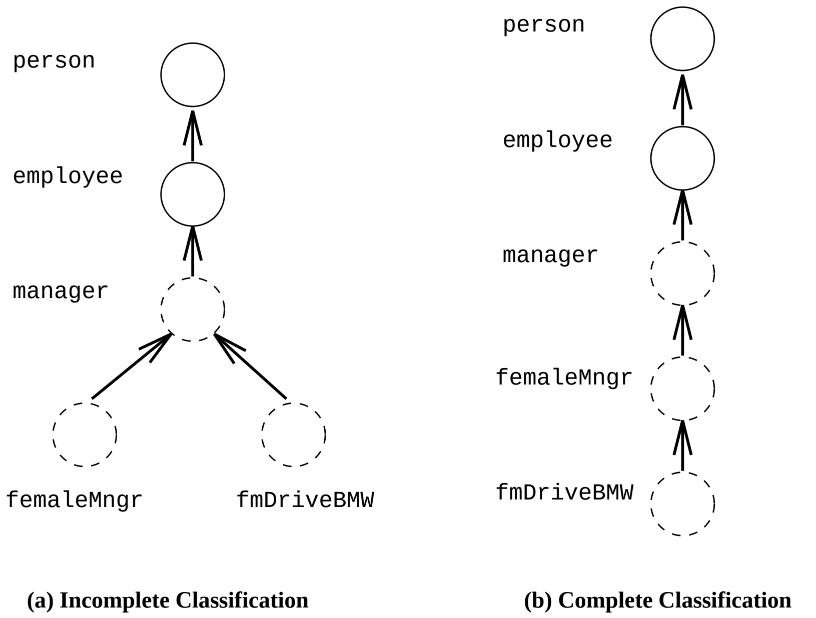


Figure 6.6: Placing Class $fmDriveBMW$ in the Hierarchy

and testing for subsumption starting at the source class *employee*, will correctly get the new class beyond *employee* and *manager*, but not by *femaleMngr*, even though it should. This means that *fmDriveBMW* will be placed directly under *manager* in the hierarchy (Figure 6.6(a)). This placement is correct, but it is not the most informative. On the other hand, by using step 2 to expand the definition of *femaleMngr*, we get the supplementary predicate:

$$p_4 : \qquad P : femaleMngr \leftarrow P : employee \wedge P[rank \rightarrow \text{“}Mngr\text{”}] \wedge P[sex \rightarrow \text{“}Female\text{”}].$$

Then the containment mapping $h(P) = P$ turns p_4 into p_3 (with subgoal $P[car \rightarrow \text{“}BMW\text{”}]$ missing). This means that *femaleMngr* subsumes *fmDriveBMW*, and therefore, *fmDriveBMW* is placed under *femaleMngr* in the class hierarchy ((Figure 6.6(b)). This placement is clearly more informative than the one in Figure 6.6(a), where the class definitions are not extended.

As we mentioned earlier, the extended definition of a class is the union of all the membership predicates that describe this class within the database context. Therefore, testing for subsumption between two classes that have extended definitions is translated into testing for containment between unions of conjunctive queries. Such a test has the same complexity as the complexity of testing containment between two conjunctive queries. That is, exponential in the size of the queries [Ull89].

Placing a derived class VC in the hierarchy can now be summarized as follows. The *F-logic* rule used to specify VC serves as VC's original predicate. This predicate comprises VC's initial definition. This definition is then extended by using steps 1 and 2 of the supplementary predicate definition procedure to generate VC's supplementary predicates. VC's new definition will be the union of all its membership predicates (original and supplementary). Then, in order to classify VC, we search for the subsumption relationships between VC and the other existing classes. The search starts from VC's source class, SC. We test for containment of VC's definition in the definition of SC and vice versa. The classification process then proceeds as described in Section 5.8. The interesting thing to note is that only the definition of the virtual class needs to be extended. The classes in the hierarchy need only have their initial definitions (i.e., their original predicates). After the virtual class is classified, we abandon its extended definition and the class is thereon described by its initial definition. this minimizes the storage requirements for the extended definitions.

Up to now, we have mainly focused on virtual classes derived in a top-down fashion, i.e., as specializations of source classes. In this case, $Subsumes(<source$-class$>, <new$-class$>)$ is true and the *new*-class travels downwards in the class hierarchy. However, classes can also be derived in a bottom-up fashion, e.g. as unions of source classes. This case is studied in the sequel.

Consider a virtual class VC derived as a generalization of two source classes $C1$ and $C2$. The population query for VC will be a union $\cup_{VC}$ of two *F-logic* queries:

$$\begin{array}{lcl} O : VC & \leftarrow & O : C1 \\ O : VC & \leftarrow & O : C2. \end{array}$$

This union will then be expanded by respectively extending each of the two queries that comprise it. the extended (union) definition will then be tested for containment with the unions of membership predicates of $C1$ and $C2$, in the usual way. The possible subsumption relationships are described in the following table:

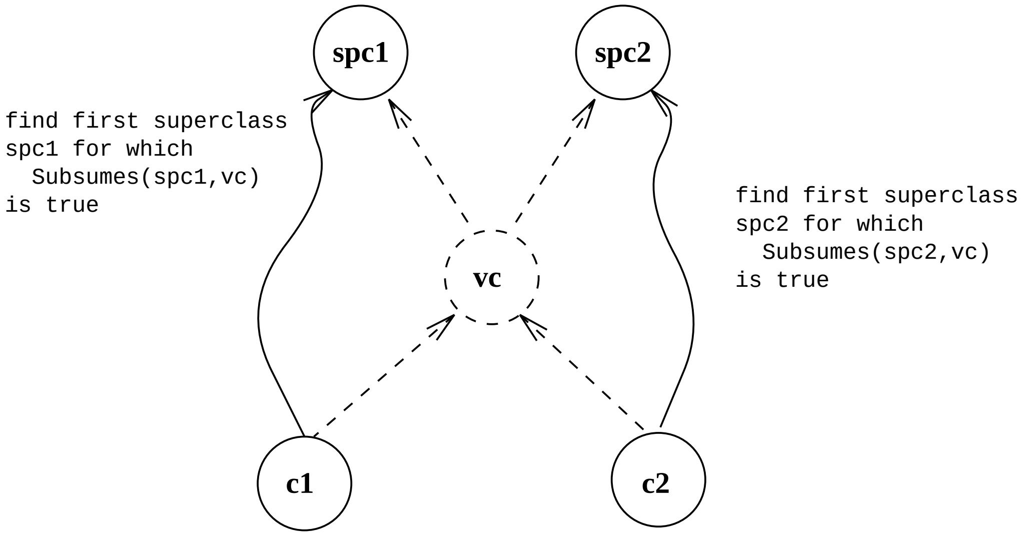


Figure 6.7: Placing a Generalization Class in the Schema

SUBSUMPTION RELATIONSHIPS	$C2$ *is-a** $C1$ $VC = C1$	$C1$ *is-a** $C2$ $VC = C2$	$C1, C2$ *not is-a* related* $C1$ *is-a* VC, $C2$ *is-a* VC
$Subsumes(VC, C1)$	*true*	*true*	*true*
$Subsumes(VC, C2)$	*true*	*true*	*true*
$Subsumes(C1, VC)$	*true*	*false*	*false*
$Subsumes(C2, VC)$	*false*	*true*	*false*

The second and third columns describe special cases of the general case described in the fourth column. Here, we consider only the general case. After the subsumption relationships are derived, VC will be placed in the hierarchy as indicated in Figure 6.7.

After VC is integrated in the hierarchy as a superclass of the source classes C_1 and C_2, what remains is to determine the superclasses of VC. To find the superclasses of the new class VC, we move upwards starting from each source classes, until we find a superclass of the source class that subsumes VC. The is-a arcs are added as displayed in the figure. If $spC1$ and/or $spC2$ are direct superclasses of $C1$ and/or $C2$ respectively, then the existing is-a arcs $C1$ *is-a* $spC1$ and/or $C2$ *is-a* $spC2$ are deleted, since they have become redundant. If there exist no classes $spC1$ and/or $spC2$ that subsume VC, this does not constitute a problem, since *F-logic* does not require classes to form a lattice.

Properties can still be inherited properly from $C1$ and $C2$ to VC, by *upward inheritance* [AB91] as described in section 3.3.4.

With the above remarks, we conclude the presentation of the second technique for testing subsumption in the context of our model. This technique tests subsumption with complexity exponential to the size of the input queries. This can not be considered a serious disadvantage if we keep in mind the small length of the input queries. The queries used to specify new classes typically involve only a few conjuncts. Therefore, exponential time with respect to the query length may still be smaller than, for instance, linear, quadratic or some higher order polynomial time with respect to the size of the database. Furthermore, the technique works for a big selection of queries (i.e., all conjunctive queries), and thus, does not restrict the expressiveness of the class derivation language. One more thing that should be noticed, is that the technique we present here is compatible to the containment mapping technique devise for the relational case. That is, the way two queries are tested for containment, remains the same. What we have devised is actually a technique that extends the definition of a virtual class in order to describe this class in all possible ways. As demonstrated earlier, this technique for subsumption results in a complete and maximally informative classification procedure. The compatibility of our technique with that devised for the relational model, allows our technique to be extended in a natural way whenever the technique for the relational case is extended, e.g., to include arithmetic comparison predicates or logical recursions [Ull89].

Chapter 7

Conclusions and Future Work

7.1 Contributions

In the preceding chapters, we presented an approach to supporting multiple views of an object-oriented database schema. In this approach, a view is treated as a database that imports all its data from other databases. Given a database schema, our approach allows for the derivation of multiple views on this schema. We define a view to also be a schema, rather than just individual virtual classes. Besides being more natural, this approach is also powerful, since views can be derived by restructuring both the generalization and the property decomposition hierarchies of the base schema. In addition, our mechanism fulfills all the specifications for view mechanisms, that were presented in Chapter 3. More specifically:

- The object-oriented views that are created by our methodology are similar to, and behave like regular object schemas. A view is just a database, but with no proper data of its own. In this way, new views can be defined on top of existing views.

- Views can easily be specified by non-expert users. The query language that we provide for the derivation of virtual classes is quite descriptive, and the whole process is clear and comprehensible.

- The user is relieved of tedious and mundane tasks. When defining a view, the user specifies the population of the virtual classes. The system then derives the type and behavior of the new class. The new class is then automatically placed into the class hierarchy.

- The flexibility and power of the view mechanism are enhanced by the introduction of new features. Features such as behavioral generalization and parameterized classes add to the users' convenience and to the overall performance of our mechanism.

The view specification procedure in our approach can be thought of being comprised of two tasks. The first task, namely the derivation of the *virtual* classes, is performed by the programmer, while the second task, which is the classification of the new classes into the hierarchy, is performed automatically by the system.

For the first task, we have provided a query-like class derivation language. The operations performed by this language are formally defined, and illustrative examples have been used to clarify the whole procedure. For the second task, we support the automatic integration of classes into the hierarchy, instead of a manual placement by the view definer. By doing so, not only do we prevent inconsistencies resulting from human error, but we also simplify the job of the view definer. We, therefore, make it easier for non-database experts to create application-specific views on their own.

For the classification of the virtual classes, we use a general algorithm that is quite common in the literature. That is, when a virtual class VC is derived, we find all those classes that VC subsumes, as well as all those classes that subsume VC. VC is then placed above the former and under the latter. The main task in this classification process is carried out by the $Subsumes()$ function, which derives subsumption relationships among classes. The complexity of this function strongly affects the overall complexity of the whole classification algorithm. Efficient subsumption techniques are, therefore, a necessity.

We have devised two such techniques. The first adopts earlier work done on this issue in the context of concept languages [DLNN91]. This technique is of complexity polynomial in the size of the input queries, i.e., the queries that describe the classes that are tested for subsumption. Low complexity is achieved by restricting the set of permitted queries for class derivation. The second technique is based on a technique for testing containment of conjunctive queries in the context of the relational model [Ull89]. We have modified the technique in order to make it suitable for the object-oriented model. Our technique accounts for all conjunctive queries. This expressiveness is accomplished with a complexity that is exponential to the size of the input queries. Considering that the virtual class derivation queries are generally of small size (compared to the size of a database), we can conclude that exponential complexity does not constitute a major problem.

We have given a translation of our view definition language into *F-logic*. For each view definition command of Chapter 3, we give an equivalent *F-logic* rule in Chapter 4. The stored information of the database is represented by the database facts and the general class information, as displayed in Figure 4.1.

Implementation issues and our solutions, have been extensively covered. We create a view by importing data from a database. The imported data is generally a portion of the base schema, which can be restructured by hiding and/or redefining properties, and by creating new view classes. These new *virtual* classes are derived from other classes existing in the specific view, and are then classified within the class hierarchy.

7.2 Future Work

While we have given a spherical presentation of the issues underlying the object-oriented view creation process and provided solutions to them, there are additional issues, that require further examination. Allowing, for example, an object instance to simultaneously be a member of many classes means that this object can take on different types [AB91, SLT91]. This immediately raises several implementation issues, as for instance, the development of efficient strategies for method resolution.

Other important database issues for future work, include query processing techniques on views, materialized views and view updates. When updates are considered and when imaginary objects are not allowed, updates can propagate naturally [SLT91]. In our case, however, where additional functionality is provided by permitting the generation of new objects, the issue of view updates requires a further examination. Generated objects result from the assignment of object identifiers to tuples of values. As a result of an update, two tuples can become identical. The question then is whether the two objects should be merged into one. This would lead to *object merging*. Similarly, view updates can lead to *object splitting*.

Another area of research concerns the derivation of the subsumption relationships between classes. As discussed in Chapter 6, the classification problem for object-oriented models is undecidable since it may involve the comparison of arbitrary functions and predicates. Therefore, the development of efficient $Subsumes()$ functions for existing object models should be further investigated. In our case, it would be interesting to search for special cases for which our second subsumption technique, based on the containment of conjunctive queries, could be reduced to linear complexity.

Finally, designing graphical user interfaces for incremental view definition also seems to be an interesting issue for future consideration. Such an interface would make it easier for non-database experts to define application-specific views on their own in application domains such as computer aided design (CAD) and software engineering environments.

Bibliography

[AB91] S. Abiteboul and A.J. Bonner. Objects and Views. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 238–247, Denver, Colorado, 1991.

[AFD+89] M. Atkinson, F.Bancilhon, D.DeWitt, K.Dittrich, D.Maier, and S.Zdonik. The Object-Oriented Database System Manifesto. In *Proc. 1st International DOOD Conf.*, Kyoto, Japan, 1989.

[AK89] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, pages 159–173, Portland, 1989.

[Ban88] F. Bancilhon. Object-Oriented Database Systems. In *Proc. 8th ACM Symposium on Principles of Database Systems.*, pages 152–162, 1988.

[BCG+87] J. Banerjee, H.T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim. Data Model Issues for Object-Oriented Applications. *Transactions on Office Information Systems*, 5(1):3–26, 1987.

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In *Proc. Int. Conf. on Extended Data Base Technology*, pages 136–152, 1992.

[BL84] R.J Brachman and H.J Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In *Proc. 4th National Conference on Artificial Intelligence.*, pages 34–37, 1984.

[BM87] M.L Brodie and J. Mylopoulos. *(eds). On Knowledge Base Management Systems.* Springer-Verlag, 1987.

[BMS84] M.L Brodie, J. Mylopoulos, and J.W Schmidt. *(eds). On Conceptual Modelling.* Springer-Verlag, 1984.

[BS81] F. Bancilhon and N. Spyratos. Update Semantics in Relational Views. *ACM Trans. on Database Systems*, 6(4):557–575, 1981.

[CM77] A.K. Chandra and P.M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. In *Proc. Ninth Annual Symp. on the Theory of Computing*, pages 77–90, 1977.

[Cod70] E.F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Dat90] C.J. Date. *An Introduction to Database Systems.* Vol. 1, Fifth ed., Addison-Wesley, 1990.

[DB82] U. Dayal and P.A. Bernstein. On Correct Translation of Update Operations on Relational Views. *ACM Trans. on Database Systems*, 7(3):381–416, 1982.

[DHL+90] F.M. Donini, B. Hollunder, A. Lanzerini, A.M. Spaccamela, D. Nardi, and W. Nutt. The Complexity of Existential Quantification in Concept Languages. Technical report, DFKI, Kaiserslautern Germany, 1990.

[DLNN91] F.M Donini, M. Lanzerini, D. Nardi, and W. Nutt. Tractable Concept Languages. *International Joint Conference on Artificial Intelligence (IJCAI'91*, pages 458–463, 1991.

[HZ90] S. Heiler and S.B. Zdonik. Schema Virtualization in Object-Oriented Databases. In *Proc. IEEE Data Eng. Conf.*, pages 86–93, 1990.

[KC86] S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proc. of the International Conference of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'86)*, pages 406–416, 1986.

[Kim89] W. Kim. A Model of Queries in Object-Oriented Databases. In *Proc., VLDB 1989*, pages 423–432, 1989.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[KLW93] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 93-06, SUNY, at Stony Brook, 1993.

[Mot87] A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7):785–798, 1987.

[MS87] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In *Research Directions in Object-Oriented Programming., B.Shrier and P.Wegner, (Eds.), MIT Press*, 1987.

[MS89] M. Missikoff and M. Scholl. An Algorithm for Classification Into a Lattice: Application to Type Classification. In *3rd Int'l Conf. on Foundations of Data Organization and Algorithms*, pages 64–82, 1989.

[NS90] B. Nebel and G. Smolka. Representation and Reasoning with Attributive Descriptions. In *Blasius,K.H., Hedstuck,U., Rollinger,C.R. (Eds.), Lecture Notes in Artificial Inteligence 418*, pages 112–139, 1990.

[RB92] E.A. Rundensteiner and L. Bic. Automatic View Generation in Object-Oriented Databases. Technical Report 92-15, University of Cal., Irvine, 1992.

[RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A Model of Authorization for Next Generation Database Systems. *ACM Trans. on Database Systems*, 16(1):88–131, 1991.

[Run92a] E.A. Rundensteiner. A Class Classification Algorithm and its Application For Supporting Consistent Object Views. Technical Report 92-50, University of Cal., Irvine, 1992.

[Run92b] E.A. Rundensteiner. Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc., 18th VLDB Conf.*, pages 187–198, 1992.

[SL83] J.G Schmolze and T.A Lipkis. Classification in the KL-ONE Knowledge Representation System. *International Joint Conference on Artificial Intelligence (IJCAI'83)*, 1:330–332, 1983.

[SLT91] M. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In *Proc 2nd DOOD Conf.*, pages 190–207, Germany, 1991.

[SLW88] A.P. Sheth, J.A. Larson, and E. Watkins. TAILOR, a Tool for Updating Views. In J.W.Shmidt, S.Ceri, and M. Missikofff, editors,. In *Proc. Int. Conf. on Advances in Database Technology (EDBT)*, pages 190–213. Springer Verlag, 1988.

[SS89] J.J. Shilling and P.F. Sweeney. Extending the Object-Oriented Paradigm. In *Proc. of the International Conference of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'89)*, pages 353–361, New Orleans, 1989.

[SSS88] M. Schmidt-Schaus and G. Smolka. Attributive Concept Descriptions with Unions and Complements. Technical Report SR-88-21, SEKI, FB Informatik, Univeristat Kaiserslauten, Kaiserslauten, Germany, 1988.

[SY81] Y. Sagiv and M. Yannakakis. Equivalence Among Relational Expressions With the Union and Difference Operators. *J.ACM*, 27(4):633–655, 1981.

[Thu89] M.B. Thuraisinghani. Mandatory and Discretionary Security Issues in Object-Oriented Database Systems. In *Proc. of Object-Oriented Programming Languages, Systems and Applications Conf.*, 1989.

[TYI88] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema Virtualization in Object-Oriented Databases. In *Proc. IEEE Data Eng. Conf.*, pages 23–30, 1988.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Vol. 1, Computer Science Press, 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Vol. 2, Computer Science Press, 1989.